A Framework for Developing Adaptive Service Compositions

by

Mahdi Bashari

Master of Computer Engineering, SUT, 2009
Bachelor of Computer Engineering, IAUM, 2006

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

In the Graduate Academic Unit of Faculty of Computer Science

Supervisor(s):	Weichang Du, Ph.D., Computer Science
	Ebrahim Bagheri, Ph.D., Computer Science
Examining Board:	Michael W. Fleming, Ph.D., Computer Science
	Arash Habibi Lashkari, Ph.D., Computer Science
	Donglei Du, Ph.D., Business Administration
External Examiner:	Marsha Chechik, Ph.D., Computer Science, University of Toronto

This dissertation is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

October, 2018

©Mahdi Bashari, 2019

Abstract

This thesis proposes a framework for automatic generation of self-healing service composition which can recover from functional and non-functional failures. To this end, it firstly proposes an automated method for generation of service composition which enables a user to build a service composition by selecting a set of desired features and secondly it proposes a method for adapting the generated service composition to recover autonomously from service failures or non-functional constraint violations.

The proposed service composition method uses software product line engineering concepts to build a repository of features and link them to their corresponding services. Using this repository, it uses AI planning to build a workflow of service interactions based on the requirements. It then uses concepts from partial-order-planning to optimize the generated workflow. Eventually, the generated workflow is converted to structured and executable BPEL code.

The proposed adaptation method extends the composition software product line to become a dynamic software product line. The proposed dynamic software product line is capable of re-selecting features of a running service composition to continue service with limited features to recover from a service failure or a violation of critical non-functional requirements. A method has been proposed which uses linear regression to determine the effect of features on the non-functional properties of service composition. Knowing how each feature affects non-functional requirements, a method has been proposed which reduces the problem of finding an alternate set of features which recovers service composition from service failure or non-functional requirement violation to a pseudo-boolean optimization problem, which can then be solved.

An online tool-suite realizing the proposed framework has been implemented and the usability, effectiveness, and reliability of the proposed framework have been evaluated with extensive experiments.

Dedication

This thesis is dedicated to my wife, Shirin. Without her unconditional love and support, finishing this thesis was not possible.

Acknowledgements

In the first place, I would like to thank my supervisors who effortlessly supported me through this long journey. I am highly thankful to my senior supervisor Dr. Weichang Du for his insightful comments and continuous support. I would also want to express my deepest gratitude to my co-supervisor Dr. Ebrahim Bagheri who went beyond being a supervisor and helped me as a friend in both my thesis and my life. He was there pushing me forward when I felt lost and offered me a helping hand when I needed one. I would also like to give special thanks to my doctoral committee members Dr. Michael W. Fleming and Dr. Natalia Stakhanova for taking the time to provide me with constructive feedback.

I am overwhelmed with gratitude by the support of all those who helped me in different ways so that I may accomplish my goals. First of all, I would like to thank my friend Dr. Mahdi Noorian from whom I asked for help when I needed someone to discuss an idea with or I needed someone to share my feeling. I would also want to thank my friend Javad Ghatta who helped

me with brainstorming, with my English, or when I needed to have a coffee and a good discussion. In addition, I express my heartfelt gratitude to my wonderful friends, most notably Andisheh Keikha, Dr. Zeinab Noorian, Dr. Ali Bagherpour, Dr. Amir Abbas Bazyar Lakeh, Maryam Ghorbani, Hossein Fani, Mostafa Samiee and my colleagues in Laboratory for Systems, Software, and Semantics (LS3).

I would also like to thank my mother and father for their unconditional support for my education. Although I was away from them while doing my thesis, their help and support motivated me to go forward all through my studies. They raised me to be a knowledge seeker and for that, I will be always grateful to them. I would also want to thank my brothers Mohammad Bashari and Masoud Bashari who always were there for me when I needed them.

Finally, I would like to thank my lovely wife, Shirin Ghorbani, who was there for me all through this long journey and supported me through the ups and downs. Her unconditional care and support helped me to be best of myself both in my research work and in my life.

Table of Contents

Abstract **ii**

Dedication **iv**

Acknowledgments **v**

Table of Contents **xi**

List of Tables **xii**

List of Figures **xvi**

1 Introduction **1**

1.1 Motivation 4

1.2 Research Objectives 8

1.3 Contributions 12

1.4 Thesis Organization 14

2 Background and Related Work **16**

2.1 Preliminaries 17

2.1.1 Self-adaptation 17
2.1.2 Software Product Line Engineering 21
2.1.2.1 Feature Model 22
2.1.2.2 Dynamic Software Product Lines 23
2.2 Automated Service Composition 25
2.3 Adaptation in Service Compositions 28
2.3.1 Cause of Adaptation 30
2.3.2 Goal of Adaptation 31
2.3.3 Realization of Adaptation 33
2.4 A Taxonomy for DSPLs 34
2.4.1 Properties of an Adaptation 35
2.4.2 DSPL Engineering Approaches 43
2.4.2.1 Monitoring & Analysis 44
2.4.2.2 Planning 47
2.4.2.3 Execution 60
2.4.3 The Proposed DSPL 68
2.5 Summary 69
2.6 Related Publications 70

3 Automated Construction of Service Compositions 71

3.1 Approach Overview 72
3.2 Case Study 76
3.3 Domain Models 77

3.4 Workflow Generation 88
3.5 Workflow Optimization 99
3.6 Generating BPEL Code 108
3.7 Summary 115
3.8 Related Publications 116

4 Run-time Adaptation of Service Compositions 117

4.1 Approach Overview 119
4.2 Case Study 122
4.3 Features and Non-functional Properties 125
4.4 Feature Adaptation Management 140
4.4.1 Loss of Utility 146
4.4.2 Feature Model Structural and Integrity Constraints (Constraint S) 148
4.4.3 Service Composition Precondition Constraints (Constraint E) 148
4.4.4 Service Independence Constraints (Constraint I) 150
4.4.5 Non-functional Constraints (Constraint N) 153
4.5 Summary 158
4.6 Related Publications 159

5 Tooling Support 161

5.1 Domain Design Tool 163
5.2 Service Composition Configuration Tool 165

5.3 Adaptation Management Engine and API 167
5.4 Summary 169

6 Experiments and Evaluation 170

6.1 Evaluating the Composition Approach 171
6.1.1 Workflow Generation 171
6.1.2 Workflow Optimization 175
6.1.3 BPEL Process Generation 178
6.1.4 Threats to Validity 183
6.1.4.1 Internal Validity 183
6.1.4.2 External Validity 184
6.2 Evaluating the Adaptation Approach 185
6.2.1 Effectiveness of Adaptation in Functional Failures 186
6.2.2 Robustness of Adaptation Towards Functional Failure . 192
6.2.3 Effectiveness of Regression For Estimating Non-functional Properties 194
6.2.4 Effectiveness of Feature Adaptation for Addressing Non-functional Violations 199
6.2.5 Threats to Validity 201
6.2.5.1 Internal Validity 201
6.2.5.2 External Validity 202
6.3 Summary 203

7 Conclusions **206**

7.1 Concluding Remarks 206

7.2 Future Work 210

Abbreviations **213**

Index **214**

Bibliography **241**

Vita

List of Tables

2.1 Comparison of recent approaches for automated service composition. 24

2.2 Comparison of recent approaches for self-healing in service compositions. 29

List of Figures

2.1 MAPE-K loop [77] 20

2.2 Dimensions characterizing an adaptation. 36

2.3 The taxonomy of the proposed framework. 45

3.1 The proposed two phase process for composing service. 73

3.2 A sample feature model for an order processing family. 76

3.3 Graphical representation of a possible WS-BPEL process for order processing. 78

3.4 A workflow solution satisfying requirements of feature model configuration in Figure 3.2. 83

3.5 An annotated feature model for the order processing family. . 86

3.6 Example of Planning Problem 93

3.7 Example of Planning Domain 94

3.8 Solution returned by a planner for our example problem . . . 97

3.9 (a) A service composition workflow and constituent services annotations. (b) Resulting workflow after removing edge marked with ($\star$) from the workflow (a) 104

4.1 General Overview of the proposed adaptation approach. 118
4.2 An annotated feature model for the upload image service family. 123
4.3 The context model for the upload image service family 124
4.4 Part of the service repository and annotations for the upload image service family. 125
4.5 BPEL process visualization for a possible service composition satisfying requirements of the feature model configuration in Figure 4.2. 126
4.6 **A.** Feature model annotated with contribution value of each feature to the reliability of the service composition. **B.** Reliability of the individual services in the service family **C.** An example of a dataset used for estimating the contribution value of each feature to the reliability of the service composition. . . 128
4.7 Initial, failed, and degraded state of the service composition in Figure 4.5 after failure of the *Object Detection* service. . . . 153
4.8 BPEL process visualization for an alternate solution for BPEL process in Figure 4.5 after adaptation by feature model reconfiguration as a result of the *ObjectDetection* failure. 154
4.9 Initial, failed, and degraded state of a service composition in Figure 4.5 after change in the reliability of the *Object Detection* service. 156

4.10 BPEL process visualization for an alternate solution for BPEL process in Figure 4.5 after adaptation by feature model reconfiguration as a result of the decrease in the reliability of the *ObjectDetection* service. 157

5.1 A screenshot of the Magus.Online tool suite. 162
5.2 The Domain Design Tool interface for annotating features in the feature model. 163
5.3 The Service Composition Configuration Tool interface for specifying user's requirements. 165
5.4 The Adaptation Management Engine interface for monitoring the current state of the service composition. 167

6.1 Workflow generation time in terms of service repository size. . 174
6.2 Workflow generation time in terms of feature model configuration size. 175
6.3 Workflow optimization time in terms of workflow size. 176
6.4 Time to completion in terms of workflow size. 177
6.5 Comparison between different BPEL generation methods in terms of element distribution. 178
6.6 Comparison between different BPEL generation methods in terms of generation time. 179
6.7 Possible BPEL representation of workflow in Figure 3.4. . . . 180

6.8 Feature adaptation execution time in terms of feature model configuration sizes. 186
6.9 Distribution of success versus failure of the adaptation mechanism. 187
6.10 Distribution of success versus failure mechanisms after a service failure in terms of number of failed services. 191
6.11 Distribution of the distance between the alternate feature model configuration and the failed one after service failure in terms of number of failed services. 191
6.12 R Squared of regression model for estimation in terms of number of training feature model configurations. 194
6.13 Root Mean Squared Error of the estimation method in terms of number of training feature model configurations. 194
6.14 Distribution of success versus failure mechanisms after a non-functional constraint violation. 198
6.15 The distribution of the distance between the alternate feature model configuration and the failed one after a non-functional constraint violation. 199

Chapter 1

Introduction

More and more companies make their application programming interfaces (APIs) of their products available to non-affiliated developers through online services. These services can be accessed through well-defined interfaces. The popularity of online services and the wide usage of service oriented architectures have motivated end-users to build new applications by composing existing services [23, 132, 30]. Such services describe a service which is composed of a number of other services to provide new functionalities. In such cases, these new functionalities were not available through the constituent services in the first place. Composing services enables end-users to satisfy their needed functionalities at a high-level without being concerned with details of individual services' implementation.

For a system, runtime adaptation is the process of changing the system's internal structure and hence the system's functional and non-functional properties. Runtime adaptation usually takes place in response to changes in a system, in its environment or its requirements. The goal of runtime adaptation is primarily that of maintaining or enhancing the services a system provides. A system which is able to adapt itself is called a self-adaptive system.

It has been suggested that self-adaptation should be introduced to service compositions in order to address reliability problems to which service compositions are susceptible since they have been built upon third-party services [46, 70, 87]. Firstly, a service composition is likely to fail or operate incorrectly as a result of failure or changes in implementation, or other changes on the part of its constituent services such as when one of the services goes down for maintenance. This may be likely considering that a service composition can be made of different services potentially provided by third-parties. A service composition can adapt in order to maintain a sound service. Secondly, a service composition may fail to fulfill anticipated non-functional properties as a consequence of violation of a promise made by its constituent services for their provision of their non-functional properties. In such circumstances, self-adaptation can be used in order to maintain the promised non-functional properties. For example, a data processing service composition needs to adapt in order to keep its promised response time when one of the constituent services' response time exceeds the maximum

accepted response time for that service. Although service compositions can benefit from self-adaptation, designing self-adaptive software requires expertise in the area of engineering self-adaptive systems [151] which end-users usually lack.

One way to address this shortcoming is using ideas and methods from Software Product Line Engineering (SPLE)[119]. Software Product Line Engineering is a software engineering paradigm which focuses on building a system from a repository of existing artifacts in a product family based on the requirements of a specific context with the goal of reducing development effort, time, and cost. In the Dynamic Software Product Line (DSPL) engineering [63, 20, 68] paradigm, software product line engineering models and methods for system variability management are used in order to build self-adaptive software with the goal of facilitating the development of higher quality self-adaptive systems with less effort. This thesis suggests that taking a dynamic software product line approach toward the development of self-adaptive service compositions can considerably reduce the complexity of developing these services for the end-users. This thesis also proposes a novel dynamic software product line engineering framework in order to aid the end-user in building desired self-adaptive service compositions.

The proposed framework, MAGUS, is built upon the DSPL paradigm to provide an architectural model, a process, system models, and its supporting tools to build service compositions which can adapt effectively in response to

change. In MAGUS, a self-adaptive composition development process consists of three phases. In the domain engineering phase, a domain engineer develops a service composition family which includes all the features relevant to that domain. In the composition engineering phase, a service composition is configured according to the features demanded by the end-user. In the runtime adaptation phase, a control loop monitors the operation of the deployed service composition and reconfigures it when the user's requirements are not satisfied. The control loop is anticipated within the framework and is responsible for self-adaptive properties of the service composition. Using this process, end-users will avoid getting involved in the details of adaptation realization without forsaking its advantages.

1.1 Motivation

With the widespread use of software applications in businesses, a growing number of non-experts seek to develop proprietary applications that respond to their specific needs. According to U.S. Bureau of Labor and Statistics, around 55 million Americans engage in some level of programming, of whom less than 3 million are professional programmers [80]. This has led to the burgeoning development of tools and kits which enable users to build their desirable applications. The focus in these tools and kits is on the provision of non-complex platforms for application development in specific domains. Several software applications (such as Microsoft Office and Adobe Photoshop)

provide these tools and kits in their software packages in order to enable users to develop specific functionalities in various domains. Furthermore, with the number of online services growing, more and more applications are accessible through their service APIs. These services of these applications can be combined to develop a new application specific to a user's need. Increasing interest in developing software applications by users and increasing availability of online services boost the widespread use of service compositions in the development of applications. The widespread use of services compositions necessitates supporting methods and tools for the development of service compositions.

In the following, two primary causes that motivate adopting a self-adaptive approach in service compositions are discussed using a case study. Let us consider a hotel search service composition which gets a date range, a budget limit and a list of city attractions and returns a list of hotels with information about their price, rating and are sorted based on their proximity to the user specified attractions and also have a room in the specified range for the date and within the user's budget. In terms of implementation, the service composition uses a hotel search service to locate hotels in the city and then uses a mapping service to calculate the average distance between the said hotels and those attractions. It then returns a list of results based on all parameters and shows them with the rating retrieved from a hotel rating service, sorted based on their proximity to the user's specified attractions.

- *Failure or changes in services functionality*: Online services change and become available/unavailable dynamically. These changes can affect the availability and reliability of the composed service. For the above example, the mapping service can go down or its implementation changes such that it requires an additional parameter. Such changes result in the failure of the whole service composition. The very likelihood of having these changes take place affects the reliability of a service composition. Adaptation in response to the changes toward alternative solutions can be considered as the right strategy. In the above example, the composition can adapt in response to this change by not calling the mapping service and sorting the hotels based on their ratings. This allows the service composition to continue service with limited functionality rather than completely failing.

- *Changes in services' non-functional properties*: Maintaining satisfactory non-functional properties of services can be as critical as providing functional requirement for a business. As an example, it has been observed that a website can lose 53% of its visitors if it has load time higher than 3 seconds [4]. Self-adaptation can be used to ensure a service composition satisfies the constraints over its nonfunctional properties. For example, assume that there is a constraint on the response time of the service composition and the service composition currently violates it. The service composition can adapt by removing one of

its non-critical features such as hotel ratings in order to satisfy that constraint.

There are existing approaches in the literature on the design of self-adaptive service compositions. However, these approaches are more often of little or no use to the non-expert end-user due to the required technical knowhow when attempting to design a self-adaptive service composition. The DSPL approach is now widely and properly used in the adaptation design by employing SPL models and methods. The following reasons have motivated this research to adopt a DSPL approach toward service composition self-adaptation design and implementation:

- The two-phase process of SPL reduces the effort required for developing a product by restricting it to the variants of a specific product family. This idea can be used in the service composition context where the service composition family can be designed by a domain expert while the final composition can be built by the end-user. Although this approach may limit the applicability of the possible service compositions to specific variants of the service composition family, it nonetheless facilitates the service composition development for end-users optimally and significantly.

- In the software product line engineering paradigm, reducing efforts in product configuration for specific contexts is one of the main goals. Some works in SPLE have proposed automated product derivation

methods which automatically build the final product based on the product configuration. These methods have been used in dynamic software product line approaches for self-adaptation as well as product derivation. Ideas from these methods lend themselves to the composition and self-adaption of service compositions.

- Software product line engineering has rich models for representing variability. These models are used in the DSPL context to manage the variability of self-adaptive systems at runtime. These models can be similarly used in the context of self-adaptive service compositions, which allows them to manage variability at runtime.

1.2 Research Objectives

This research in general tries to develop a method which enables users to develop service composition which can adapt autonomously in response to functional and non-functional failures. In the following, detailed objectives of this research are reviewed and the challenges in accomplishing them are discussed.

- **Enabling users to develop readily executable services based on his/her desired features.** One of the goals of this research is finding a method for enabling a user to develop desired service composition. There are existing methods for developing service compositions.

However, the existing methods have the following shortcomings which need to be addressed by the proposed method.

  - Existing approaches for automated service composition often require the user have a deep knowledge in an area related to the service composition to specify the requirement for the composed service [70, 79]. For example, the properties of goal service composition should be defined using semantic rules on the ontology representing the system in [70]. One of the challenges in achieving the above goal is developing a method which works on the feature of goal service composition as input to compose the desired service.
  - Existing approaches often represent the generated service composition abstractly. For example, the service composition is represented as a sequence of service invocations [55] or using an abstract process model such as BPMN [12]. This abstract representation cannot get executed readily and requires additional processing to be executed. One of the research challenges that is addressed in this research is to develop a service composition whose output can be readily executed.

- **Enable autonomous run-time recovery of service compositions from service failures.** Considering that the service composition often uses third-party services which can go offline or change without further

notice, one of the objectives of this work is to enable generated service composition to adapt autonomously to recover from service failure or mitigate its effect on the service composition. In the following, challenges that are going to be addressed while accomplishing this objective are reviewed.

- The existing approaches for enabling run-time adaptation require that the exact policy for reacting to failure specified. Therefore, a policy for any way that service composition can fail should be defined. This can be hard since the service composition can fail in many different ways. Consequently, creating and maintaining those policies can be hard. One of the challenges in reaching the above objective is to enable self-healing without requiring policies to be defined to do so.
- The existing approaches for runtime adaptation take an all or nothing approach in realizing self-healing. This means the adaptation mechanism tries to restore all of the functionalities of the service composition. If it is not possible to restore all of the functionalities of the failed service composition, the adaptation process fails and none of the functionalities will be provided. One of the challenges which needs to be addressed in achieving the above goal is enabling service composition to recover with partial features.

- **Enable autonomous runtime recovery of service compositions from non-functional constraint violation.** Third-party services which are used in a service composition can have unpredictable non-functional properties which make the non-functional property of the service composed of them unpredictable. One of the objectives of this work is to enable service composition to adapt to maintain acceptable non-functional performance. This work tries to address the following challenges while reaching this objective:

  - Existing works do not address both functional and non-functional failures at the same time while these two aspects of service composition are highly interrelated in practice. In this work, we try to address this shortcoming by developing a runtime adaptation mechanism that considers the effect of actions on both functional and non-functional properties of service composition when deciding about adaptation.

  - The existing work focuses on changing the way the service composition is realized to address non-functional constraint violations. This is typically done by replacing constituent services by services with the same functionality and different non-functional properties. These methods fail when a service composition with the same functionality and with a desired non-functional property cannot be composed. In this work, this shortcoming is addressed by en-

abling service composition to recover from failures by losing features which are not critical but affect non-functional properties negatively.

1.3 Contributions

The main contribution of this research will be the conceptualization and implementation of a DSPL-based approach called MAGUS. Of equal importance is the provision of supporting tools that guide end-users in building self-adaptive service compositions based on their requirements. These service compositions should be able to optimize their structures in response to changes in the environment in order to maintain desirable services. Specifically, the following are the main contributions of this research as each is enabled through the proposed framework, MAGUS:

- **A taxonomy for comparing dynamic software product line approaches:** This thesis proposes a taxonomy for classifying and comparing DSPL approaches from two distinct perspectives: adaptation properties and adaptation realization. This taxonomy relates DSPL design decisions to properties of adaptation. The proposed taxonomy therefore can be used for comparing different DSPL approaches when designing a new DSPL approach or adopting an existing DSPL approach based on the adaptation requirements in the problem domain.

- **An SPL-based automated service composition method:** This work proposes an AI planning based method for automated service composition which operates based on software product line configurations as the main input model for specifying requirements and automatically generates a workflow that satisfies the functional requirements. It further proposes a method for optimizing the created workflow by considering the concepts of *safety* and *threat* from the AI planning domain in order to inject parallelism into the generated workflow and improve its execution efficiency (e.g., reduce execution time). Consequently, it is demonstrated how the generated optimized workflow can be directly translated into WS-BPEL code that can be executed without any input from the designer. This provides the added benefit for the designer in that this approach would only require her to select the desirable features from a software product line as a result of which an executable WS-BPEL code will be generated.

- **An automated service composition adaptation method for functional and non-functional failure recovery:** This work proposes an automated failure mitigation method, which focuses on finding an alternate feature model configuration for the service composition that recovers critical functional and non-functional requirements. This method is able to find an alternate service composition to replace the failed service composition. In order to enable finding an optimal feature model configuration with desired non-functional properties at runtime, this work

proposes a method which is able to estimate the effect of each selected feature on the non-functional properties of the service composition. Additionally, the proposed method is able to update its estimates at runtime as the non-functional properties of the constituting services of the service composition change.

- **A tool-suite for developing self-adaptive service compositions and runtime management API:** The proposed methods have been implemented and added to our existing service development suite, called *MAGUS*. The tool suite allows for the specification of the product line representation of the domain, as well as the modeling of the desired functional and non-functional requirements. MAGUS will automatically generate executable BPEL code. This tool provides an API for a BPEL execution engine for continuously monitoring the generated service composition and adapting it as necessary.

1.4 Thesis Organization

The rest of this thesis is organized as follows:

- **Chapter 2 - Related Work.** This chapter presents the overview of the needed concepts in the area of service composition, software product lines and self-adaptation. Furthermore, a review of the most related research works which are related to the contributions of this

thesis is provided. Specifically, a taxonomy for comparing methods is provided and the existing work in this area is surveyed.

- **Chapter 3 - Service Composition and Optimization.** The focus of this chapter is to develop an end-to-end technique that would enable the automatic generation of composite services based on a specific configuration of the feature space that would be directly executable in WS-BPEL format.

- **Chapter 4 - Service Functional and Non-functional Adaptation.** In this chapter, an adaptation mechanism which is able to effectively maintain functional and non-functional requirements in service compositions is proposed.

- **Chapter 5 - Tool Support.** This chapter goes through three different tools that comprise the MAGUS tool-suite. This chapter describes how these tools can be used to compose self-adaptive service compositions.

- **Chapter 6 - Evaluation.** This chapter reports on the experiments performed to evaluate the practicality and usefulness of the proposed methods for composition and adaptation of service compositions.

- **Chapter 7 - Conclusion and Future Work.** This chapter provides a conclusion for this thesis and discusses potential points of improvement based on the limitations of this work.

Chapter 2

Background and Related Work

This chapter provides background on the concepts which are being used in this thesis. Then, it surveys the existing approaches for composition and adaptation of service compositions in the literature and compares the work proposed in the thesis with them. Additionally, it proposes a taxonomy for characterizing dynamic software product line engineering approaches and uses it for comparing existing dynamic software product line engineering approaches. This taxonomy is later used as the base for developing the DSPL-based framework proposed in this thesis.

2.1 Preliminaries

This section covers two main concepts that are the basis for the work in this thesis. First, it gives an introduction to runtime adaptation and how it is realized in practice. Afterwards, it provides basic concepts behind software product line engineering. Afterwards, it discusses the relation between self-adaption and software product lines and introduces dynamic software product engineering.

2.1.1 Self-adaptation

Some of the approaches to software system engineering are built upon the assumption that the requirements of the system and characteristics of the operating environment are completely known during the development period and built upon building blocks which are under control. Consequently, the approaches view the lifecycle of the software as two completely separate phases where in the development phase the system fully satisfying a user's needs based on the assumed requirements is created and used by the user in the runtime phase. However, systems are increasingly made of components which are built by different organizations (e.g., using services from third party vendors) and additionally have very dynamic operating environments (e.g., ubiquitous computing systems). In this situation, some of the software engineering approaches are ineffective considering that manual reconfiguration

and offline repair and enhancement may not be practical when the changes are frequent. Therefore, new approaches are necessary for engineering these types of systems.

In order to provide better service, software systems require behavior modification at runtime, particularly in response to users' dynamically varying needs as well as environmental constraints [114]. Such changes could be on the features of the system or on how the system delivers those features. This is referred to as *runtime adaptation*, and a system capable of deciding about and performing runtime adaptation is called a *self-adaptive* system. Developing self-adaptive systems has been a significant concern in many research areas such as mobile computing, autonomic computing, robotics and ubiquitous computing [35].

Runtime adaptation is supported in existing software systems, for example, in mobile operating systems which switch how they provide internet connectivity functionality at runtime from wireless connection to data connection when the wireless connection becomes unavailable. However, runtime adaptation often is not implemented in a systematic way in these systems. Considering the fact that runtime adaptation is a complex process [151], systematic approaches have been proposed for designing runtime adaptive systems [151].

In many implementations of runtime adaptations, the adaptation takes place by exchange of runtime entities using parameterization, inheritance,

and preprocessor directives in the application code [59, 138]. This makes the application and adaptation logic tangled together, which makes the code complex and hard to modify. Therefore, from the early efforts made in the area of systematic implementation of self-adaptive systems, the separation between parts of the system focusing on adaptation logic and application logic has been promoted [113]. Adaptation logic is concerned with selecting the best possible variant of the running system based on the current context while the application logic provides the required functionalities of the system. This practical distinction makes the development of a self-adaptive system easier and results in a more reliable system [151]. The separation is often implemented as a two-layered structure. In this structure, an adaptation management layer (which is also called a control loop) is implemented over the application layer. The adaptation management layer accommodates adaptation logic. It enables runtime reconfiguration and also provides the interfaces for monitoring and reconfiguring the system for the adaptation management layer as well as accommodating the application logic. Figure 2.1 shows the adaptation manager layer and the application layer.

The selection of the most suitable variant of the system is one of the most challenging parts of designing adaptation [64] and the focus of self-adaptive system engineering approaches is mostly on building adaptation managers for self-adaptive systems. In order to break down the concerns for building an adaptation manager, the adaptation manager is conceptually represented as a MAPE-K loop model [77]. The MAPE-K loop model is used in auto-

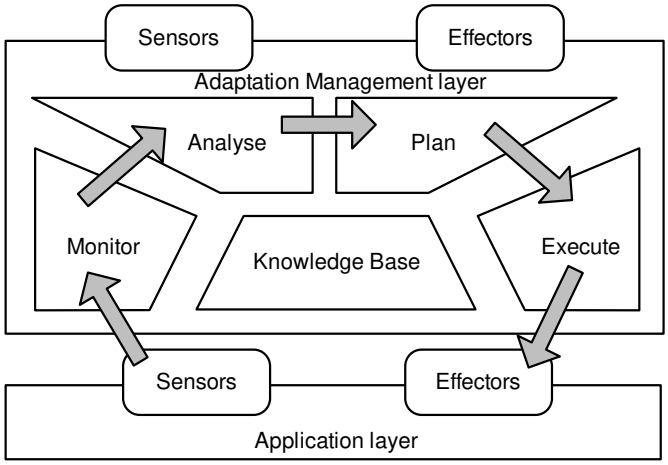


Figure 2.1: MAPE-K loop [77]

nomic computing to represent the main concerns for building an autonomic manager. An autonomic manager is responsible for handling autonomic properties of an autonomic agent in an autonomic system. MAPE consists of the initial characters of the four main steps to be taken by an autonomic manager: Monitor, Analyze, Plan, and Execute. The added K character following the hyphen stands for knowledge which is often represented by models which are used in these steps. The monitoring step is about capturing those properties which are required for deciding about the adaptation of the system. The analysis step addresses concerns regarding the process of the monitored data by considering a combination of monitored values and taking into account the monitoring history which will be useful for the planning step. The planning step decides if adaptation is essential and selects the best system variant if adaptation is necessary. After planning for the adaptation, the execution step safely executes the adaptation.

Despite the efforts in building approaches for runtime adaptive systems, the research community still tries to introduce more efficient frameworks and processes for the development of self-adaptive systems [35].

2.1.2 Software Product Line Engineering

The Software Product Line Engineering (SPLE) paradigm suggests an effective way to deal with the variability of similar products especially when satisfying requirements of different operating environments and users. This group of similar products is called a product family which is a group of products sharing common features. An example of a product family is the operating systems for mobile phones that a company produces. Although all these operating systems share common functionalities, some may offer some functionality such as Near-field communication (NFC) while others do not. Software product line engineering provides models and methods to manage such variability effectively. In software product line engineering, the development of an application that fits user needs and the operating environment relies heavily on the reuse of assets through two phases: domain engineering and application engineering.

In the domain engineering phase, those reusable assets needed for developing a product are specified and developed. These assets consist of the common parts and variation points. In the application engineering phase, the final product for the specific operating environment and user requirements

are derived using the reusable assets developed in the domain engineering phase. The binding of these assets into the target product is mostly performed at design, compile or link time, after which the software application stays the same during its lifetime.

2.1.2.1 Feature Model

There are many different techniques for representing variability in a product family. One of the most popular models for representing variability in software product line engineering is feature modeling [41, 75]. Feature models have good tool support [18] and are understandable for all stakeholders [86], which makes them a good choice for modeling variability in different problem domains. In feature models, a product family is represented using features which are increments in system functionality or to be more exact a logical unit of behavior specified by a set of functional and non-functional requirements [27].

A feature model provides a tree-like structure that shows the organization of features. In a feature model, features are hierarchically organized by structural constraints which can be typically classified as: 1) Mandatory: a feature must be included in the product if the parent feature is included; 2) Optional: a feature may or may not be included in product if the parent feature is included; 3) Alternative feature group: one and only one of the features from the feature group can be selected if the parent node is

included; 4) Or feature group: one or more features from a feature group can be included in the product if their parent feature is included. In some cases, cross-tree mutual interdependencies among the features exist; thus, additional constraints are often added to feature models and are referred to as integrity constraints such as *requires* or *excludes*.

In order to specify a valid variant of a feature model one should specify a set of features which respect structural and cross-tree constraints. Such a set of features is called a *feature model configuration* and represents a viable system.

2.1.2.2 Dynamic Software Product Lines

Software product line engineering and self-adaptive systems have much in common. The software product line paradigm is a solution for managing the variability of products in a product family. In software product line engineering, the variability of the products is captured at the domain engineering phase, and the best variant for a specific operating environment and user requirements is selected during the application engineering phase. As such, designing a self-adaptive system may be considered to be a variability management problem, where first and foremost the variability of the system is captured at design time, and then the best product variant is selected at runtime according to context requirements. The fact that both of these two paradigms are dealing with variability management, one as problem and

Approach	Requirement Specification	Composition Specification	Models	Composition Method
Hristoskova et al. [70]	OWL-S Service Description	OWLS composite service	Ontology, OWL-S Service Annotation	HTN Planning
Bertoli et al. [25]	Goal State by prepositional logic	WS-BPEL	Service representation using state transition diagram	Planning with Model Checking
Hatzi et al. [65]	Initial/Goal State	OWLS composite service	Ontology, OWL-S Service Description	Conversion to PDDL
Fujii et al. [55]	Natural Language	Sequence of Operations	Ontology, Semantic graph (representing modeling operations, context)	Inference Engine
Proposed Aproach	Feature Model	WS-BPEL	Context Ontology, FM Annotation, Service Annotation	Conversion to PDDL

Table 2.1: Comparison of recent approaches for automated service composition.

one as a solution, has motivated researchers to consider the synergy between these two paradigms in dynamic software product line (DSPL) engineering.

In dynamic software product line engineering, SPL models and techniques are used at runtime to manage and decide about the variability of self-adaptive systems. This is achieved by using SPL-inspired models and mechanisms at runtime for selecting the most suitable variant of the system and adapting the system to that variant.

2.2 Automated Service Composition

Our work is positioned among considerable other research on service composition methods. Recent surveys on these approaches can be found in [131] and [139]. Here, some of the approaches that are closely related to the approach proposed in this thesis are discussed and summarized in Table 2.1. In this table, these approaches have been compared in terms of the models they use for requirement and composition specification and linking these two and the method they use for composing the desired service. In the following, existing models and methods in each of these dimensions are discussed.

Given the fact that planners often take initial and goal states as the way to define the planning problem, adopting this approach for modeling the expected outcome of a service composition is quite intuitive. Therefore, many existing approaches specify the expected outcome using a planner input language or a model that is easily convertible to a planner input [70, 79]. For example, in [79], an XML dialect of PDDL is used to define the expected service specification. However, specification of requirements in those languages requires expert knowledge. In order to facilitate the design of service compositions in some approaches, the concrete service composition is generated from the abstract process created by the user using some GUI interface. For example in [109], users drag and drop required components of their service composition into a canvas and create the flow by connecting these components using arcs where this process is facilitated using semantic annotations

for components. Such approaches still rely on the users for designing the logic of the service interactions. Another approach used for specifying service requirements is through natural language specifications. For example in [55], an approach is proposed where a composite service is created based on a request in natural language using semantic annotation of the components. Although natural language seems an easy to use method of specifying requirements, it does not provide the user with a tangible model of functionalities, which makes it unreliable [55]. Feature models provide a tangible way to represent functionalities and have been used to represent service families.

Many existing approaches use simple models such as sequence and direct graphs to represent service composition. However, more complex functionality is often needed in the practical application of service composition. Therefore, many languages and standards have been proposed in the industry such as WS-BPEL, BPML [145], and ebXML [58]. These languages and standards provide capabilities such as complex logic support, exception handling and ability to be executed by an engine. One of the popular ways to represent service composition is OWL-S (previously DAML-S). Although main purpose of the OWL-S is to describe semantic web services, it also allows for defining a composite service which is built on other services using basic operations. OWL-S composite services are widely used for describing service composition because of its semantic description and reasoning support which facilitates composition of the service. Examples of service compositions using OWL-S are [70, 65, 79, 134]. For example in [70], existing services are

defined as OWL-S atomic services and described using OWL-S precondition and effect tags. In this method, the HTN planning approach is adopted, which uses OWL-S control constructs to define the process of a composite service which serves as the goal service. Although OWL-S semantic support facilitates composition of services, it lacks capabilities which are required in many industrial contexts such as error handling [131]. This work has adopted BPEL as a standard for defining service composition in our approach since it provides a rich language for defining business processes and it is widely used in the industry, which facilitates its possible adoption by the industry.

Planning is often considered the main activity in service composition since it figures out the workflow of the composite service. The methods used in service composition can be categorized into two groups: top-down and bottom-up [140]. In the top-down methods, the planner commences with the description of the desired composite service which it then decomposes to make it less abstract until the process reaches a concrete workflow. Moving from a more abstract workflow to a less abstract workflow is guided by domain knowledge. For example, in [98], using an ER-model an ontology representing the domain knowledge is built. The ontology is used in order to create an abstract workflow. In the bottom-up approaches, available services are connected by matchmaking methods which attempt to build desired services using AI planning approaches such as HTN [134, 70], Model Checking [25, 147], Theorem proving [121], and others [95]. Different AI planning techniques have different advantages and restrictions. For example, HTN

planning can provide more efficient composition but it requires the specification of certain decompositions. PDDL has been proposed to standardize the way a planning problem is represented which facilitates changing the planner based on input requirements. Conversion of web service composition to PDDL allows for the selection of the most appropriate planner based on the context and provides the ability to use state-of-the-art planners as they are introduced in the future. In [95], the service composition problem has also been reduced to a planning problem represented by PDDL and then solved using estimated-regression planning. However, the proposed approach requires extending PDDL in various ways. The works presented in [134] and [65] are similar to our work in the sense that they convert the problem into PDDL and use a planner to find the solution. However, our work is different in terms of how users represent their requirements and the way service composition is represented in addition to the extra steps that are proposed for workflow optimization and code generation.

2.3 Adaptation in Service Compositions

Given that services are executed in highly dynamic environments, languages which define service compositions such as BPEL provide fault-handling and rollback capabilities. However, it is the developer's responsibility to implement strategies to address failures. Some researchers have worked on extending BPEL or its execution engine in order to automate recovery from failure

Approach	Adaptation Cause		Adaptation Goal		Adaptation Realization	
	Functional	Non-functional	Functional	Non-functional	Strategy	Method
Hristoskova et al. [70]	Service Failure	–	Recovery	–	Substitution / Reorganization	AI Planning
Tan et al. [142]	Service Failure	–	Recovery	Mitigation	Reorganization	Genetic Algorithms
Angarita et al. [10]	Service Failure	–	Recovery	Mitigation	Retry / Substitution	Rule-based
Nallur et al. [107]	–	Constraint Violation	Recovery	Recovery	Substitution	Market-based Heuristics
Carzaniga et al. [31]	Service Failure	–	Recovery	–	Reorganization	Rule-based
Proposed Approach	Service Failure	Constraint Violation	Mitigation	Recovery	Reorganization	Reduction Pseudo-boolean Optimization

Table 2.2: Comparison of recent approaches for self-healing in service compositions.

[105, 137, 81]. For example in [137], an enhancement to the BPEL engine is proposed which is able to detect seven different categories of failures and perform different possible healing strategies such as re-invoking the failed service or substituting it. However, it is still the developer's responsibility to specify what strategies to adopt in response to each failure. Here, the focus is on methods which automatically recover the failed service composition.

The concept of enabling a system to automatically recover from a failure is not new. There have been efforts for enabling fault tolerance for mission-critical systems mostly through redundancy. However, the popularity of services, their modular nature as well as their dynamic operating environment have motivated the introduction of self-healing strategies. Table 2.2 shows some of the more recent works in this area (2013 and later) compared to the proposed method. In the following, the different aspects of self-healing mechanisms covered in Table 2.2 are reviewed, and similarities and differences between existing methods and the proposed method are discussed. In the following, we review existing work in the literature from three three aspects of what causes an adaptation, what is the goal of an adaptation, and how an adaptation is realized.

2.3.1 Cause of Adaptation

Most of the work in the literature has focused on service failure, which is when a service does not respond, because it is not available or has crashed.

Examples of such approaches include the work by Hristoskova et al. [70] where a failure in a service composition is addressed by substituting the failed service with a service with the same functionality using semantic matching or using AI planning to find a process fragment to replace the failed service when the substitute service cannot be found. Self-adaptation in order to address non-functional failures has also been addressed in the literature. Service compositions which are built using these methods will self-adapt when the non-functional properties for service composition is not in the acceptable range to restore it. As an example, Nallur et al. [107] propose a self-adaptive mechanism based on a market-based strategy which is able to reselect services which are used in the service composition execution process in order to restore failed non-functional constraints. Although providing both functional and non-functional requirements can play a critical role in the usability of the system, existing methods in the literature are designed such that they can only respond to one of these failure types. In the proposed method, this shortcoming is addressed by proposing a decision-making mechanism which considers both functional and non-functional requirements at the same time in planning for addressing a failure.

2.3.2 Goal of Adaptation

Although a system often fails because it cannot satisfy a requirement which is functional or non-functional, the adaptation mechanism should decide by

considering both of those requirements simultaneously since addressing a functional failure ideally should not affect the provision of non-functional properties of the service composition and vice-versa. As an example of an approach that considers both of these requirements at the same time while deciding on adaptation, the work by Angarita et al. [10] proposes a method for handling service failures where the adaptation management mechanism dynamically uses contextual information such as execution state to decide on existing failure handling strategies such that it addresses the failure while minimizing the effect of the failure on the system quality of service criteria. In practice, restoring both of these types of properties may not be possible; therefore, the failure should be mitigated by partially providing the properties. This is often done through the full recovery of functional properties while degrading the non-functional properties of the service composition, considering it is hard to define degradation over functional properties. For example, Tan et al. [142] propose a method which uses Genetic Algorithms at runtime to find an alternative BPEL process which fully restores functional properties of the system while optimizing the quality of service criteria. Failure mitigation in terms of functional properties is difficult since it requires a mechanism which can guarantee that the set of functional properties provided in the degraded state is logically valid. In our proposed method, feature models are used to address this issue. A feature model introduces a set of hierarchical and cross-tree constraints between features of the system such that a feature model configuration respecting those constraints represents a valid system.

This allows the user to specify critical features in addition to critical non-functional requirements. Therefore, the adaptation mechanism can degrade a service composition in terms of its functional requirements while providing both critical functional and non-functional requirements, which was not possible before.

2.3.3 Realization of Adaptation

Strategies for realizing adaptation can be classified into three categories [71]: *retry*, *substitution*, and *reorganization*. In the retry strategy, the functional failure of a service composition is addressed by re-running the composition assuming that the failure is temporary, such as network failure, and retrying will address it [49, 45]. In substitution strategies, existing redundant services with the same functionality are used to replace the currently used services in the service composition such that the new composition can satisfy required functional and non-functional requirements [31, 71]. One example of such methods is the approach proposed by [36] in which a proxy is placed over all services with the same functionality which calls the redundant services and decides which result to be used in the execution process. This method is able to address failure without substantial change in the logic of the service composition. Reorganization strategies recover from a failure by finding alternate solutions which involve changing the services and how they interact with each other. Existing strategies often use adaptation rules [7, 90], AI

planning [71, 70], and other methods [28, 73] to recover from failure. For example, the failure recovery method proposed in [71] looks for a service with the same functionality when a failure occurs; in cases that such a service does not exist, the method uses planning as model checking to find a sub-process which can replace the failed service. Our work is similar to these works in the sense that it tries to address failure by re-planning to find an alternate composition. However, our method is different as it works on features rather than at the implementation level.

2.4 A Taxonomy for DSPLs

Considering that the focus in DSPL engineering approaches is the ability of runtime adaptation, this work describes what adaptation is through defining dimensions which characterize it and compares the existing approaches according to the type of adaptation they offer. Subsequently, this work describes how adaptation realization can be conceptualized using the MAPE-K loop [77], over which the dimensions of our taxonomy will be defined. The dimensions of the taxonomy are defined such that they can be generally used to characterize a DSPL engineering approach. Possible values for each dimension are discussed, and examples from existing DSPL engineering approaches are provided.

2.4.1 Properties of an Adaptation

Self-adaptation covers a wide range of runtime changes. In order to arrive at a better understanding of the kind of adaptations which are addressed by an approach, it is necessary to characterize the system adaptations developed by each approach. The characteristics of alternative adaptations are determined by a number of points of difference (also called dimensions). These points refer to the properties of an adaptation itself regardless of techniques and tools which are used to realize it. In [9], 20 dimensions have been proposed for characterizing an adaptation in a specific system. However, these dimensions are defined for characterizing adaptation in a single system and are too specific for comparing approaches for developing self-adaptive systems. This is because different approaches often have some degree of generalization to support versatility. Therefore, a set of dimensions for comparing different approaches, dimension set is defined. Figure 2.2 shows the dimensions classified in three categories. The dimensions in each of the goal, change, and mechanism categories represent characteristics of the goal of an adaptation, change that caused an adaptation, and mechanism used for enabling adaptation, respectively.

i **Goal type.** This dimension is concerned with the intent of adding runtime adaptation in a DSPL. In the definition of autonomic computing [77], Kephart and Chess cite four aspects for self-managing systems: self-configuring, self-optimizing, self-healing and self-protecting. A fully auto-

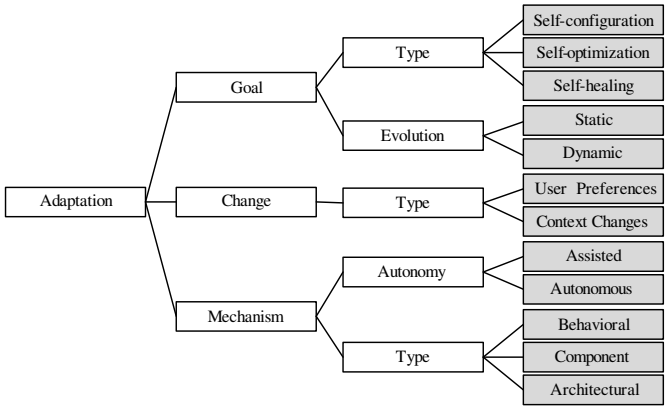


Figure 2.2: Dimensions characterizing an adaptation.

nomic system should to some extent address these four aspects. However, these aspects may be considered as references for classifying the goal of adaptation in different systems. One or more of these goals may be the motivation for enabling runtime adaptation. In the following, some of the possible goals of adaptation have been covered, except that of the self-protection aspect due to the unavailability of a DSPL engineering approach with this goal.

- *Self-configuring.* The goal here is to enable the system to automatically reconfigure itself automatically as the context or requirements change such that the system satisfies the high level policies. This is similar to product configuration in software product line engineering where a product is configured for different situations such that it satisfies the needs. The goals can be defined as a condition that should be always satisfied for the corresponding system; for example, the response time of a specific service of the system should be

always lower than a specific duration. In [21], the Genie approach was used to develop an application that runs on sensor nodes in a sensor network for flood monitoring where the application needs to be self-configuring. In this specific problem domain, the goal of adaptation was to enable the application to switch from the battery preserving configuration to the high performance configuration so as to be able to communicate faster when flood was likely to occur.

- *Self-optimizing.* A system supports self-optimizing adaptation when the goal of enabling adaptation is to make the system perform optimally under different environments and contexts. Usually the optimality of the system is represented with a single utility value which is typically the weighted sum of different quantitative attributes of the system (such as non-functional attributes). For example, In [64], MADAM was used to develop a supporting application for service technicians working in geographically dispersed installations. The application adapts itself to changes in the context (such as connectivity, noise, etc.) to optimize user experience represented with a utility value.

- *Self-healing.* In self-healing adaptation, the goal of adaptation is to have a system that adapts itself after a failure in order to reduce the impact of failure on the system. A self-healing system enables search for alternatives when a component of the system fails. For example, the MoRE approach is used for developing self-healing smart

homes where failure of devices is likely to happen [34]. In this example, failure of devices is handled by exploiting alternative ways; for example, alarm failure results in the blinking lights of the house. As an another example, Abotsi et al. [1] developed a DSPL engineering approach for developing a web-based application where the system is monitored for critical conditions. In case of a critical condition, a suitable variant of the system is identified and selected to recover from the critical condition. For instance, variants that use less resource-demanding contents are selected when there is a lack or dearth of resources.

ii **Goal evolution.** This dimension denotes whether the goal of adaptation can change at runtime or not. The change in the goal is often required when the system faces an unforeseen situation not supported by the adaptive system. The ability to evolve adaptation goals at runtime depends on how self-adaptation has been defined:

- *Static.* In this type of adaptive systems the goal of adaptation stays the same during the lifetime of the adaptive systems. These systems often have a fixed adaptation policy and system variants. In the case that new goals are needed, the system should be stopped and modified to support new goals. However, the verification of these systems are often easier considering their limited flexibility.

- *Dynamic.* Here, the system adaptation manager provides a method to introduce new adaptation goals to the system. In an approach proposed by Helleboogh et al. [67], a *meta variability model* is defined over the variability space model of the system, which provides the information required for anticipating goal updates and explicitly expressing how these goal updates are supported. The Models@Run.time framework also enables evolution of system goals by allowing modification to planning rules at runtime.

iii **Change type.** The trigger of an adaptation influences how monitoring and planning activities operate in a self-adaptive system. The types of changes which can trigger adaptation may be generally classified as context changes and changes in operational preference [103].

- *Context.* Context changes are those changes that take place in the internal and/or external environment of a system. When the system needs to responds to context changes, it requires the right sensors to measure the properties of interest in real-time. This is basically done in order to have a consistent view of the context. The gathered information is then used in planning for adaptation. Therefore a mechanism needs to be provided by the engineering approach for capturing the context information. For example, CAPucine [115] uses the COSMOS framework for monitoring the context. In this framework, a component is developed for each sensor. The context

information gathered by these components is then used to trigger Event-Condition-Action rules for adaptation.

- *User preferences.* These changes are alterations in user requirements or priorities at runtime, and therefore the trigger for adaptation is an external entity such as a user. The operational preferences are often captured using an interface. The change in requirements is often expressed in high level terms such as features, and is realized by an adaptation manager. In the Helleboogh et al. approach [67], the user of the system can change the operation mode of an automotive transportation system in a warehouse in response to changes in the operating condition. For example, when a load of goods arrives at the warehouse, the operation mode of the system is changed by the user to efficiently unload the packages. Another example for operational preferences is that proposed by Wolfinger et al. [153] where the users' preferences are captured by wizard-like dialogs following a decision model. In each dialog, the user selects from available alternatives in a decision point while the system offers the consequence of selecting each alternative. The final configuration of the system is built based on user decisions captured by the wizard.

iv **Mechanism autonomy.** This dimension defines if an outside entity (mostly humans) is involved in deciding about the adaptation. In autonomic computing, the degree of a system's autonomy is demonstrated

in five levels from adaptations which are completely manual (basic level) to adaptations which are completely automatic and where the human agent only specifies high level business rules and policies. In this work, a simpler classification presented by [9] is used.

- *Assisted.* In the assisted adaptation mechanism, the system cannot always complete adaptation without the help of an external entity. The degree of involvement of the external entity could range from deciding adaptation actions to a mere confirmation. An example for assisted adaptation in DSPL engineering is the approach of Baresi et al. [14], where a variability designer, in the form of an Eclipse plug-in is used for managing the variability of the system represented by a feature model.
- *Autonomous.* In an autonomous mechanism, the system decides and performs the needed adaptations automatically. The required behavior of the system is often provided using high-level goals or rules. It is according to these goals or rules that the system makes its adaptation decisions. Therefore, the system should be context-aware, which implies it needs to monitor its interior state as well as the surrounding environment. It also needs to have access to some form of reasoning mechanism.

v **Mechanism type.** This dimension defines the type of changes which occur in the structure of a system as a result of adaptation. The way the

adaptation is performed in the system limits its applicability. Gomaa et al. [60] classify adaptations into three classes according to the extent by which they affect the architecture of the system:

- *Behavioral adaptation.* In this type of adaptation, system behavior changes although its architecture stays the same. This type of adaptation can be implemented by changing runtime entities using parametrization, inheritance, and preprocessor directives in the code [59, 138]. This type of adaptation is simpler to implement; however, it limits system adaptability since system adaptation is limited to the code of the constituents within the system. Additionally, modification and extension of the system is harder because, in most cases, adaptation logic is interwoven with application code.

- *Component adaptation.* Here, adaptation is achieved by substituting a component with another similar interface though with a different behavior. This substitution provides a plug-in mechanism, where plug-ins are replaced with other plug-ins of the same type as the system adapts. However, some components and also the way these components are connected to each other stay the same. An example for this is MADAM [53], where a number of ports are defined in the architecture which can be filled with components of the same type. In order for the system to adapt, the system middleware allows components connected to the ports to be replaced at runtime.

- *Architectural adaptation.* In this type of adaptation, the structure or the architecture of the system changes to modify the collaboration between components or the incorporation of new components in to the system. For example, in an approach by Lee et al. [84], the workflow services, which show the way services work together, change at runtime, changing the behavior of the system. In another approach by Cetina et al. [34], the channels which connect services together activate and deactivate at runtime changing the architecture of the system.

These dimensions characterize the type of adaptation for a self-adaptive system. The required value for each of these dimensions should be specified when an approach is selected for developing a self-adaptive system.

2.4.2 DSPL Engineering Approaches

The focus of DSPL engineering approaches is mostly on building adaptation managers for self-adaptive systems. In order to break down the concerns for building an adaptation manager, the adaptation manager is conceptually represented as a MAPE-K loop model [77], similar to what Bencomo et al. [20] have proposed. Several dimensions have been extracted that can be used to categorize and distinguish between existing DSPL engineering approaches using the MAPE-K loop. In each category, the dimensions of the mechanism used for implementing each step as well as the knowledge related to that step

are listed. For each dimension, the aspect of the DSPL engineering approach that is characterized by that dimension is described and a number of possible values for that dimension is listed. For each of these values, its description and supporting examples from the literature are provided.

It should be noted that our goal is not to exhaustively cover all the possible dimensions of dynamic software product line engineering approaches but to provide an initial taxonomy for comparing them. Figure 2.3 shows the taxonomy categories, dimensions and some of their possible values. White boxes represent dimensions, and grey boxes display features.

2.4.2.1 Monitoring & Analysis

The DSPL engineering and SPLE paradigms have different motives for creating new variants of the system. On the one hand, in the SPLE paradigm, different variants are built according to users' requirements. Therefore, requirement elicitation approaches are used for extracting information needed for selecting system variants. On the other hand, variants are built according to the context in a DSPL by monitoring and analysis the context. Thus, SPLE ideas are not directly applicable for the monitoring and analysis of DSPL. DSPL engineering approaches do not often focus on the monitoring and analysis steps. Additionally, a wide range of monitoring mechanisms as well as a comparison framework for them already exist in the literature (such as [42]). Therefore, instead of addressing the dimensions for these mecha-

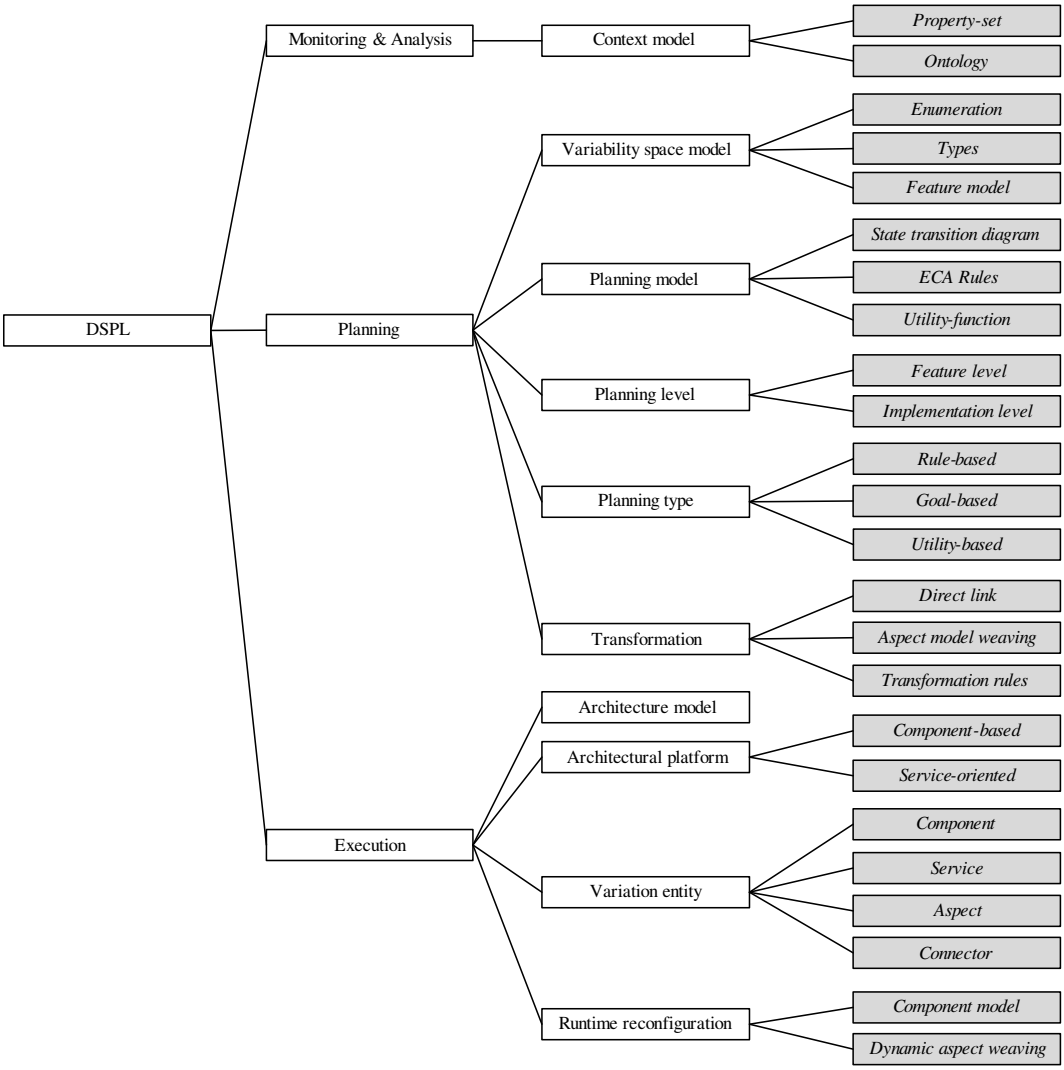


Figure 2.3: The taxonomy of the proposed framework.

nisms, this only covers the context model, which is also used in the planning step.

i. **Context model.** A context model is used to represent the knowledge acquired by the monitoring and analysis steps. This is a minimal representation of the system context that is important for planning. The information which the analysis step produces is more abstract than the information provided by monitoring, which allows a more efficient processing by the planner. There are generally two types of context models as presented in follows:

   - *Property-set.* In this type of context model, the context of the system is represented by a set of properties crucial for planning. The values of these properties are updated at runtime through monitoring and analysis which may lead to the adaptation of the system. For example in the MUSIC framework [126, 54], the information about the environment (e.g., ambient noise) as well as information about internal state of the system (e.g., device memory) and non-functional properties (e.g., network quality) are kept and updated as values to a number of context variables. The planner accesses these context variables to decide about the system adaptation.

   - *Ontology.* In this type of context model, ontological representation languages such as OWL [96] are adopted to relate (together) the important properties of the system using semantic relationships in

the ontology. The use of an ontology allows a more structured representation of the context. It also enables reasoning over the context model using existing semantics reasoners and sharing of the context model between different systems. For example in [6], the OWL language is used for expressing the context information in a way that allows for reasoning. In this framework, the properties are organized using classes and their relations are expressed using semantic relations. In another case study [32], the OWL language is used to represent the context model of a smart hotel.

2.4.2.2 Planning

In a typical software product line process, the application engineers develop a suitable variant of the product for a specific user by selecting and instantiating the assets developed during domain engineering with the help of variability models. In the context of dynamic software product line engineering, variant selection should be performed at runtime, often without any external help. Therefore, the system should take the role of the application engineer to select and instantiate new assets to generate a new variant of the system. This process is called planning. Most efforts in the dynamic software product line community have been in the area of creating planning approaches [20]. A planner is the component in planning which uses context information provided in the context model to decide if an adaptation is necessary, and if

so, finds the architecture model for the most suitable variant of the system. The planner uses a *planning model* to select the next variants of the system. The planning model represents how the system selects the desired variant of the system according to the current context and the possible variability of the system represented in the *variability space model*. The variability space model represents possible variants of the system at runtime. In order to separate higher abstract concerns from implementation requirements in planning, in practice, the variability space model is represented at a higher abstraction level than the architectural fragments of the system. The adaptation manager uses the context model, variability space model, and the planning model to select the target system variant from the variability space model. If the variability space model is more abstract than the implementation model, the high-level model (e.g., feature model) is mapped to the implementation level model (e.g., architecture model) using a *transformation* approach. The implementation level model is then used by the execution step to reconfigure the running system to the selected target system.

i. **Variability space model.** The variability space model represents the difference points in alternative variants of the system and the possible choices in each of the points. The model can represent variability at different abstraction levels. For example, the variability of the system can be represented by what features different variants of the system have or what components are different between the variants of the system.

- *Enumeration.* In this type of variability modeling, the possible variants of the system and their properties are enumerated as a set of possible variants [154]. The planner selects the most suitable variant of the system for the current context from the enumerated set and adapts to it. Those systems that model their variability using this type of modeling are generally simpler to design and validate. However, it is likely that the size of the possible variants of the system increases rapidly and becomes unmanageable as the system grows [103]. This type of variability modeling is usable when the possible variants of the system are possible to define. For example in a case study of using DSPL [22], the enumeration model is used to model three possible variants of a network discovery agent. These variants showed what kinds of roles the agent can have in the service discovery process at runtime.

- *Types.* In this type of variability space modeling, difference points in a base system are specified and called variation points. A type is defined for each variation point within the system where different alternatives of the same type can be selected for each variation point. The adaptation in the system takes place by the replacement of alternatives with the same type in the variation points of the system. The current variant of the system can be specified by knowing the alternatives that have been selected for the variation points. An example for this type of variation management is the

MADAM approach [53], where ports are defined in the architecture of the system. Ports are the locations where different components with the same interface can be substituted with one another. Types provide a simple and manageable solution for managing variability in a system [64]. However, one problem facing this type of variability modeling is that it does not support cross-cutting variations, where the selection of alternatives in one variation point puts constraints such as exclusion or inclusion on other variation points [64].

- *Feature modeling.* Feature models [75, 41] are widely used in SPL engineering for representing the variability of the product family [103, 33, 116, 74, 5]. A feature model provides a hierarchical structure that represents the organization of the features which are 'increments in product functionality' [19]. When decomposing a feature into sub-features, the sub-features may be optional or mandatory, or may form Xor-, Or-, or And-groups. These relations express the possible variability of the system. Feature models also represent cross-cutting variations using integrity constraints. Feature models are commonly used as a variability space model in DSPL engineering approaches because of their power to represent complex variability of a system.

ii. **Planning model.** The adaptation planner of a system is often implemented using a reasoner which considers the variability model, the

context, and the current variant of the system as inputs. It then develops the most suitable variant for the current context according to its planning knowledge [114]. The planning knowledge could be hard-coded into the reasoner or expressed explicitly by a model which is available at runtime and used by the reasoner. Using models for representing the planning knowledge eliminates the need for the development of a new reasoner from scratch for every system developed with a DSPL engineering approach. A planning model (decision model) specifies the policy of the system for selecting the suitable variant of the system by having access to the current variant of the system and the current context at runtime [8]. The type of planning model is determined by the planning method that is used in the system. Here, the planning models used by the reviewed systems in this study are covered.

- *State transition diagrams.* In these models, which can only be adopted when enumeration is used for representing the variation space model, the adaptation policy of the system is represented using a state transition diagram. The states are variants of the system and the transitions are the possible adaptations. The guard condition of a transition defines the situation in which that specific adaptation could take place. One problem associated with the state transition diagram is that it is not easy to maintain after they are initially designed because of their relatively complex structure [103]. An example of using this approach to define the adaptation policy

is a case study of the Genie framework [21] where a state transition diagram is used to specify when a flood monitoring agent should change its behavior according to its context and what target variant it should adapt to.

- *ECA rules.* Event-Condition-Action rules represent what actions the system should perform in response to an event when certain conditions hold [103]. Events could be changes in the environment (e.g., network traffic) or the system state (e.g., processor usage); Conditions could be conditions of the current environment or the current variant of the system; and finally Actions can be a set of activations and deactivations over the system components or similar changes in system features. Expressed using different languages and models, the ECA rules are often elegant and readable [52]. The planning model of the system can also be modified easily by adding and removing ECA rules. However, a problem with ECA rules is that when the number of rules is large, conflicts are likely to arise between the rules which are difficult to detect [52]. In such a case, it is ambiguous how the system should act, and a conflict resolution mechanism is essential, for example, giving priorities to some actions. An example for using ECA rules to define adaptation policy is a case study where the ECA rules are used in a dynamic access control product line to specify the changes in the available functionality of the system as users with different roles use the system [43].

In this system, the ECA rules specify access to what functionality should be granted to a user according to its role.

- *Utility functions.* In this type of planning modeling, the planning model consists of a utility function which is often a heuristic representation of the desirability of adapting to a variant of the system or performing an action in a specific context [52]. This utility function is used to select the most desirable variant of the system to adapt or determine the action that should be performed in each context of the system. An example for this is the MADAM approach [53], where a utility function represents the desirability of each system variant and is the weighted sum of the important properties of the system. One problem associated with the utility function is the difficulty often encountered in the design of such a function that precisely models the desirability of the system [64].

iii. **Planning level.** The planning level represents the level of abstraction on which the planner is working. In other words, this dimension specifies the planner's abstraction level. For example, the planner may see the system as a number of components collaborating with each other and therefore decides which components should be active, their configuration and how they should collaborate(e.g., [64, 21]). Furthermore, a system can be represented in a planner by its features. Therefore, the planner decides on what features should be active in the system (e.g., [103, 34]).

The selection of an appropriate abstraction level removes unnecessary details while preserving the required information for decision making.

- *Feature level.* Similar to feature model configuration in software product line engineering, the planner could decide at the feature level which would abstract away implementation details. Deciding on the features promotes the separation of concerns by dissociating the context requirements from realization details [62]. Using high level abstraction for the planner necessitates the implementation of a transformation method which converts the feature level configuration to the implementation level configuration [103]. For example in [124] in a robotic case study, the adaptation planner for the robot software selects required features according to the context. The configured feature model is used by a weaver to compose related architectural models to create system configuration at the implementation level.
- *Implementation level.* Here, planning is performed on actual implementation-level variation entities (such as components). Planning on the implementation level is often harder than planning on features because reasoning becomes more complicated when context requirements and implementation constraints are taken into consideration at the same time. For example in MADAM [53], the requirements of the system are defined on the properties of the components and the

planner tries to find the components which satisfy these requirements best.

iv. **Planning type.** Planning type represents the kinds of planning methods which have been employed for selecting the most suitable variant of the system. Selecting the most suitable variant of the system is often expressed informally in the SPLE [8], but it should be specified explicitly and formally in DSPL in order to automate the selection process. Generally, the planners can be categorized into three groups [72]:

- *Rule-based.* In rule-based planning, adaptation takes place by following a set of rules which specify what actions should be performed in each particular context [20]. In this planning type, the planning strategy is often modeled using ECA rules or state-transition diagrams and the strategies are then enforced by a reasoner. In this type of planning, adaptation rules are often defined at design time which requires thorough knowledge of the operating environment at design time [20]. However, some rule-based planners allow the modification of the rules at runtime [103]. In a case study of the MoRE engine used in the context of smart homes [34], adaptation rules are organized as a set of condition/resolution rules. The conditions are specified using propositional logic over context variables. The planner is made of a reasoner which checks if a condition holds for

any of the rules and performs the resolution for those rules whose condition holds.

- *Goal-based.* In goal-based planning, the high level goal of the adaptation is defined, and finding how to achieve the goal is left to be figured out by the planner. In goal-based planning, unlike rule-based planning where the actions of the system in different contexts are specified at design time, the most suitable action to be performed is figured out at runtime by using a planning method. Usually in this planning type, the effects of different actions on the adaptation goals are formally defined. Subsequently, the problem of finding the most suitable action is reduced to a satisfiability problem (SAT) or a constraint satisfaction problem (CSP) which is then solved using a SAT-solver or a CSP-solver. For example in Parra et al. [117], a constraint-satisfaction problem solver is used at runtime to find the most suitable variant of the system which satisfies the goals of the adaptation. The goals of the adaptation are expressed through constraints over system quality of service properties.

- *Utility-based.* Goal-based planners categorize the current configuration of the system into two possible states: desirable, when the goals of the adaptation are satisfied, and undesirable, when the goals are not satisfied. However, it is not always possible to satisfy all goals of the adaptation at the same time or exactly specify the goals of adaptation. Therefore, the goal of planning changes to keeping the

system at the most desirable possible state. In such cases, a utility which is a quantitative value representing the desirability of the current system variant in a current context is defined for the system [56]. This utility is approximated by a utility function over the properties of the system and the system context. Similarly, the best action is often chosen using a heuristic which approximates the effect of each action on the utility of the system. For example in [126], the planner of an adaptive travel assistant mobile application uses a utility function to approximate the fulfilment of user preferences. This prediction function is used to find the configuration of the system which has the highest predicted utility.

v. **Transformation.** In order to promote the separation of concern between system requirements and the requirements of implementation, in many DSPL engineering approaches, planning is performed on the abstract model of the system (such as the feature model) rather than on the implementation model of the system (such as the architecture model). This separation necessitates a linking strategy between high level and implementation level models of the system. The linking is also required in many of the SPLE approaches and has been considered in several (automated) product derivation approaches in [155, 40, 66, 57] where the goal of linking is to develop a product using the shared product family artifacts based on the selected features of the system. These works focus on relating the variability space model of the system to system code

[150, 11] or its architectural model [118, 40, 155, 129], or its domain specific modeling language [152]. DSPL engineering approaches have been inspired by such SPLE approaches in their development of a customized link between the high level and the implementation models at runtime. The trend is simple and resource efficient for runtime usage. Here are the main three approaches used for this purpose:

- *Direct link.* One simple way of linking models at two levels of abstraction is to create a direct mapping between the features in the high level model to fragments of the architectural model in the system implementation (examples of these fragments are system components) [148]. Consequently, the architecture model of the system after adaptation can be built by selecting the fragments of the architecture model that are mapped to the selected features of the system. However, this type of linking is not practical in certain cases because direct mapping between features and the architectural model fragments would not exist [148]. In these cases, the linking between the feature model and the architecture model can only be enabled if the feature model is built considering the architecture. However building the feature model by considering the architecture does not provide proper separation of concerns. For example in the MoRE framework [34], the link between features and architecture model of the system is defined by a superimposition operator which

takes a feature as input and returns the components and connectors which should be active if that feature is selected in the system.

- *Aspect model weaving.* Some Model-Driven Development (MDD) techniques allow for the automated generation of detailed architecture models of the system from high level designs. Specially for creating a detailed architecture model from the configured feature model, aspect model weaving can be used as presented in SMARTADAPTERS [82] and Kompose [51]. In these approaches, application architectural commonalities are represented in a base model of the system. To create the final model of the system, each feature is mapped to a set of aspect models which are woven into the base model if that feature is selected in the feature model configuration of the system. For example in [104] this approach was used to build adaptive home automation system for dependent people. In their system, the authors defined a *pointcut model* representing where an aspect could be woven into the system and an *advice model* which represents what can be woven at joint points. When the application adapts, the advices are woven into the pointcut model according to *weaving directives*.

- *Transformation rules.* In this linking strategy, in addition to the feature model of the system, an alternative feature model is defined over the variability of the architecture model such that all possible variants of that model could be represented by that feature model

configuration. Subsequently this feature model is linked to the main feature model of the system using a set of transformation rules. The configuration for the feature model over the architectural model can be found using these transformation rules whenever a feature model configuration is available. For example in [3] this approach was used to link between features of the system and the architecture model of the system in an adaptive video surveillance system. The system was modeled using two feature models, one on the features of the system and another on the components of the system. These two models were linked using a set of rules represented using prepositional logic. When the system adapts, the planner finds a configuration for the feature model over components of the system satisfying the rules using a SAT solver.

2.4.2.3 Execution

The execution step is responsible for actual system modification such that the current variant of the system gives its place to the target variant of the system while ensuring system consistency. The following points address the dimensions related to the execution of the adaptation.

i. **Architecture model.** When an adaptation occurs, the planner should provide the architecture model of the target variant of the system to the reconfiguration middleware, where the actual system reconfiguration

takes place. This model is often represented with an Architecture Description Language (ADL) [37] such as a UML component model. This model is available at runtime and used for representing the current and target configuration of the system. Some of the DSPL engineering approaches define their own ADL in order to only include information crucial for the reconfiguration engine to perform the adaptation. This facilitates building an adaptation engine with a higher memory and processing speed efficiency. For example in MoRe [34], an architecture model named PervML [106] has been proposed and adopted where only the component and their connections are represented. Some other approaches use the architecture model which is used by the adopted reconfiguration middleware to define the system architecture. Fore example in [115], Fractal ADL [29] has been used, also used by the Fractal middleware, which performs the runtime reconfiguration.

ii. **Architectural platform.** Many of the DSPL engineering approaches and all of the approaches reviewed in this study use system architecture as the supporting element in the adaptation. Performing adaptation at the architectural level abstracts away the details of the implementation of the system constituents, allows managing the system in a higher level, and helps to reduce the complexity of adaptation [114, 143]. Component-based and SOA-based models are amongst the most prevalent:

- *Component-based.* In component-based architecture, components communicate through connectors to provide the required functionalities of the system. Although components are built to work independently from their context, they are not often designed to be reconfigurable at runtime. Thus, architectural styles have been proposed to facilitate runtime variability [114, 143]. Many DSPL engineering approaches use these architectural styles. Gomma et al. [61] propose a process for developing a component-based dynamic software product line where the architecture is built using reusable architectural patterns (such as master/slave, client/server), and where for each of these architectural patterns, a specific reconfiguration pattern is defined. Modeled with state and scenario-based models, these reconfiguration patterns provide a safe method for reconfiguration.

  Adaptation in a component-based architecture can be realized by substituting the components of the system with other components or by changing components' interconnectivity. In DSPLs, adaptation is often realized by replacing components or through component activation/deactivation. For example, in the MADAM approach, adaptation is performed by replacing a component with another component which has a similar interface.

- *Service-oriented.* Both service-oriented architectures and software product lines share the common goal of reusing existing assets for

developing new products. A number of approaches have already been proposed for developing service-oriented product lines [85, 97, 102, 2]. Service-oriented architecture has also been used for developing dynamic reconfigurable systems because it eases the dynamic composition of self-contained and loosely-coupled services [35, 128, 126]. Considering the motivations for using service-orientation both for software product lines and dynamic reconfigurable systems, there have been a number of works that developed service-oriented DSPLs [14, 84, 116, 60, 133, 34, 39]. Baresi et al. [14] propose a service-oriented DSPL where the system adapts dynamically by changing the process that defines the service interaction. The process is expressed using the Business Process Execution Language (BPEL). BPEL is a language for representing the interactions of services in a process and by consequence the behavior of the system. In their approach, the BPEL code of the system changes as the system adapts. Cetina et al. [34] propose a service-oriented DSPL engineering approach for pervasive systems (e.g., smart homes). Pervasive systems are highly dynamic in nature since new entities are frequently introduced to the system or removed. In their approach, Cetina et al. define two categories of components: devices and services which implement system functionality. These components are connected using communication channels. The system adapts by activating/deactivating both services and communication channels.

iii. **Variation entity.** Variation entities are parts of the system that change when the adaptation is carried out. The type and the granularity of the variation entities affect the type of adaptation which is supported by the system.

- *Components.* According to the definition of a component by Szyperski [141], components can be deployed independently, and their context dependencies are always specified explicitly. This disconnection of the component from its environment allows the component to be used as the variation entity in the adaptation. However, it has been argued that components do not have the same granularity as features most of the time [125] for they may be more coarse or fine grained. In order to support feature realization with a higher granularity than that seen in components, some approaches define reconfiguration over a component composition. For example in the Genie approach, a collection of components which addresses a specific area of concern is called a *component framework*. Component frameworks act as the variation entity in adaptations. One concern with using components as variation entities of the system is that certain features do not have corresponding components at the component level. For example, the implementation of an authorization feature may be distributed between multiple components. Another concern with using components as variation entities is in

maintaining states of the components as they are replaced with other components while the system adapts.

- *Aspects.* Aspect-oriented programming and dynamic aspect weaving can be used in order to enable runtime reconfiguration in a dynamic software product line adaptation platform. However, efforts for using aspect-oriented programming in both software product lines and DSPL have shown that this approach may lead to certain problems [76, 14]. Features are more abstract than aspects, and a feature may be represented by many aspects; therefore, handling the linking between the features of the system and the aspects may become unmanageable.

- *Services.* Services can be simply deployed into a system and easily composed with each other. They also have characteristics such as being loosely coupled, self-describing and having well-defined interfaces which make them viable candidates for runtime adaptive systems. For example, Rouvoy et al. [126] used services as the variation entities for ubiquitous environments, where new services constantly enter and leave the system. Additionally, alternative third-party services can be discovered and added to the system so they enable open adaptation where the system can evolve into new variants which have not been considered at design time. However, a problem of using services as reconfiguration entities appears when the links between services and system features are defined. Simi-

lar to components and aspects, services are not the most accurate representatives of the features in a system.

- *Connectors.* Connectors define how the constituent elements of a system are connected together. The connectors in a system could be glue codes, communication channels, or workflows. The connectors often define the flow of the data in the system. Therefore, they are selected as the variation entity when the flow of data defines the behavior of the system. Examples of this function for the connectors may be encountered in data processing systems (e.g., video filters) and business processes [143]. An example of using connectors at runtime adaptation is the MoRe framework [61] where connectors activate/deactivate in order to introduce new services and devices in a smart home application.

iv. **Runtime reconfiguration.** In order to execute the actual adaptation actions, the execution platform should provide the ability to modify the structure of the running system (such as the binding and unbinding of the components) and observe the running system's architectural configuration. These functionalities are often provided by an application-independent middleware which frees the applications from dealing with actual adaptation concerns by taking responsibility for performing the adaptation while providing interfaces for managing that. This middleware performs by implementing a type of reflection. Reflection in the

middleware [135] is the ability to provide introspection and intercession. A reflective middleware provides a model of the system which allows for any system change to be reflected on the model and likewise for any change on the model to be reflected on the system. Therefore, the planner can access the representation of the system behavior using this model (introspection). It can then reconfigure the system by modifying this model (intercession). In this dimension, the mechanism used by middleware that enables runtime reconfiguration in the system needs to be reviewed.

- *Component model.* A component model is a definition of the semantics and syntax of components which means what they do, how they are structured and how they are composed [83]. A system can reconfigure at runtime if the architecture and its component conform to a component model that allows runtime reconfiguration. For example, OpenCOM middleware [38] use a component model for enabling runtime reconfiguration in the system. This middleware is used in the Genie tool for developing DSPLs. OpenCOM is a reflective middleware which supports hierarchical implementation of the system using component frameworks. Component frameworks are sets of components addressing a specific feature of the system. The Fractal component model [29] is another reflective hierarchical component model that can be extended and customized. The OSGi framework [92] is yet another reflective component model

which works based on services and supports runtime installation, starting, restarting and uninstallation of services.

- *Dynamic aspect weaving.* Dynamic aspect weaving can be utilized to modify the aspects of the system at runtime. This ability can be used to adapt the system when the variation entity of the system is the specific aspect [120]. For example, in [44] features are implemented using dynamic aspects in AspectJ [78]. These dynamic aspects are added or removed at runtime using dynamic aspect weaving. In another work, Baresi et al. [14] used dynamic aspect weaving in a business process to add/remove features at runtime, which were represented with BPEL code fragments.

2.4.3 The Proposed DSPL

Having the taxonomy for characterizing adaptation and DSPL framework, it can be said the goal of adaptation in the proposed work is self-healing. Currently, the goal of adaptation is static and do not change at runtime. However, the proposed adaptation mechanism can be extended to allow goal evolution by modifying the feature model and service repository at runtime. The type of change that causes adaption in the proposed adaptation mechanism is context changes. Furthermore, the adaptation mechanism is autonomous and works by making architectural changes.

The proposed adaptation mechanism is realized as a DSPL with following dimensions: The context model used in DSPL is an ontology. For planning, it uses a feature model for modeling variability space and uses domain models that are used for service composition as the planning model. The planner decided on the feature level and it is an utility-based planner. The transformation between features and implementation is done through AI planning. In terms of execution of changes, a graph called workflow captures the architecture of the service composition at high-level which later is converted to BPEL code. The proposed adaptation mechanism is applicable to service-oriented architectures where the variation entity is a combination of services and how the services are connected to each other. For runtime reconfiguration, the proposed work simply deploys the replacement service composition and redirects the incoming requests to it.

2.5 Summary

This chapter has presented the preliminary concepts in the area of self-adaptation and dynamic software product lines. It also reviewed and compared existing work in the area of service composition and runtime adaption. Since this thesis focuses on proposing a new dynamic software product line engineering approach for service composition, it has developed a taxonomy characterizing a dynamic software product line engineering approach and used it for comparing dynamic software product line engineering approaches.

The proposed taxonomy is also used for designing the approach proposed in this thesis.

2.6 Related Publications

- Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Dynamic software product line engineering: A reference framework", *International Journal of Software Engineering and Knowledge Engineering* 27 (1) (2017) 144

Chapter 3

Automated Construction of Service Compositions

This chapter proposes an automated service composition and workflow optimization method which receives a set of functional requirements in the form of a feature model configuration and automatically builds fully executable WS-BPEL code. Specifically, the approach proposed in this chapter makes the following contributions:

- It proposes an AI planning based method for automated service composition for *variability-rich domains*, which operates based on software product line configurations as the main input model for specifying requirements and automatically generates a workflow that satisfies the functional requirements.

- It further proposes a method for optimizing the created workflow by considering the concepts of *safety* and *threat* from the AI planning domain in order to inject parallelism into the generated workflow and improve its execution efficiency (e.g., reduce execution time).

- It demonstrates how the generated optimized workflow can be directly translated into WS-BPEL code that can be executed without any input from the designer. This provides the added benefit for the designer in that this approach would only require her to select the desirable features from a software product line as a result of which an executable WS-BPEL code will be generated.

The following section provides the overview for the proposed approach and specifies how the rest of chapter is organized.

3.1 Approach Overview

The objective of our work is, given a set of requirements and constraints from the users, to automatically *optimally compose* services in order to satisfy the presented requirements. To achieve this objective, we rely on the integration of *software services* and software product line *features*. Researchers such as Lee et al. [85] have already explored and concretely investigated how services and features can be integrated. There is ample literature that builds on a two-phase lifecycle that integrates services and features in its first phase and

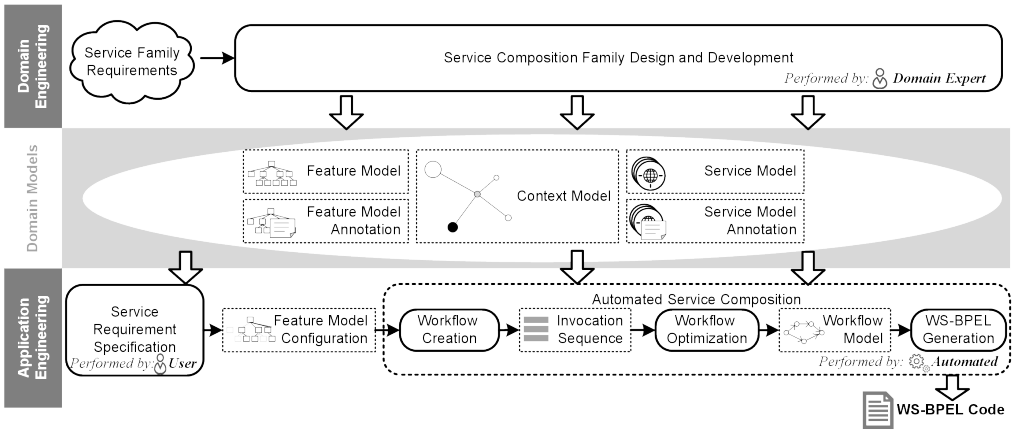


Figure 3.1: The proposed two phase process for composing service.

then, in the second phase, uses the integrated model to derive a product that satisfies the end-users' desired feature selections [97, 89]. The derived product will then be operationalized by the services that are connected to the selected features.

Current automated service composition methods work on inputs such as OWL-S service descriptions [70], temporal logic [25], or other formal languages, which are used to specify the characteristics of the desired service composition. However, we are interested in an input specification model *abstract* enough to be used by non-expert end-users to specify their requirements and an output that would be *concrete* enough to be directly executable. For this purpose we use feature models as the input specification model and generate the final outcome in WS-BPEL.

Figure 3.1 shows the overview of our proposed service composition and optimization process. Our process adopts SPLE's two-phase lifecycle [119].

In the first phase, a feature model and the other related artifacts for the variability intensive family of products are built based on the requirements of that family. Here, the requirements of the family are captured and modeled in the form of a set of models that are collectively called *domain models* [119]. In our approach, the domain model consists of five sub-models, namely the feature model, the feature model annotation, the context model, the service model, and the service model annotation. The feature model captures possible variability in the service composition in terms of features. The service model represents the services which are available for realizing different variants of the service composition family. The context model provides the means for representing the state of the service composition environment. The context model serves as a bridge, which links features to services. Feature model annotations use the context model to describe how each feature affects the preconditions of a service composition and how it impacts the operating context. Similarly, service model annotations describe the preconditions and effects of individual services. On the basis of these models, the features in the feature model are linked to services, realizing them using a context model, feature model annotations and service model annotations.

In the second phase, the feature model from within the domain model is used to select the desired features that need to be included in the final service composition through the *Requirement Specification* activity. Once the features to be included are selected and a feature model configuration is developed, the *Automated Composition Construction* activity works with

the feature model configuration as the input and automatically generates the appropriate BPEL code for the service composition. This work realistically assumes that domain engineering phase has already been performed and the features of the feature model configuration have been connected to relevant services based on existing techniques from the literature [85]; therefore, only the realization of the '*automated composition construction*' activity is the focus of this work.

The proposed automated service composition and optimization process consists of three steps. In the first step, the problem of finding the service composition which satisfies the selected features of the input feature model configuration is reduced to an AI planning problem. The result of this step is an intermediate model, which is referred to as the *workflow*, which abstractly represents the final goal service composition. The second step is optimizing the generated workflow in order to effectively use parallelism where possible. Although this step is optional, it significantly improves the efficiency of the workflow from different perspectives. In the third step, the optimized workflow is converted into a well-structured readable BPEL code which can be directly executed. This thesis first provides a case study which will be used as a running example throughout the chapter. Then, individual models in the *domain models* are first formally defined since the three mentioned steps work on domain models. Afterwards, how each of these three different steps are realized is elaborated.

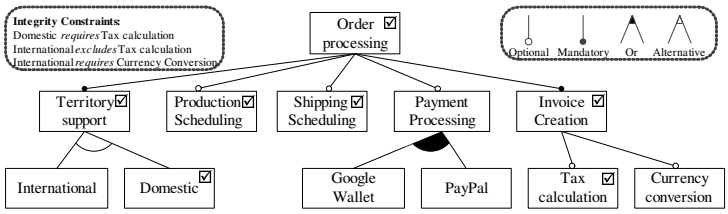


Figure 3.2: A sample feature model for an order processing family.

3.2 Case Study

Figure 3.2 depicts a sample feature model for a software family that processes a purchase request and creates an invoice. The family represented through the 'Order Processing' root feature has five sub-features, namely invoice creation, production scheduling, shipping scheduling, payment processing, and territory support, where shipping scheduling and payment processing features are optional. Territory support sub-features are mutually exclusive. Furthermore, the selection of the 'international' feature prevents the selection of the 'tax calculation' feature and requires the selection of the 'currency conversion' due to the integrity constraints. The selection of features marked with a checkbox in Figure 3.2 represents a valid feature model configuration that can also be considered to be the functional requirements expressed by an end-user.

Figure 3.3 represents a graphical representation for a WS-BPEL code for the possible realization of the feature model configuration in Figure 3.2 where features marked with checked boxes are selected. The variables which

this process works with and their types have been defined in the top left corner of the figure. This activity is a sequence which starts with receiving a request whose outputs are assigned to c and po variables which are variables containing information about the customer and purchase order. The next step in the sequence is invoking $requestInvoiceCreation$ that creates an invoice for a purchase order. This service makes a callback when it has all the required information for creating an invoice. The information is then sent in a flow activity. This flow activity is made of three sequences which can be run in any order and send tax information, production schedule and shipping schedule of a product to the scheduling service. The only observable dependency is that shipping schedule should be invoked after production schedule has been invoked. This dependency between the children of the flow is enforced using a link between these two activities. Afterwards, all the information is sent to the invoice creation service in the flow activity. The process waits for receiving an invoice from it and assigns it to variable i and then responds by replying with i.

3.3 Domain Models

As mentioned earlier, this work defines the domain model to consist of five sub-models, namely the feature model, the service model, the context model, the service model annotation, and the feature model annotation. In addition to the domain models, our method works with two other models, namely the

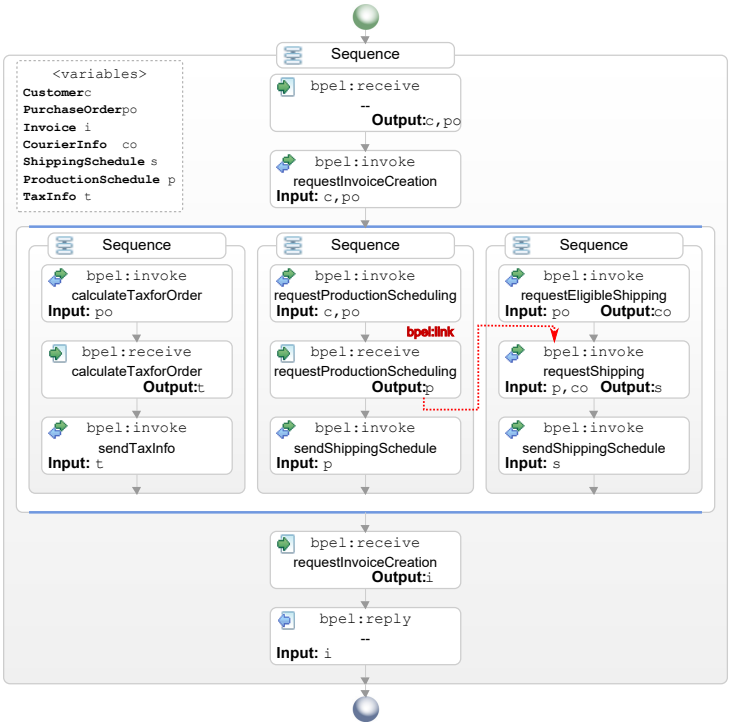


Figure 3.3: Graphical representation of a possible WS-BPEL process for order processing.

feature model configuration and the workflow model. A feature model configuration is a subset of features in the feature model, which respects constraints enforced by the feature model and is created by the end-user to specify a desired functionality for the system. The workflow model is an internal model, which is generated through an optimization process and is used to generate a BPEL code. In essence, the workflow model is a graph representing the precedence relations between service executions. This method enables us to simplify the solution by separating the activities that are required for the creation of a service composition from those activities that are required for

generating BPEL code from that service composition. A feature model and feature model configuration are formally defined as follows:

Definition 3.3.1. ***(Feature Model)*** *A feature model is a tuple* $fm = (F, \mathcal{P}, \mathcal{F}_O, \mathcal{F}_M, \mathcal{F}_{IOR}, \mathcal{F}_{XOR}, \mathcal{F}_{req}, \mathcal{F}_{exc})$ *where*

- F *is a set of features;*
- $\mathcal{F}_O : F \mapsto F$ *is a function which maps an optional child feature to its parent;*
- $\mathcal{F}_M : F \mapsto F$ *is a function which maps a mandatory child feature to its parent;*
- $\mathcal{F}_{IOR} : F \mapsto F$ *and* $\mathcal{F}_{XOR} : F \mapsto F$ *are functions which map child features to their common parent feature, grouping the child features into optional and alternative groups, respectively;*
- $\mathcal{P} : F \mapsto F$ *is a function which maps each feature to its parent and hence it can be said* $\mathcal{P} = \mathcal{F}_O \cup \mathcal{F}_M \cup \mathcal{F}_{IOR} \cup \mathcal{F}_{XOR}$*;*
- $\mathcal{F}_{req} \subset F \times F$ *is a set of requirement relations which represents the dependencies between features.*
- $\mathcal{F}_{exc} \subset F \times F$ *is a set of exclusion relations between features which represents pairs of features which cannot be both simultaneously selected in a valid feature model configuration.*

For the feature model shown in Figure 3.2, the set F would include all the features in the model and the function $\mathcal{F}_M$ will include mappings such as $(ShippingScheduling, InvoiceCreation)$ which represents the constraint that the $ShippingScheduling$ feature should be in the configuration when the $InvoiceCreation$ feature is in that configuration. Other structural and integrity constraints in the feature model can be defined similarly. Using the above definition, a feature model configuration can defined as follows:

Definition 3.3.2. ***(Feature Model Configuration)*** *A feature model configuration is a set $C \subseteq F$ where*

- *if $f \in C$ then $\mathcal{P}(f) \in C$*
- *if $f' \in C$ and $(f, f') \in \mathcal{F}_M$ then $f \in C$;*
- *if $f, f' \in F$ and $f'' = \mathcal{P}(f) = \mathcal{P}(f')$ and $(f, f''), (f', f'') \in \mathcal{F}_{\mathcal{XOR}}$ then $f \in C \Rightarrow f' \notin C$*
- *if $f, f' \in F$ and $(f, f') \in \mathcal{F}_{req}$ then $f \in C \Rightarrow f' \in C$*
- *if $f, f' \in F$ and $(f, f') \in \mathcal{F}_{exc}$ then $f \in C \Rightarrow f' \notin C$*

Based on this definition, a valid feature model configuration is a subset of features that satisfy the structural and integrity constraints expressed in the feature model. Now, given such a feature model configuration, our objective is to develop a workflow that would realize the feature model configuration

using services. In order to represent how requirements represented by a feature model configuration are fulfilled using services, the specific orchestration of services that satisfy those requirements is captured in a workflow model. A workflow specifies the sequence of interactions between the services in a service composition. The service model, which is the basis for the workflow model, is formally defined as follows:

Definition 3.3.3. ***(Service Model)*** *A service model $s = (I, O, O_c)$ is a triple where*

- *I is a set of entities that the service accepts as input when invoked.*
- *O is the set of entities that the service returns as output after being invoked.*
- *O_c is the set of entities that is received in service callback.*

Assuming $\mathcal{IO} = I \cup O$ for each entity $i \in \mathcal{IO}$, $\mathcal{T}(i)$ shows the type of the entity and is defined in Definition 3.3.5.

In this definition, a service is defined by the entities that it takes as input, the entities that it returns after invocation, and the entities that it returns in its callback if it results in a callback. Each of these entities is strictly typed using the function $\mathcal{T}$. For example in Figure 3.3, the service $requestProductionScheduling$ can be defined with $I = \{inputPurchaseOrder\}$, $O = \{\}$, and

$O_c = \{outputProductionSchedule\}$ where the type of entities $inputPurchaseOrder$ and $outputProductionSchedule$ is $PurchaseOrder$ and $ProductionSchedule$, respectively. Using this definition, the workflow model is defined as:

Definition 3.3.4. ***(Workflow Model)*** *A workflow model is a triple* $w = (E, N, \mathcal{E})$ *where*

- E *is a set of entities which can be used as input or output in the operations of the workflow. Each entity* $e \in E$ *has a type.*
- N *is a set of operation nodes which can be:*
  - *An invocation node represented as a triple* $(s, \mathcal{I}, \mathcal{O})$ *where* s *represents the invoked service and* $\mathcal{I}$ *and* $\mathcal{O}$ *specify the mapping relation between the workflow entities, and the inputs and outputs of the services.*
  - *A receive node is a pair* $(s, \mathcal{O}_c)$ *where* s *represents the invoked service which has resulted in callback and* $\mathcal{O}_c$ *specifies the mapping relation between workflow entities and the outputs of service callback.*
- $\mathcal{E} \subset N \times N$ *shows the dependencies between operation nodes such that for each* $n, n' \in N$, $(n, n') \in \mathcal{E}$, *the operation of node* n *should be performed before* n' *in the execution process.*

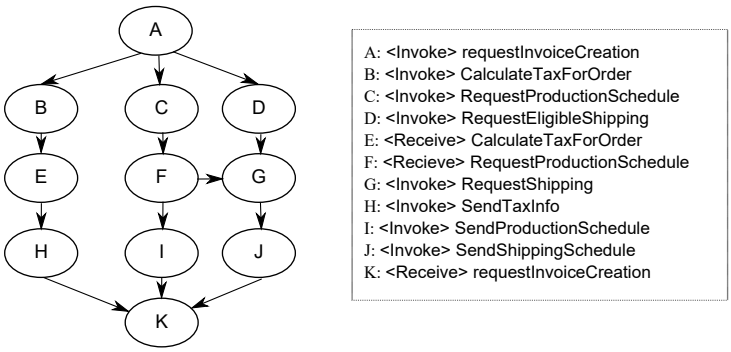


Figure 3.4: A workflow solution satisfying requirements of feature model configuration in Figure 3.2.

Figure 3.4 shows a graphical representation of a workflow model represented as a directed graph in which nodes are operation nodes and edges represent the dependencies between operation nodes. For example, node A shows an operation node which involves the invocation of the *requestInvoiceCreation* service and the edge from node A to C shows that the invocation of *requestProductSchedule* should be performed after the invocation of the *requestInvoiceCreation* service.

In order to be able to automatically make a transition from a feature model configuration to a workflow model, the *context model* which represents the environment in which the service composition will operate is defined. Relations between the feature model, the service model and the context model are represented with annotations on these models. These annotations are used for creating a workflow from the feature model configuration. The context model is formally defined as follows:

Definition 3.3.5. ***(Context Model)*** *A context model is a triple* $c = (c_T, c_E, S)$ *where*

- c_T *denotes context types, which is a tuple* $(\Theta, \Phi, \mathcal{F})$ *where*
  - Θ *is a set of data types*
  - Φ *is a set of fact types*
  - $\mathcal{F} : \Phi \mapsto \Theta \times .... \times \Theta$ *is a function which specifies the data type of entities that each fact type is defined on. The number of types which fact type is defined is arbitrary based on the type of the fact.*
- c_E *is context entities which is a pair* $(E, \mathcal{T})$ *where*
  - E *is a set of entities that exist in the context*
  - $\mathcal{T} : E \mapsto \Theta$ *is a function which defines the type of each entity*
- S *is a context state which is a set* $S \subset \Phi \times E \times .... \times E$ *such that for each fact* $f = (\phi, e_1, ..., e_k) \in S \Rightarrow (\phi, \mathcal{T}(e_1), ..., \mathcal{T}(e_k)) \in \mathcal{F}$ *and shows the facts which are true in that context. The number of entities in a fact is specified by the fact type.*

In our context model definition, *context entities* are similar to object instances passed between functions, and *context types* are used for strictly specifying entity types. Furthermore, the context model also consists of the *context state*, which is defined by *facts*. Facts can express the relationship between zero or more context entities. Let us elaborate on this using Figure

3.5. In this example, c and po are two context entities, which are of customer and purchase order types, respectively. Furthermore, the fact $ordered(c, po)$ expresses that customer c has ordered the purchase order po. This fact is represented using the fact type *ordered* which relates an entity of type customer to an entity of type purchase order. In the following, how the context model information will be used to annotate features and services will be explained.

Based on the context model, each feature in the feature model needs to be annotated with three sets: *i)* the set of entities that are required for the execution of a service composition that includes this feature; *ii)* the set of facts that should be true in the current state of the context model in order for the service composition that includes this feature to safely execute, and *iii)* the set of facts that will become true in the context model once a service composition which includes this feature is executed. These annotations are formally defined in the feature model annotation:

Definition 3.3.6. ***(Feature Model Annotation)*** *The annotation for feature model fm is a function $\mathcal{A}_{FM}$ which maps each feature f in the feature model to a triple $(E_f, \mathcal{P}_f, \mathcal{E}_f)$ where*

- *$E_f \subset E$ is the set of entities that must exist in a context model in order to execute any service composition with feature f.*
- *$\mathcal{P}_f \subset \Phi \times E \times .... \times E$ is the set of facts which should be true in the context model in order to execute a service composition with feature f.*

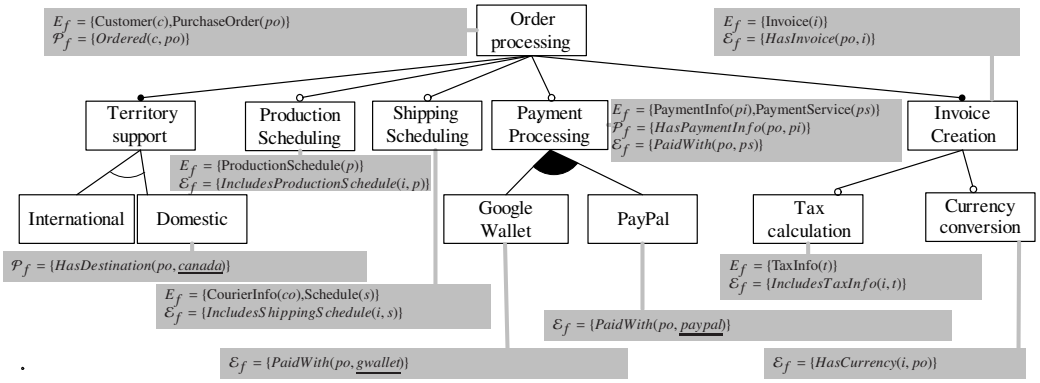


Figure 3.5: An annotated feature model for the order processing family.

- $\mathcal{E}_f \subset \Phi \times E \times .... \times E$ *is the set of facts that will be true in the context model after executing a workflow with feature* f.

Figure 3.5 shows the annotations for our order processing feature model. As seen in the figure, for each feature, $E_f, \mathcal{P}_f, \mathcal{E}_f$ are defined as needed. For instance, the figure shows that for the 'Invoice Creation' feature to be included in the goal service composition, a context entity i of type *Invoice* needs to be present in the context model. Furthermore, when the service composition consisting of the 'Invoice Creation' feature is executed, the fact $hasInvoice(po, i)$ will become true as an effect, which means purchase order entity *po* will have an invoice entity i.

In addition to feature model annotations, services are also annotated in a similar vein. The annotation of services with pre-conditions and post-

conditions (effects) has been already widely used in the literature [70] and a similar strategy is adopted.

Definition 3.3.7. ***(Service Model Annotation)*** *A service annotation for service s is a tuple $\mathcal{A}_s = (\mathcal{P}_I, \mathcal{Q}_I, \mathcal{R}_I, \mathcal{P}_C, \mathcal{Q}_C, \mathcal{R}_C)$ where assuming $\mathcal{IO} = I \cup O \cup O_c$ where*

- *$\mathcal{P}_I, \mathcal{P}_C \subset \Phi \times \mathcal{IO} \times .... \times \mathcal{IO}$ are the facts that should be true over the entities interacting with the service (including inputs, output, callback output) in order to invoke the service and receive any callback.*
- *$\mathcal{Q}_I, \mathcal{Q}_C \subset \Phi \times \mathcal{IO} \times .... \times \mathcal{IO}$ are the facts that become true over the entities interacting with the service after the service is invoked or the callback has been received.*
- *$\mathcal{R}_I, \mathcal{R}_C \subset \Phi \times \mathcal{IO} \times .... \times \mathcal{IO}$ are the facts that become false over the entities interacting with the service after the service is invoked or the callback has been received.*

For example in the service *requestProductionScheduling* in Figure 3.3, assuming the input $inputPurchaseOrder$ is of type $PurchaseOrder$ and the callback output $outputProductionSchedule$ is of type $ProductionSchedule$ in the context model, one could define the annotations for this service as $\mathcal{P}_I = \{ordered(?customerInfo, inputPurchaseOrder)\}$, $\mathcal{Q}_C = \{$ $hasProductionSchedule\ (inputPurchaseOrder, outputProductionSchedule)\}$, and the other annotation sets would be empty. This annotation means that

after the invocation of this service, the value of the output would be the shipping information for the input customer and after receiving the callback the value of the callback output would be the shipping schedule for the input customer.

In our model, the feature and service annotations serve as a bridge between the feature and service spaces, which allow us to automatically generate a service composition. This bridge will be used in order to find the workflow of a service composition, which satisfies the requirements specified by the end-users' feature selections.

3.4 Workflow Generation

Using the definition for the domain model, the workflow generation problem can be formally defined as: Given a context model type c_T, a feature model fm, a feature model configuration C, a feature model annotation $\mathcal{A}_{FM}$, a set of services S, and their corresponding annotations $\mathcal{A}_S$, the goal will be to generate a workflow w using services in S which satisfies the requirements of feature model configuration C.

This work proposes to represent the above problem as a standard planning problem and represent it using Planning Domain Definition Language (PDDL) [94] and then find a solution through AI planning. A planning problem is defined by the initial context state as the starting point of the planner

and the expected context state as the goal of the planner. A planning problem can be defined in different domains. A domain defines a set of possible actions that can be applied in order to change the state. A planner can find a sequence of actions, which move from the initial state to a goal state for a given domain if such a sequence exists. Representing the problem of generating a desired workflow as a planning problem allows us to take advantage of existing highly optimized planners for finding a possible workflow solution.

In order to represent the desired workflow generation problem as a planning problem, the planning problem and the planning domain are first formally defined and then it is discussed how generating a desired workflow can be accomplished as a result. This work adopts the widely used STRIPS planning specification model to provide our problem formalization, which can easily be converted to a Planning Domain Definition Language (PDDL) model [94]. A planning problem in STRIPS [50] can be defined as follows:

Definition 3.4.1. ***(Planning Problem and Domain)*** *A planning problem is* $p = (S_{initial}, S_{goal})$ *and problem domain is a set* A *where:*

- $S_{initial}, S_{goal}$ *are the initial and goal states. These states are represented by a set of atomic facts;*
- A *is the set of available actions. This set includes all the actions that can be done in order to change the state. Each action* $a \in A$ *is a tuple* $(I, F_{pre}, F_{add}, F_{del})$ *where*

- *I is the set of parameters that an action takes;*
- *F_{pre} is the set of atomic facts, which should be in a state in order for that action to be applicable (i.e., action a is applicable in state S where $F_{pre}(a) \subseteq S$);*
- *F_{add} is a set of facts that are added to a state after the action has been applied;*
- *F_{del} is a set of facts that are deleted from the state after the action has been applied. Therefore, if S_{succ} is the state after applying action a to state S then $S_{succ} = \Big(S - F_{del}(a)\Big) \cup F_{add}(a)$.*

Definition 3.4.2. ***(Planning Problem Solution)*** *Sequence $s =< a_1, ..., a_k >$ is a solution to the planning problem $p = (S_{initial}, S_{goal})$ in a domain A if*

- *a_1 is applicable on state $S_{initial}$;*
- *for each $1 < j \leq k$ action a_j is applicable in state S which has been reached by consecutive application of actions $a_1, ..., a_{j-1}$ on the initial state $S_{initial}$;*
- *consecutive application of actions $a_1, ..., a_k$ on initial state $S_{initial}$ will result in a state S such that $S_{goal} \subseteq S$ which means all the facts that are true in S_{goal} are also true in S.*

In our proposed method, the initial state of the planning problem is built using the sets of $\mathcal{P}_f$ in the annotations of the selected features in the

feature model configuration. These are the facts, which are assumed to be true before executing the workflow and therefore can be considered to be the initial state of the planning problem. Similarly, the goal state can be built using the sets of $\mathcal{E}_f$ in the annotations of the selected features in the feature model configuration given that these are the facts that are expected to be true after the execution of the workflow. This can be formally defined as:

Definition 3.4.3. ***(Planning Problem based on Feature Model Configuration)*** *For a feature model configuration C, the initial and goal states of a planning problem $p = (S_{initial}, S_{goal})$ are defined as follows:*

- $S_{initial} = \bigcup_{f \in C} \mathcal{P}_f$
- $S_{goal} = \bigcup_{f \in C} \mathcal{E}_f$

Initial and goal states of the planner are built by aggregating the annotations of the selected features from the feature model configuration. Figure 3.6 shows the PDDL representation of a planning problem of the feature model configuration shown in Figure 3.2. Lines 13-17 show the initial condition for the planning problem, created by unioning over all $\mathcal{P}_f$ sets of the selected features. For example, the fact on Line 15 has been created as a consequence of the annotation on the *Domestic* feature, which enforces that the destination of the purchase order is Canada. Lines 19-26 show the goal condition for the planning problem, which has been created in a similar vein using $\mathcal{E}_f$ sets of the selected features.

Considering the fact that the available services represent the possible actions in our problem definition, the problem domain is modelled based on the available services. Therefore, the possible actions would be the invocation of different services and receiving their callbacks. This can be formally defined as follows:

Definition 3.4.4. ***(Planning Domain based on Services)*** *In a domain where a set of services S is available, and for each $s \in S$ its corresponding annotation is $\mathcal{A}_S(s)$, the problem domain would be the set of actions A. For each service $s \in S$, A includes an action $a_{invoke}(s)$ and it includes action $a_{callback}(s)$ if the service call results in a callback. These actions can be defined as:*

- *Action $a_{invoke}(s)$ is defined as a quadruple $(I, F_{pre}, F_{add}, F_{del})$ based on the invocation of service $s = (I, O, O_c) \in S$ with annotation $\mathcal{A}_s = (\mathcal{P}_I, \mathcal{Q}_I, \mathcal{R}_I, \mathcal{P}_C, \mathcal{Q}_C, \mathcal{R}_C)$ where*
  - *input of the action is $I = I(s) \cup O(s) \cup O_c(s)$*
  - *$F_{pre} = \mathcal{P}_I(s)$*
  - *$F_{add} = \mathcal{Q}_I(s)$ and a predicate showing that service s has been invoked with parameters I and the callback is pending ($invoked(s, I)$) if it has a callback.*
  - *$F_{del} = \mathcal{R}_I(s)$*

```
(define (problem orderProcessing)
 (:domain orderProcessingDomain)
 (:requirements :strips :typing  :negative-preconditions )
 (:objects
  vShippingSchedule - ShippingSchedule
  vCustomer - Customer
  vProductionSchedule - ProductionSchedule
  vTaxInfo - TaxInfo
  vShippingCourier - ShippingCourier
  vInvoice - Invoice
  vPurchaseOrder - PurchaseOrder
  Canada - Location  )
 (:init
     (and
     (Destination vPurchaseOrder Canada)
     (Ordered vCustomer vPurchaseOrder)
     )
 )
 (:goal
     (and
     (IncludesTaxInfo vInvoice vTaxInfo)
     (IncludesShippingSchedule vInvoice vShippingSchedule)
     (HasInvoice vPurchaseOrder vInvoice)
     (IncludesProductionSchedule vInvoice vProductionSchedule)
     )
 )
)
```

Figure 3.6: Example of Planning Problem

- *Action* $a_{callback}(s)$ *is defined as a quadruple* $(I, F_{pre}, F_{add}, F_{del})$ *based on the callback for a service* $s = (I, O, O_c) \in S$ *with annotation* $\mathcal{A}_s = (\mathcal{P}_I, \mathcal{Q}_I, \mathcal{R}_I, \mathcal{P}_C, \mathcal{Q}_C, \mathcal{R}_C)$ *where*

  - *input of the action is* $I = I(s) \cup O(s) \cup O_c(s)$
  - $F_{pre} = \mathcal{P}_C(s) \cup \{invoked(s, I)\}$
  - $F_{add} = \mathcal{Q}_C(s)$
  - $F_{del} = \mathcal{R}_C(s) \cup \{invoked(s, I)\}$

The sets F_{pre}, F_{add}, and F_{del} for an action are built based on the set of $\mathcal{P}, \mathcal{Q}, \mathcal{R}$ of the corresponding operation in the related service. Figure 3.7 shows parts of the PDDL representation of the planning domain based on the services used for realizing the service composition satisfying the require-

```
(define (domain orderProcessingDomain)
(:requirements :strips :typing  :negative-preconditions )
(:types
          PurchaseOrder
          Customer
          ...
)
(:predicates
          (InvoiceCreationRequested ...)
          (HasProductionSchedule ....)
          ...
)
(:action IrequestProductionScheduleService

:parameters (
                  ?purchaseOrder - PurchaseOrder
                  ?customer - Customer
                  ?productionSchedule - ProductionSchedule
          )
:precondition
    (InvoiceCreationRequested ?customer ?purchaseOrder)

:effect
    (RequestProdScheduleCalled ?purchaseOrder ?customer ?prodSchedule)
)
...
)
```

Figure 3.7: Example of Planning Domain

ments of the feature model configuration shown in Figure 3.2. Lines 13-25 show the action representation of invoking the *requestProductionSchedule* service. Lines 15-19 shows the parameters for the action which is made of inputs and callback output since invocation of the service has no output. The preconditions for the action corresponding to the invocation of this service (i.e., the set F_{pre}) are represented in Lines 20-21. The effects for the action corresponding to the invocation of this service (i.e., the sets F_{add}, F_{del}) are represented in Lines 23-25.

Now that the planning goal and planning problem domain are concretely defined, a planner can be used in order to find a solution for the planning problem. The solution will be a sequence of actions with input variable assignments, which would take us from the initial context state to the goal

context state. Figure 3.8 shows the sequence of actions which has been found by the FF planner [69] as a solution for the example problem where each action and its input assignment is represented in a separate line. In the following, how the workflow can be built based on the planner solution is formally defined and it is proven that the generated workflow is a valid workflow and satisfies the requirements specified in the feature model configuration.

Definition 3.4.5. ***(Workflow based on Planning Problem Solution)*** *Based on a solution to a planning problem denoted as* $s =< a_1, ..., a_k >$ *for an input feature model configuration C, a workflow* $w = (E, N, \mathcal{E})$ *can be built where:*

- *The workflow entities set* $E = \bigcup_{f \in C} E_f$.
- *The operation node set* $N = \{n_1, ..., n_k\}$ *is made from the action sequence where* n_j *is built based on* a_j *where the service for the operation is the corresponding service for that action. Similarly, the assigned input and output for the operation nodes are corresponding entities assigned to the action parameters.*
- *The dependency set* $\mathcal{E}$ *is* $\{(n_{j-1}, n_j)$ *such that* $1 < j \leq k\}$ *which means the operation nodes should be executed in the order specified in the action execution.*

Lemma 1 *(Soundness of the Workflow)* Let C be a feature model configuration. A workflow w is generated based on Definition 3.4.5. w is a valid workflow in that the following conditions hold for it:

- **Cond 1.1** For each operation node n_j , all preconditions for it hold before its execution.
- **Cond 1.2** The receive operation node for each service is not executed before the execution of the invoke for that service.

Proof Cond 1.1 The proof goes by contradiction. Assume that there exists some nodes in w whose preconditions do not hold before their execution. Therefore there must be an i^{th} node in the workflow execution sequence which is the first node in the sequence whose precondition is not satisfied. It is shown that such a node cannot exist. If precondition for operation node n_i does not hold, it means there is at least one precondition p which does not hold. This operation node can be mapped to its corresponding planning action a_i. According to Definition 3.4.4, each precondition for n_i can be mapped to a precondition for a_i. Therefore, if p does not hold after executing operation nodes 1 to $i-1$ then there exists a p' which is a precondition for a_i that does not hold after executing actions 1 to $i-1$. The planner has chosen a_i for the next action, meaning that the planner has chosen an action whose preconditions are not satisfied. This is not possible.

```
1 (IREQUESTINVOICECREATIONSERVICE VCUSTOMER VPURCHASEORDER)
2 (ICALCULATETAXFORORDERSERVICE VPURCHASEORDER VCUSTOMER)
3 (IREQUESTPRODUCTIONSCHEDULESERVICE VPURCHASEORDER VCUSTOMER)
4 (IREQUESTELIGIBLESHIPPINGSERVICE VPURCHASEORDER VSHIPPINGCOURIER VCUSTOMER)
5 (RCALCULATETAXFORORDERSERVICE VPURCHASEORDER VTAXINFO)
6 (RREQUESTPRODUCTIONSCHEDULESERVICE VPURCHASEORDER VPRODUCTIONSCHEDULE)
7 (IREQUESTSHIPPINGSERVICE VPURCHASEORDER VPRODUCTIONSCHEDULE VSHIPPINGCOURIER
8       VSHIPPINGSCHEDULE VCUSTOMER)
9 (SENDSHIPPINGSCHEDULESERVICE VSHIPPINGSCHEDULE VPURCHASEORDER)
10 (SENDTAXINFOSERVICE VTAXINFO VPURCHASEORDER)
11 (SENDPRODUCTIONSCHEDULESERVICE VPRODUCTIONSCHEDULE VPURCHASEORDER)
12 (RREQUESTINVOICECREATIONSERVICE VCUSTOMER VPURCHASEORDER VINVOICE VTAXINFO
13       VSHIPPINGSCHEDULE VPRODUCTIONSCHEDULE)
```

Figure 3.8: Solution returned by a planner for our example problem

Proof Cond 1.2 The proof goes by contradiction. Assuming there exists a receive operation node n_i whose corresponding invoke has not been called. This operation can be mapped to its corresponding action a_i. According to Definition 3.4.4, one of the requirements of this action is a condition which only becomes true when the corresponding invoke action has been executed. Since the corresponding invoke action has not been executed, the planner has chosen an action whose preconditions are not satisfied. This is not possible. □

Lemma 2 *(Completeness of the Workflow)* Let C be a feature model configuration and w be a workflow generated based on Definition 3.4.5, the execution of w requires only the conditions specified by the feature model configuration and results in a state which satisfies all effects specified in the feature model configuration.

Proof It is proven that the execution of the workflow results in all the effects specified by the feature model configuration. Similarly it can be proved

that the workflow only requires preconditions specified by the feature model. The proof goes by contradiction. Assume that there is a feature f in a feature model configuration whose required effects are not satisfied by the workflow. This means that there exists an effect $e \in \mathcal{E}_f$ which is not satisfied. The selection of f in the feature model configuration means that $e \in S_{goal}$ which would mean that the planner has found a solution which does not satisfy all of its goal conditions, which is not possible. □

Theorem 1 *(Correctness of the Workflow)* Let C be a feature model configuration and w be a workflow generated based on Definition 3.4.5, w is a valid workflow and it consists of all the features specified in the feature model configuration.

Proof This theorem descends directly from Lemmas 1 and 2. □

It is important to mention that Lemmas 1 and 2 and Theorem 1 show that the service composition generated based on our proposed method has two characteristics: (1) the proposed service composition approach is guaranteed to find a service composition that would include all of the required functional requirements of the users if such a solution exists; and (2) the service composition is executable in that it respects the execution semantics of the services whereby all required preconditions of the services are satisfied before they are called. Therefore, the generated service compositions satisfy both the *verification* and *validation* requirements of an executable service composition.

Since the planner returns the solution as a sequence, the generated workflow dependency graph would be a chain of nodes. In the next section, how this workflow can be optimized to a more efficient one that considers parallel execution will be discussed.

3.5 Workflow Optimization

Although the generated workflow can be used to generate BPEL code, given that the AI planners produce strictly sequential plans, the generated workflow could benefit from potentially more efficient and valid plans which use parallel execution of operations when possible. Using parallelism in a service workflow can significantly affect the efficiency of the composed service [123]. Therefore, once a plan is generated by the AI planner, an additional step to optimize the workflow is taken.

Let us first provide an overview of how the optimization is performed before providing the formal details. The idea of our proposed approach for workflow optimization is to remove as many edges from the workflow as possible in order to relax the execution sequence without violating the semantics of the execution. In order to do this, this work assumes each edge in our workflow is a candidate for being removed from the workflow and evaluates whether it can be removed or not. The condition for removal is that the required preconditions for any forthcoming service execution in the workflow

```
1: function OPTIMIZE(workflow w = (E, N, E))
2: repeat
3:    W ← {}
4:    for all e = (n1, n2) ∈ E do
5:       E' ← E ∪ {(n', n2) s.t. (n', n1) ∈ E}
6:              ∪{(n1, n') s.t. (n2, n') ∈ E} − {(n1, n2)} :
7:       w' ← (E, N, E')
8:       if SAFE(w') then
9:          W ← W ∪ {w'}
10:      end if
11:   end for
12:   w ← SELECT(W)
13: until TERMINATIONCONDITION(w)
14: return w
```

Algorithm 1: Pseudo-code for workflow optimization.

are not violated; in other words, it can only remove an edge if its removal does not lead to the premature execution of services whose preconditions are not yet prepared. This is referred to as the *safety* condition. Therefore, simply stated, the process of workflow optimization is the iterative removal of edges from the workflow that respect the safety condition. Safety concept has been introduced in the partial-order planning[93]. In the partial order planning, the goal is finding the partial-order of action executions that is safe and solves the planning problem. This work uses the same concept and redefines it on a workflow of service invocations to ensure that the optimization process does not affect the validity of the initial workflow.

More formally, workflow optimization is performed by consecutive removal of the dependency edges in the workflow, which do not affect the

safety of the workflow. The details of our method for optimization have been shown in Algorithm 1. In the main loop in the algorithm (Lines 2-13), the dependency edges are removed consecutively until the termination condition (Line 13) is met. In each iteration of the loop, each dependency edge in the workflow is examined (Line 4) to see whether the workflow stays safe even after the removal of that dependency edge or not (Line 8). If so, the new workflow after removal of that edge is added to set W (Line 9). The new workflow after removal of a dependency edge would be a revised workflow which would not include the removed dependency edge but instead new dependency edges are added to preserve the connectivity of the workflow. This is done by adding edges from the start node of the removed dependency edge to the immediate nodes after the end node of the removed dependency edge and similarly the immediate nodes before the start node and the end node of the removed dependency edge (Lines 5-6). This ensures that the order of execution for the nodes before and after stay the same. After all dependency edges are examined, the best workflow is selected from the set W and the current workflow is replaced by that workflow (Line 12).

The definition of SELECT and TERMINATIONCONDITION in Algorithm 1 depends on the optimization method which has been selected. The definition for SAFE which is responsible for examining the safety of a workflow has been defined in Algorithm 2. The definition of this function has been inspired by the safety condition in partial order planning [93]. In this algorithm, the main loop iterates over all operation nodes of the workflow (Lines 2-16) and

```
1: function SAFE(workflow w = (E, N, E))
2: for all n ∈ N do
3:   for all p ∈ P(n) do
4:     safeCausalLinkFound ← false
5:     for all n' ∈ N do
6:       if AFTER(w, n', n) and p ∈ Q(n') then
7:         if ¬THREATEXISTS(w, n, n', p) then
8:           safeCausalLinkFound ← true
9:         end if
10:      end if
11:    end for
12:    if ¬safeCausalLinkFound then
13:      return false
14:    end if
15:  end for
16: end for
17: return true

18: function THREATEXISTS(workflow w, node n, node n',fact p)
19: for all n'' ∈ N do
20:   if ¬AFTER(w, n'', n') or ¬AFTER(w, n, n'') and p ∈ R(n'') then
21:     return true
22:   end if
23: end for
24: return false
```

Algorithm 2: Pseudo-code for examining safety of a workflow.

its immediate inner loop iterates over all of the facts that are required to be true as the precondition of the node (Lines 3-15). For each precondition fact p of each node n, this algorithm iterates over all the nodes in the workflow (Lines 5-11) in order to find an operation node n' which makes that fact true; meaning that $p \in \mathcal{Q}(n')$ and is executed before node n ensured by calling $\text{AFTER}(w, n', n)$ (Line 6). The function $\text{AFTER}(w, n', n)$ (Line 6) always

looks for a path from n' to n in the workflow. If such a path exists, it means that n will be executed after n'. The relation between node n' and n is called a causal link for p. If such a node is found, it is examined if a *threat* to that causal link exists (Line 7). If there is no threat to the causal link between two nodes, a safe causal link has been found (Line 8). If there exists no safe causal link for a precondition fact of a node (Line 12), the workflow is not safe.

A threat exists for a causal link when there exists an operation node that can be executed between the two nodes of the causal link and makes the fact of the causal link false. The function which examines a causal link for a possible threat has been shown in Algorithm 2. This algorithm works by iterating over all nodes in the workflow and analyzes if it can pose a threat to the causal link (Lines 2-6). A node can be considered a threat to a causal link if it does not execute before the start node or after the end node of the causal link and makes the related fact to that causal link false (Line 3).

Figure 3.9(a) shows an intermediate workflow in the process of optimization which is used as an example to illustrate a step of optimization. This figure also shows the annotations for the services involved in this workflow. In each step of the optimization, the optimization process looks for those edges in the workflow that their removal does not affect the safety of the workflow. This is done by removing different edges of the workflow and examining if there is a causal link without a threat for every precondition of each service

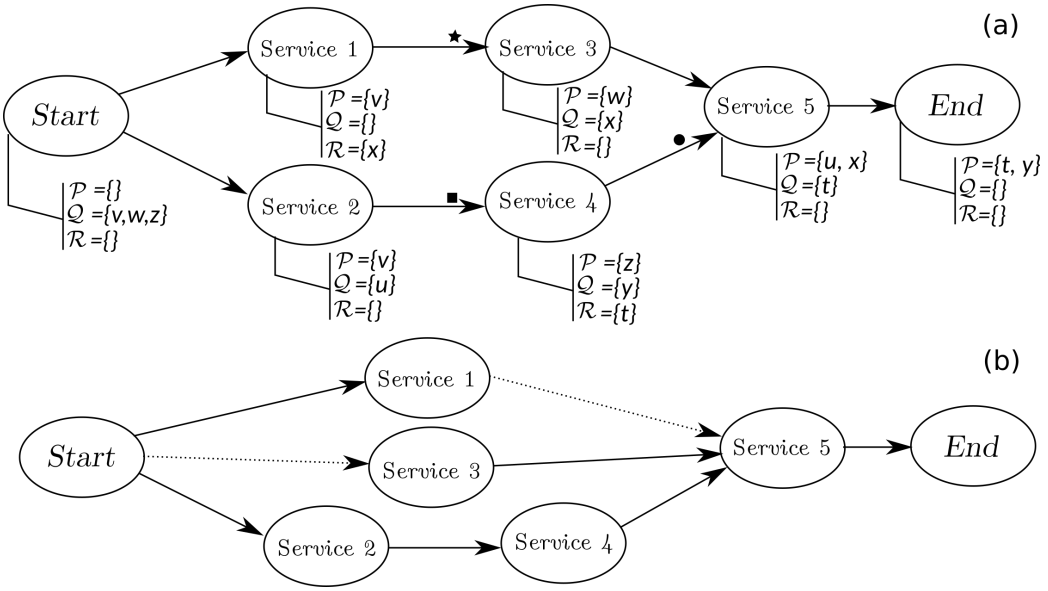


Figure 3.9: (a) A service composition workflow and constituent services annotations. (b) Resulting workflow after removing edge marked with ($\star$) from the workflow (a)

invocation. Figure 3.9(b) shows the workflow resulted after removal of the edge marked with ($\star$) in the workflow in Figure 3.9(a). The dashed edges are added to this workflow to maintain the correct order of service execution after the edge removal. The workflow after removal of this edge is no longer safe. Although there is a causal link for all the preconditions of services invocation in this workflow, removing the edge marked with ($\star$) introduces a threat to the precondition x of *Service* 5 invocation. Invocation of *Service* 3 is only causal link for precondition x of *Service* 5. However, invocation of *Service* 1 after *Service* 3 becomes possible in the new workflow which can make this causal link false before *Service* 5 invocation. Conversely, the

edge marked with (■) in Figure 3.9(a) is an example of an edge that can be removed without affecting the safety of workflow.

Theorem 2 *(Validity of the workflow)* Let w be a workflow and if SAFE(w) holds for it, then all preconditions for each operations node in w is satisfied before its execution.

Proof The proof goes by contradiction. Assume there exists a workflow w for which SAFE(w) does hold and there exists a sequence of execution s_w for this workflow such that there exists an operation $n \in s_w$ whose preconditions p do not hold. There could be two reasons that such a situation can happen: 1) There exists no operation in the execution sequence s_w which results in p before the of execution n; 2) there are nodes which result in p but there also exist nodes which result in $\neg p$ such that in this sequence $\neg p$ would be true before execution of n. It is shown that none of these situations is possible, as follows:

(1) If there exists no operation node in s_w before n which results in p, the condition for the if statement in Line 6 cannot be true for any $n' \in N$ because if this condition becomes true for a node n' it means that there exists a node which is always executed before n and results in p which contradicts the assumption that in sequence s_w there exists no operation that makes p true before n. If the condition in the if statement is not true for all the nodes in the workflow, *true* cannot be assigned to the variable $safeCausalLinkFound$. Therefore, the condition on Line 12 would be true

and the function should have returned false which is a contradiction with the assumption that SAFE(w) holds for this workflow.

(2) Assume that n_j is the last node in sequence s_w before n which results in $\neg p$ meaning $p \in \mathcal{R}(n_j)$. Additionally, there should exist nodes which result in p before execution of n based on (1). Assuming that n_i is the last one before n in sequence s_w, the condition $i < j$ must hold in order for $\neg p$ to be true when n is executed. It is shown that this is not possible by contradiction. Let $N(n, p) \subset N$ be the set of nodes that satisfy the condition on Line 6 which means that it is always executed before n and results in p. Since n_i is the last node before n which results in p and $i < j$, it can be said AFTER(w, n_j, n') would return false for every $n' \in N(n, p)$. Additionally, it can be said AFTER(w, n, n_j) would return false since n should be executed before n_j on all possible execution sequence of the workflow in order for AFTER(w, n, n_j) to be true and s_w is one sequence that n_j did not execute after n. Using these two facts, it can be inferred that THREATEXISTS(w, n, n', p) on Line 7 would be *true* for every $n' \in N(n, p)$. This is because AFTER(w, n_j, n') and AFTER(w, n, n_j) do not hold and the condition $p \in \mathcal{R}(n_j)$ must hold which causes the condition of the if statement on Line 20 to be true and consequently causes the THREATEXISTS(w, n, n', p) to be true. Since it is required for THREATEXISTS(w, n, n', p) to be false for one $n' \in N(n, p)$ in order for a safe causal link to be found, SAFE(w) would return false which is a contradiction. □

Although it is proven that preconditions of all workflow operation nodes will be satisfied, it is still needed to make sure that the whole workflow precondition and effects stay the same during optimization. This can be done by a small modification in the input workflow: a new start operation node with workflow preconditions as its effects is added to the beginning of the workflow and an operation node with workflow expected effects as its preconditions is added to the end of the workflow. Considering that the optimization will not affect precondition satisfaction, it can be easily proven that if the start and end node is removed from the workflow after optimization, the resulting workflow will satisfy its expected preconditions and effects. For example, these nodes have been added to the workflow in Figure 3.9. The effects for the *Start* node represent service preconditions and preconditions for *End* node represent the expected effects of the service composition. Not having these nodes may result in edge removals which threaten service preconditions and effects. For example, the edge marked with (•) cannot be removed in Figure 3.9(a) since it introduces a threat to the precondition t of *End* node. However, the workflow remain safe if this edge get removed when there is no *End* node. If this edge gets removed, the *Service* 4 can be executed after *Service* 5 and service composition execution will no longer result in expected effect t in that scenario.

3.6 Generating BPEL Code

Two main approaches have been used in the literature for representing service compositions: approaches based on block structure [100, 144] and graph-based approaches [112, 88]. In the block structure approaches, the process is represented with hierarchical blocks which decompose the process structure to smaller blocks that can themselves be built of smaller blocks. Examples of such models are Windows Workflow Foundation (WF) [100], Microsoft XLANG [144], and conventional programming languages. In the graph-based models, the structure of the dependency between operations is represented using arcs between the operations similar to how workflow is represented in this work. Examples of such models are BPMN [112] and IBM's WSFL [88]. BPEL has been built based on IBM's WSFL [88] which is graph-based language and Microsoft XLANG [144], which is a block structured language. BPEL has inherited from both these languages and therefore has the capability to represent a service composition in both ways. The generated workflow in our work can easily be represented in BPEL using a graph-based structure using <*link*> tags. For example, Figure 6.7-A shows the graphical representation of a solely graph-based BPEL code for the workflow in Figure 3.4. However, the graph-based representation of a workflow is usually hard to comprehend and is often not chosen as a way to represent a process [149, 122].

In BPEL, the structured way to represent dependencies between operations other than control operations are <*flow*> and <*sequence*> structures. Therefore, some approaches have proposed methods for building fully block structured BPEL using only those structures from graph-based languages such as BPMN [112]. However, the dependency in a graph-based language cannot be fully captured through only hierarchical use of these two structs. That is the reason why BPEL provides the link tag which allows the representation of graph-like dependencies between nested children of the flow activity. Considering that, it has been shown that using more graph-based structures (i.e. link tags) in the BPEL process reduces its readability [122]. For this reason, a method for converting our generated workflow into a BPEL process is proposed which minimizes the number of graph-based structures while preserving the actual dependency represented in the workflow graph.

In the proposed approach, a BPEL process is built which is solely graph-based using the <*link*> structure of the BPEL process and then an incremental approach is taken to find the part which can be replaced with one of the block structure types. Such parts will be replaced with the block structure activity and the process continues until no more replacement is possible. In the following, BPEL processes and the proposed approach are formally defined.

Definition 10. (BPEL Process) A BPEL process $p = (V, a)$ is a pair where

- V is the set of variables used in BPEL operations;
- a is an activity which can be :
  - A flow $(A, \mathcal{L})$ made of a set of sub-activities A such that these sub-activities can be executed in any order. Synchronization between descendant activities of a flow activity is done through links in $\mathcal{L}$. Each link $l \in \mathcal{L}$ is a pair (a, a') that enforces the execution of a' after a;
  - A sequence $< a_1, ..., a_n >$ which is an activity made of a sequence of sub-activities that should be executed in the order specified in the sequence;
  - An atomic activity which can be any of the BPEL atomic operations such as invocation of a service.

This definition of BPEL is abstract and has only been mentioned here in order to capture the parts of a BPEL process which this approach focuses on. A BPEL process $p = (V, a)$ can be built from a workflow $w = (E, N, \mathcal{E})$ where the set V corresponds to the elements of the entities set E and $a = (A, \mathcal{L})$ is a flow where A is a set of atomic activities corresponding to operations in nodes in N and $\mathcal{L}$ is the equivalent of $\mathcal{E}$ defined over atomic activities of the operation nodes.

This work proposes a process that takes a BPEL process as input and attempts to reduce the number of graph-based structures in the process and

1: **function** OPTIMIZEBPEL(flow $f = (A, \mathcal{L})$)
2: $C \leftarrow$ FINDFOLDINGCANDIDATES(f)
3: **while** $C \neq \emptyset$ **do**
4: $c \leftarrow$ SELECTBESTCANDIDATE(C)
5: $f =$ FOLD(f, c)
6: $C \leftarrow$ FINDFOLDINGCANDIDATES(f)
7: **end while**

8: LINK $(f) = \mathcal{L}(f) \cup \{$LINK$(f')$ s.t. $f' \in A(f)\}$
9: UNFOLDABLELINK $(f) = \{l = (a, a') \in \mathcal{L}(f)$ s.t. $a \notin A(f)$ or $a' \notin A(f)\} \cup \{$LINK(f') s.t. $f' \in A(f)\}$
10: **function** SELECTBESTCANDIDATE(flow $f = (A, \mathcal{L})$, activityset C)
11: $bc = \emptyset$
12: **for all** $a \in C$ **do**
13: $f' =$ FOLD(f, c)
14: $sizeL =$ | LINK (f')| , $sizeUL =$ | UNFOLDABLELINK (f')|
15: **if** $bc = \emptyset$ or $sizeUL < sizeUL_{bc}$ or ($sizeUL_{bc} = sizeUL$ and $sizeL < sizeL_{bc}$) **then**
16: $bc = c$, $sizeL_{bc} = sizeL$, $sizeUL_{bc} = sizeUL$
17: **end if**
18: **end for**
19: **return** bc

Algorithm 3: Algorithm for BPEL process optimization.

replaces them with block-based structures with the goal of reducing the number of graph-based structures. This is done through a *fold* operation in which the set of activities whose dependencies are represented using link tags are replaced with a block structured activity. This is similar to the work presented in [149]. However in [149], BPEL generation requires human intervention during its process since that method is only able to fold a completely well-structured set of activities. A set of activities is called well-structured when all of its members dependencies can be represented using a block structure

1: **function** FindFoldingCandidates(flow f)
2: $C \leftarrow \emptyset$
3: **for all** $a \in A(f)$ **do**
4: $C \leftarrow C \cup$ FindSequences$(f, a) \cup$ FindFlow(f, a)
5: **end for**
6: **return** C

7: **function** FindSequence(flow f, activity a)
8: $S \leftarrow \emptyset$
9: **for all** a' s.t. $(a, a') \in \mathcal{L}(f)$ **do**
10: $S \leftarrow S \cup \{< a, s >$ s.t. $s \in$ FindSequence$(f, a')\}$
11: **end for**
12: **return** S

13: **function** FindFlow(flow f, activity a)
14: $F \leftarrow \{a\}$
15: **for all** $a' \in A(f)$ **do**
16: **if** Preceding(f, a) =Preceding(f, a') and Following(f, a) =Following(f, a') **then**
17: $F \leftarrow F \cup \{a'\}$
18: **end if**
19: **end for**
20: **if** $|F| < 2$ **then**
21: **return** $\emptyset$
22: **end if**
23: $\mathcal{L}' = \{(a, a')$ *s.t.* $(a, a') \in \mathcal{L}(f)$ *and* $a, a' \in \{$SubActivity(f') s.t. $f' \in F\}$
24: **return** $\{(F, \mathcal{L}')\}$

Algorithm 4: Algorithm for Finding Folding Candidates.

component and therefore all graph-based structures (i.e., link structures) in which they are involved are removed when it is folded into that component. Therefore, it requires human intervention when such a set of activities cannot be found. Our proposed method is fully automated and continues to look for the sets of activities which are not well-structured but can be folded and their folding reduces the graph-based structure.

Algorithm 3 shows the function for the proposed folding algorithm. The algorithm finds the parts of the process which can be folded into a block-structured activity by calling the FindFoldingCandidates function (Line 2). This function gets a BPEL process as input and returns a set of possible folding candidates which are block-structured activities. The main loop of the function (Lines 3-7) repeats until there are no more folding candidates for creating a block structure. In this loop, the most suitable candidate for folding is selected by calling SelectBestCandidate (Line 4). After the best folding candidate has been selected, the activities in BPEL process are folded by calling the Fold method (Line 5). This work uses the same folding mechanism as the one used in [149], therefore a formal definition for the fold method is not provided. After that, folding candidates in the result process are found and the loop repeats.

The SelectBestCandidate function has been defined in Algorithm 3. This algorithm uses two functions Link (Line 8) and UnfoldableLink (Line 9). The Link function returns the total number of links which have

been used in the flow activity as well as the nested flows in it. The UnfoldableLink function returns those links in the flow structure which cannot be folded in the rest of the optimization. These are those links which begin or end in an activity which is not the immediate child of the flow and nested in one of the activities of the flow. This algorithm works by iterating over all folding candidates (Line 12-18). For each folding candidate, the flow after folding that candidate is found (Line 11). In this new flow, the number of links and unfoldable links in that flow structure is found (Line 14). If the number of unfoldable links in this candidate is less the than best candidate which has been found until now or it has an equal number of unfoldable links but fewer links, the current candidate is selected as the best candidate (Lines 15-17). This algorithm uses a greedy selection method whereby the candidate which results in the fewer immediate unfoldable links and most reduction of the links is selected. The selection method has shown to be effective in our evaluations (refer to Section 6.1.3).

Algorithm 4 defines the function for finding folding candidates. The algorithm works by iterating over all activities in the main flow and finding flow and sequence structures related to each activity (Line 2-5). Specifically, for each activity a, it finds sequence structures which start from a and the flows which a can be a part of. The algorithm then adds all found structures to the set C which stores the potential candidates (Line 4). Sequences starting from an activity are found by calling the FindSequence function. In this function, for each link element starting from an input activity, the sequences from

the end activity of the link are found and a sequence by adding a starting activity to all those sequences is created (Line 9-11). Function FindFlow finds a flow activity that an activity can be part of if such a flow activity exists. This function starts with a result set F with the input activity as its only member (Line 14). It then iterates over all the activities in the flow to find activities which can form a flow structure with this activity (Line 15-19). An activity can form a flow with the input activity when the set of all activities with a link to and from it would be the same. The functions Preceding and Following return the set of preceding and following activities for an activity, respectively (Line 16). If that is the case, then that activity will be added to the result set F. After examining all activities, if the cardinality of the result set is less than two, it means that the input activity cannot form a flow structure with any other activity and therefore the function returns an empty set. Otherwise, it will create a flow structure with the result set and with all the links in the main flow whose start and end activities are in the sub-activities of the newly created flow children.

3.7 Summary

This chapter proposed a method for generating executable service composition code based on functional requirements specified as a feature model configuration. In order to do so, the software product line engineering two-phase process was adopted. In the domain engineering phase, a set of models

including feature model, services, and a number of models connecting them are developed. In the application engineering phase, the services are then automatically composed based on AI planning so that a workflow is generated. The generated workflow is not guaranteed to have desirable properties such as execution time. For this purpose, a method has been proposed that introduces parallelism into the workflow while maintaining its validity. The optimized workflow is then converted into WS-BPEL code that is directly executable.

3.8 Related Publications

- Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Automated Composition of Service Mashups Through Software Product Line Engineering." *International Conference on Software Reuse*, Springer, Cham, 2016.

- Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Automated Composition and Optimization of Services for Variability-Intensive Domains", *Journal of Systems and Software* (Accepted)

Chapter 4

Run-time Adaptation of Service Compositions

The main objective of this chapter is to develop an automated service composition reconfiguration method that can react to violations in functional or non-functional requirements. Specifically, this chapter makes the following contributions:

- It proposes an automated failure mitigation method, which focuses on finding an alternate feature model configuration for the service composition that recovers critical functional and non-functional requirements. This method is able to find an alternate service composition to replace the failed service composition.

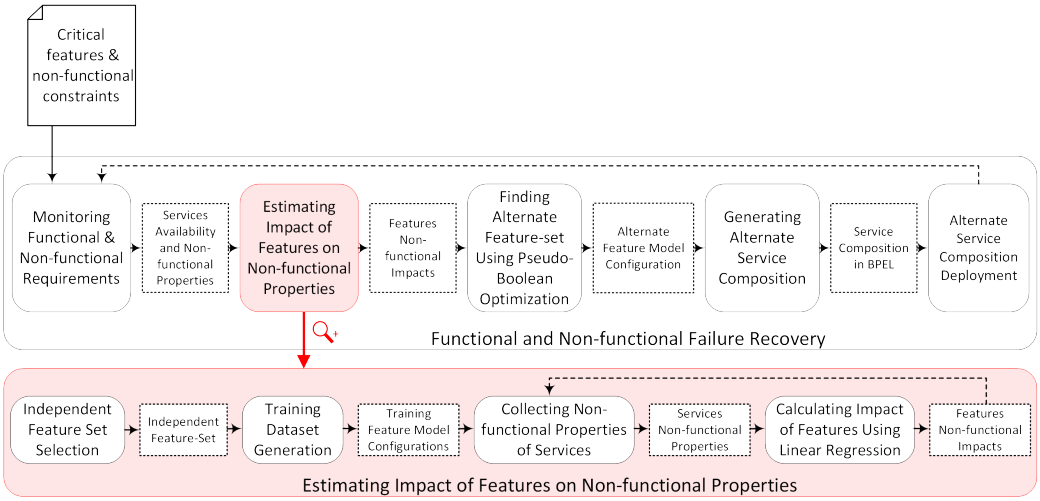


Figure 4.1: General Overview of the proposed adaptation approach.

- In order to enable finding an optimal feature model configuration with desired non-functional properties at runtime, it proposes a method which is able to estimate the effect of each selected feature on the non-functional properties of the service composition. Additionally, the proposed method is able to update its estimates at runtime as the non-functional properties of the constituent services of the service composition change.

The next section in this chapter provides an overview of the proposed approach and describes how the rest of the chapter is organized.

4.1 Approach Overview

Some existing failure recovery methods introduce redundancy into the service composition [36, 31] while others find an alternate composition for satisfying the requirements through realtime re-planning [71, 70]. Our work aligns with the second line of research in that it attempts to enable re-selection of features of a service composition at runtime such that it can satisfy a user's critical functional and non-functional requirements while operating in a degraded state. The proposed method performs adaptation by automatically choosing an alternative feature set that satisfies a user's critical requirements and at the same time circumvents those features that caused the failure. A feature can cause a functional failure by using a faulty or unresponsive service in its realization, which can also cause a violation of the non-functional constraints.

Figure 4.1 provides an overview of the approach that is proposed for recovering from such failures. The only input that the user should provide to enable adaptation is the set of critical features and non-functional constraints. Adaptation uses these inputs in addition to the domain model and the feature model configuration, which specifies desirable features, to decide about the best adaptation strategy in response to a failure. The top box in Figure 4.1 shows the activities performed to enable self-healing for a service composition. The process is an iterative process (as shown by a dashed line going from the last activity in the process to the first one) which starts by an activity which monitors the functional and non-functional requirements spec-

ified by the user and possibly detects the occurrence of any failures. In case of a failure, the process switches to those activities which are responsible for finding an alternative service composition which replaces the failed service composition. This assumes mechanisms for measuring service availability and quality of service metrics are already available and, as such, methods for measuring them would be outside the scope of this work.

In the proposed approach, the first activity that is performed is estimating the effect of features on non-functional properties of the service composition. This step is required since the adaptation mechanism decides about features when performing adaptation and needs to know how selecting a feature affects non-functional properties of the service composition. The red box in the bottom of Figure 4.1 shows the steps that should be taken in this activity. The first step of this activity is selecting independent feature sets. An *independent feature set* is a set of features in the feature model where the selection status of all features in the service composition can be determined by knowing the presence of such features in a feature model and there is no feature in this feature set whose presence can be determined by knowing the presence of other features in this set using a linear relation. In the next step, a dataset of feature model configurations and their corresponding service compositions are generated. Based on this dataset, the extent to which each of the features in the independent feature set contributes to non-functional properties of the final service composition is estimated. It can be shown that knowing the contribution value of features is enough for estimating the

non-functional properties of service compositions and also linear regression converges to a single value for contribution values of features. The first two steps of this activity only need to be executed once and the next two steps need to be executed in each iteration of the adaptation process. These steps have been discussed in detail in Section 4.3.

The next activity in the adaptation process is to find an alternative feature set to replace the failed feature set by reducing this problem into a *pseudo-boolean optimization* problem that can be solved using an optimization solver. A solution to this optimization problem delivers critical features and satisfies non-functional constraints, if and when possible, while minimizing loss of desired features. The details of this step of the process are discussed in detail in Section 4.4. After finding an alternative feature model configuration as a result of the pseudo-boolean optimization problem, the replacement service composition is generated. For generating the alternate service composition based on it feature model composition, the method that has been discussed in the previous chapter is used. After generating the alternative service composition, it will be deployed in the execution environment and new requests are redirected to it.

In the following section, a case study is provided which will be used as a running example in this chapter. Afterwards, the details of the proposed method for determining the contribution of each feature towards the non-functional properties of a service composition are provided. On this basis

and in a subsequent section, the proposed method for service composition adaptation as a response to violations of functional and non-functional requirements is described.

4.2 Case Study

Figure 4.2 shows a feature model for a family of services, which process and upload an incoming image on a website. An application instance in this family needs to have the mandatory feature of storage and can provide optional features of tagging, filtering, and editing. The tagging feature processes an image and finds keywords that describe the image to be used in different operations such as image search. There are two types of tagging in this product line, namely metadata-based and external. In meta-data tagging, information about objects and text in the image is used to create the tags for the image. In the external tagging feature, an external service is used to create the tags for the image. The filtering feature provides mechanisms for detecting nudity or profanity within the image. Similar to the filtering feature, the editing feature provides watermark or face blur capabilities. The watermark feature watermarks an arbitrary text on an image and the face blur feature obscures faces in the image. The marks on the bottom right corner of the features in Figure 4.2 are not part of the feature model notation and are used here to refer to different feature model configurations. For example, the

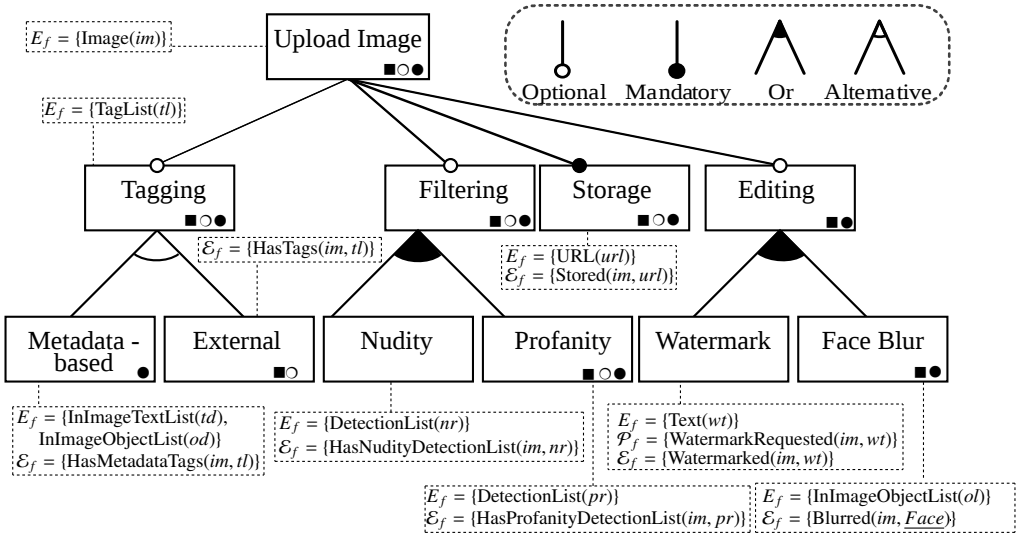


Figure 4.2: An annotated feature model for the upload image service family.

features marked with (•) in Figure 4.2 show the set of features that belong to a valid feature model configuration.

Figure 4.5 shows the visualization of the BPEL process for the feature model configuration containing features marked with (•) in Figure 4.2. Service invocations in this BPEL process have been organized using three types of structures: flow, sequence, and links. The invocations in the flow structure can be invoked in any order while the invocations in a sequence structure should be executed in order. The link structure is used when there is a precedence relation between two invocations which cannot be represented using a sequence. In this example, after an image is received, two sequences of activities are performed in parallel in a flow structure. The first sequence calls a service for detecting the objects in the image, then calls a service for filtering

Context Model $c = (c_T, c_E, S)$

$c_T = (\Theta, \Phi, \mathcal{F})$

Θ = {Image, TagList, ObjectType, InImageTextList, InImageObjectList, Text, URL, DetectionList}

Φ = {HasTags, HasText, HasObject, Watermarked, WatermarkRequested, HasType, Blurred, HasProfanityDetectionList, HasMetadataTags, HasNudityDetectionList, Stored}

$\mathcal{F}$ = {HasText(*Image*, *InImageTextList*), HasObject(*Image*, *InImageObjectList*), Watermarked(*Image*, *Text*), Stored(*Image*, *URL*), Blurred(*Image*, *ObjectType*), WatermarkRequested(*Image*, *Text*), HasTags(*Image*, *TagList*), HasType(*InImageObjectList*, *ObjectType*), HasProfanityDetectionList(*Image*, *DetectionList*), HasMetadataTags(*Image*, *TagList*), HasNudityDetectionList(*Image*, *DetectionList*)}

$c_E = (E, \mathcal{T})$ E = {Face} $\mathcal{T}$ = {ObjectType(*Face*)}

S = {}

Figure 4.3: The context model for the upload image service family

objects based on the type, only keeping the list of face objects in the image, and blurring faces by calling a face blur service and finally uploading the image on an image upload server. The second sequence calls a text extraction service and, using the text extracted from the image, it calls two services in parallel in a flow structure. One detects profanity in the image and the other generates meta-data based on the text and objects detected in the image. There are two precedence relationships between the activities in the main flow which are enforced with links. First, the invocation of the meta-data generation service should be done after the invocation of the object detection service since it is dependent on the data returned by the first service. Second, the invocation of the text-extraction service should happen before blurring the image since it may affect the text extraction service performance. After performing this sequence, the service composition will return the generated tag list, detected profanity and the url of the image that has been uploaded.

Services

ObjectDetection: I = {Image($image$)} O = {InImageObjectList($objects$)}
$\mathcal{P}_I$ = {¬Blurred($image$, ?$type$), ¬Watermarked($image$, ?$text$)} $\mathcal{Q}_I$ = {HasObject($image$, $objects$)}

TextExtraction: I = {Image($image$)} O = {InImageTextList($texts$)} $\mathcal{P}_I$ = {¬Blurred($image$, ?$type$), ¬Watermarked($image$, ?$text$)}
$\mathcal{Q}_I$ = {HasText($image$, $texts$)}

DetectProfanity: I = {Image($image$)} ,O = {DetectionList($pList$)} $\mathcal{P}_I$ = {Blurred($image$, ?$type$), Watermarked($image$, ?$text$)}
$\mathcal{Q}_I$ = {HasProfanityDetectionList($image$, $pList$)}

GenerateTagMetadata: I = {Image($image$), InImageObjectList($objects$), InImageTextList($texts$)} O = {TagList($tagList$)}
$\mathcal{P}_I$ = {HasObject($image$, $objects$), HasText($image$, $texts$)} $\mathcal{Q}_I$ = {HasMetadataTags($image$, $tagList$)}

BlurObjects: I = {Image($image$), InImageObjectList($objects$), ObjectType($type$)} O = {Image($blurredImage$)}
$\mathcal{P}_I$ = {HasType($image$, $type$), ¬Stored($image$, ?url)} $\mathcal{Q}_I$ = {Blurred($blurredImage$, $type$)}

FilterObjects: I = {InImageObjectList($objects$), ObjectType($type$)} O = {InImageObjectList($filteredObjects$)}
$\mathcal{P}_I$ = {HasObject(?$image$, $objects$)} $\mathcal{Q}_I$ = {HasType($filteredObjects$, $type$)}

DetectTextProfanity: I = {InImageTextList($texts$)} O = {DetectionList($pList$)} $\mathcal{P}_I$ = {HasText(?$image$, $texts$)}
$\mathcal{Q}_I$ = {HasProfanityDetectionList($image$, $pList$)}

GenerateTagExternal: I = {Image($image$)} O = {TagList($tagList$)} $\mathcal{Q}_I$ = {HasTags($image$, $tagList$)}

UploadImgUr: I = {Image($image$)} O = {URL(url)} $\mathcal{Q}_I$ = {Stored($image$, url)}

UploadscriptsizePic: I = {Image($image$)} O = {URL(url)} $\mathcal{Q}_I$ = {Stored($image$, url)}

Figure 4.4: Part of the service repository and annotations for the upload image service family.

4.3 Features and Non-functional Properties

Although features represent functional properties of a system [27], the presence of a feature in a service composition can affect the non-functional properties of that service composition as well. For example, the presence of the Face Blur feature in our sample service composition may result in higher response time since realizing the Face Blur feature may require additional processing on the input image. From a practical perspective, it is possible to measure the non-functional properties of features by recording the performance metrics of the services that implement those features. However, when the number of services and features within the domain increases, i.e., the domain becomes highly variable, measuring the non-functional properties of all the domain features becomes impractical. For this reason, it is desirable

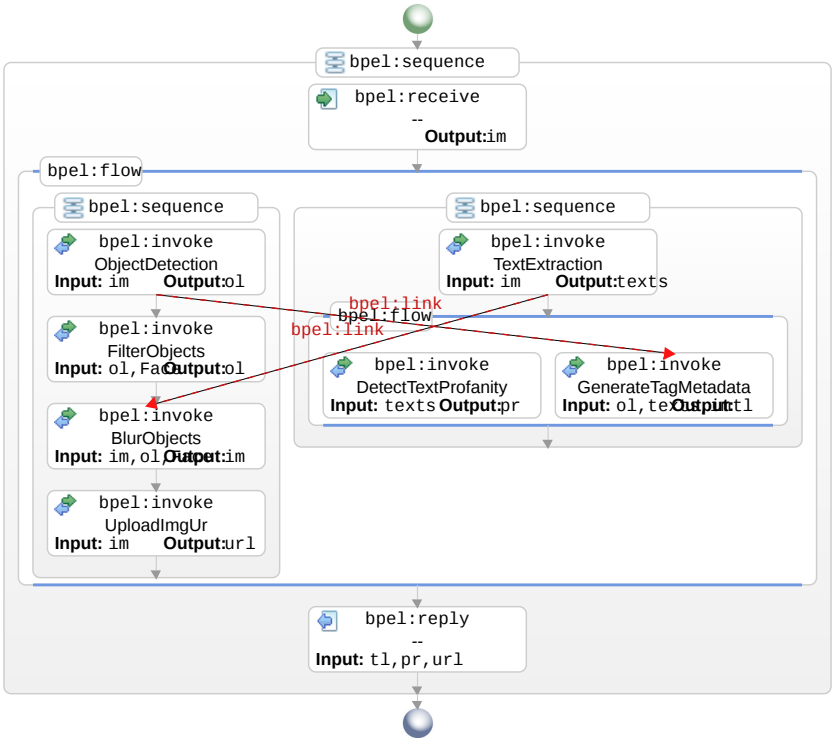


Figure 4.5: BPEL process visualization for a possible service composition satisfying requirements of the feature model configuration in Figure 4.2.

to estimate the non-functional requirements of a feature within the context of a service composition. Such an estimation function can be used in deciding about optimal alternative features in the adaptation process. Given the highly dynamic nature of services and their changing non-functional properties, it is desirable for this estimation function to take such dynamism into account.

The effect of each feature on non-functional properties comes from the services that realize the feature. However, no direct link between features

and services can be assumed since one feature can be realized using one or more services and likewise a service might implement anything from a part of a feature to a collection of features. Therefore, all possible configurations of a feature model should be realized in order to find the function that specifies the exact value of a non-functional property for each feature model configuration. This approach is not practical since the possible number of configurations can grow exponentially as the feature model grows larger. Furthermore, the function needs to be lightweight to be recalculated often as the non-functional properties of the constituent services change. Therefore, this work proposes an estimation function which receives a feature model configuration as input and estimates the value of the non-functional properties of the composed service composition based on the features available in that configuration.

The work in [18] recommends annotating features with the effects they will have on the values of non-functional properties of the system when present. It assumes that the non-functional property values of each feature are independent of the other features. Therefore, the value of a non-functional property of a system containing a set of features represented in a feature model configuration is calculated using the annotations of the selected features and an *aggregation function* [101] which aggregates the values of the non-functional properties of the individual features. While this work does not assume that non-functional property values of features are constant, it makes a similar assumption about how features' non-functional properties can be aggregated. For example, Figure 4.6-A shows that the features in

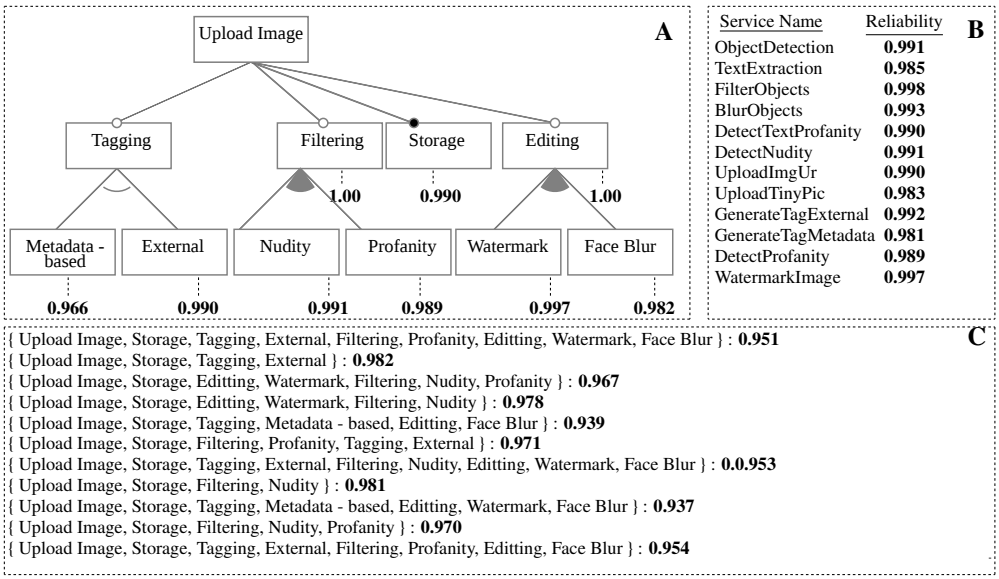


Figure 4.6: **A.** Feature model annotated with contribution value of each feature to the reliability of the service composition. **B.** Reliability of the individual services in the service family **C.** An example of a dataset used for estimating the contribution value of each feature to the reliability of the service composition.

the model shown in Figure 4.2 are annotated with values showing how they contribute to the reliability of the service composition. The reliability value for a service composition can be calculated by aggregating these values using the product operator.

This work employs a sum aggregation function for aggregating non-functional properties. Those non-functional properties whose aggregation function is the product operator, such as reliability, can be converted to summation by working with their natural logarithm. It should be noted that

the relation between features and non-functional properties might be more complex than the relation assumed in this work. However, this work suggests that this relation can be estimated using a linear function over feature presence with sufficient performance for its purpose. As such, the function for estimating the value of a non-functional property of a service composition can be represented as:

$$Q(C) = \sum_{i \in \{1,..,n\}} p_i c_i \quad c_i = \begin{cases} 0 & f_i \notin C \\ 1 & f_i \in C \end{cases} \tag{4.1}$$

In this equation, $Q(C)$ represents the function for determining a non-functional property of the service composition realizing feature model configuration C. C is a subset of features from feature model F, which respects the structural and integrity constraints. c_i can be zero or one based on the presence of the ith feature (f_i) in the feature model configuration C. Similarly, p_i is the contribution value of the ith feature to the non-functional property. Based on this equation, it is only needed to find the values for $P =< p_1, ..., p_n >$ in order to be able to determine the value of a given non-functional property for a service composition.

Considering a linear relationship between the presence of the features and the value of a non-functional property, linear regression is a suitable option for finding the p_i values for the $Q(C)$ function. The linear regression approach is

used for modeling the relationship between a dependent variable and a set of independent variables affecting the value of that dependent variable having a linear contribution. It allows estimating the contribution value for each of the independent variables to the value of the dependent variable using a limited number of dependent variable and independent variable set values. The relation between independent variables and the dependent variable can defined as show below:

$$y = \sum_{i \in \{1,..,n\}} \beta_i x_i + \alpha \tag{4.2}$$

where x_i is an independent variable, y is the dependent variable, β_i is a regression parameter, and α is the intercept. Having a dataset $D = \{d_1, ..., d_m\}$ and $d_j =< \bar{y_j}, x_{j1}, ..., x_{jn} >$ where $\bar{y_j}$ is the actual value for the dependent variable when dependent variable values are $< x_{j1}, ..., x_{jn} >$, linear regression is able to estimate the values for the β_i and α parameters.

In Equation 4.2, the intercept can also be viewed as another regression parameter where its corresponding independent variable is always one. In the case of our problem, the intercept for regression is not required since the root feature in a feature model configuration is always selected and acts as an intercept. Therefore, this relation completely matches the relation between a feature selection and a non-functional property defined in Equation 4.1. Hence, linear regression can be used in order to find $P =< p_1, ..., p_n >$

which represents contribution values of all features where p_i is the contribution of the ith feature. This can be done assuming that there is a set $D = \{(C_1, q_1), ..., (C_m, q_m)\}$ where pair (C_i, q_i) shows a feature model configuration and its corresponding value for a non-functional property.

This work employs the *Ordinary Least Squares (OLS)* regression method [136], which is widely used for estimating regression parameters. Assuming that there is a dataset, the ordinary least squares method is able to find an assignment to the regression parameters, which minimizes the squared errors in that dataset. Having dataset $D = \{d_1, ..., d_m\}$, the error for estimating the jth instance of the data would be $e_j = |y_j - \bar{y}_j|$. As such, ordinary least squares will be able to find an assignment to the regression parameters such that the following is minimized:

$$\sum_{j \in \{1,..,m\}} e_j^2 \tag{4.3}$$

This means that the sum of squared error over all instances of the data is minimized. In the case of our problem, $e_j = |q_j - Q(C_j)|$ is the error in estimating the value for the non-functional property using the estimation function and $P =< p_1, ..., p_n >$ is found such that the sum of squares of all errors is minimized.

As a requirement for OLS to converge to a single assignment to $P =< p_1, ..., p_n >$, which minimizes Equation 4.3, there should not be a linear dependence between values of independent variables in the training set. Having a dataset $D = \{d_1, ..., d_m\}$ where $d_j =< \bar{y_j}, x_{j1}, ..., x_{jn} >$, this means for the following system of linear equations:

$$\begin{aligned} a_1 x_{11} + a_2 x_{12} + ... + a_n x_{1n} = 0 \\ \vdots \\ a_1 x_{m1} + a_2 x_{m2} + ... + a_n x_{mn} = 0 \end{aligned} \tag{4.4}$$

where the goal is to find possible assignments to $A =< a_1, ..., a_n >$, the only possible solution should be zero assignments to all a_i variables. However, creating such a dataset of feature model configurations for a feature model may not be possible considering the fact that the relations between features are governed by structural and integrity constraints. Examples of such situations include the following:

- Mandatory relation between a parent feature and a child feature enforces the selection or deselection of a feature as a result of the selection or deselection of the other feature. Therefore, a parent and child always have the same selection status. Assuming that the ith feature is the mandatory child of the jth feature, the linear relation $c_i - c_j = 0$

will be always true in all instances of the dataset of feature model configurations.

- If the ith feature is an alternative group and has two child features, which are the jth and kth features, the linear relation $c_i - c_j - c_k = 0$ holds for all instances of the dataset of feature model configurations.

There can be more complex situations where feature model constraints do not allow for creating a dataset in which a linear dependence does not hold. In order to address this issue, this work proposes a method, which selects a subset of features for which the constraints of the feature model do not enforce any linear dependence and show that a linear function defined over the presence of all features in the feature model can also be defined over the new set of features. This can be formally defined as follows:

Assuming that there is a feature model fm which has the feature set F, the goal is to find a set of features $F' = \{f'_1, ..., f'_{n'}\} \subseteq F$ such that (1):

$$\nexists A \in \mathbb{R}^{n'} - \{< 0, ..., 0 >\} \; s.t. \; \forall C \subseteq F \; s.t. \; \textsc{Valid}(C, fm) \Rightarrow$$
$$a_1 c'_1 + ... + a_{n'} c'_{n'} = 0 \qquad (4.5)$$

and (2) assuming $C' = F' \cap C$:

$$\forall P \in \mathbb{R}^n \Rightarrow \exists P' \in \mathbb{R}^{n'} \; s.t. \; \forall C \subseteq F \; s.t. \; \textsc{Valid}(C, fm) \Rightarrow Q(C) = Q'(C') \tag{4.6}$$

holds.

In Equation 4.5, A represents possible coefficient values for features in feature subset F' and $\textsc{Valid}(C, fm)$ examines if a configuration C is valid in terms of the constraints represented in fm. Therefore, the equation expresses that there is no assignment to $A =< a_1, ..., a_{n'} >$ except all zeros such that for all valid configurations of the feature model, linear dependency holds for this subset of features. Equation 4.6 denotes that for all assignments to contribution values P, there also exists an assignment to P' such that $Q'(C')$, which has been defined on a the subset of features F', has the same value as $Q(C)$ for all valid feature model configurations.

In order to find the set F', Algorithm 5 has been proposed. This algorithm works on an input feature model and returns a set of features for which the conditions in Equations 4.5 and 4.6 hold. The function works by incrementally building a dataset for which no linear dependency holds. In the process of building this dataset, the linear dependencies in the current dataset are considered as candidates for the linear dependencies that holds for all configurations of the dataset. If this is not the case, a feature model configuration which does not hold the candidate linear dependency is added

```
1: function FindIndependentFeatureSet(FeatureModel fm)
2: D ← {}, F' ← fm.F
3: while GetNumberOfSolutions(D) ≠ 1 do
4:    A ← Solve(D)
5:    C ← FindConfiguration(A, fm)
6:    if C ≠ {} then
7:       D ← D ∪ {C}
8:    else
9:       F_R ← {f_i s.t. f_i ∈ F' and a_i ≠ 0}
10:      f ← SelectAMemberRandomly(F_R)
11:      F' ← F' − {f}
12:   end if
13: end while
14: return F'
```

Algorithm 5: The algorithm for finding a subset of features F' for which linear dependency does not hold.

to the dataset. Otherwise, one of the features involved in the linear dependency is randomly removed from the feature set and the process continues.

The algorithm starts by assigning an empty set of feature model configurations as the dataset D where linear dependency does not hold (Line 2) and continues by incrementally adding new instances of feature model configurations to it in the while loop. Conversely, the set of all features F is assigned to the result feature set F' and features are incrementally removed from it in the while loop. This while loop continues until there is only one solution for the system of linear equations which is built using the existing feature model configurations in the dataset. Therefore, there will be no linear dependence between the selected features F' in the dataset after exiting the loop. In Line 4 of the loop, function Solve(D) is called, which finds one of these assign-

ments using an existing method for solving the system of linear equations and assigns it to A. Line 5 examines if there exists a feature model configuration for feature model fm, which does not satisfy the linear equation built using A by calling function FindConfiguration(A, fm). In Line 7, if there exists such a feature model configuration, it is added to the dataset of feature model configurations D. Otherwise, there is a linear dependence between those features whose corresponding a_i in A is non-zero. Therefore, those features are assigned to a set F_R in Line 9. Therefore, F_R will include those features in F' which have linear dependency. This linear dependency is broken by selecting a feature f randomly in Line 10 and removing it from F' in Line 11. Function SelectAMemberRandomly simply selects the member randomly from a set.

In the following, it is formally proven that the set of features in F' after exiting the while loop will contain those features, which satisfy the requirements in Equations 4.5 and 4.6.

Lemma 1 *(No Linear Dependency)* There exists no linear dependency between the set of features F' returned by Algorithm 5.

Proof. The proof goes by contradiction. Let us assume that there exists an A for which linear dependency holds for all feature model configurations. Furthermore, it is assumed that D is the set of feature model configurations in Algorithm 5 when the algorithm exits the loop. Considering that A holds for all feature model configurations, it should hold for all feature model con-

figurations in D. Therefore, A can be considered as a solution for the system of linear equations, which is built using D. This means that there exist at least two solutions for the system of linear equations built using D (all zeros and A). Considering that the algorithm has used D after the loop exited, it is in contradiction with the condition of the loop. $\square$

Lemma 2 *(No Loss of Information)* Given a set of features, F, for a feature model fm and a set of features, F', which is the result of Algorithm 5 on feature model fm, for function $Q(C) = \sum_{i \in \{1,..,n\}} p_i c_i$, there exists an assignment to $P' =< p'_1, ..., p'_{n'} >$ such that $Q(C) = Q'(C')$ where C is a valid feature model configuration and $C' = F' \cap C$.

Proof. It is proven that there exists a P' for the set F' corresponding to every P such that $Q(C) = Q'(C')$ for all valid configurations in all iterations of the while loop. Therefore, there exists such P' for the result set F'. This is proven by mathematical induction over the validity of the lemma in the k^{th} iteration of the loop in Algorithm 5. In the case of $k = 1$, $F = F'$ and the same values for P can be used for P'.

Now, it is proven that if for iteration k there exists a P_k such that $Q(C) = Q_k(C_k)$ for all valid configurations, there also exists a P_{k+1} for which $Q(C) = Q_{k+1}(C_{k+1})$ holds for all valid configurations. It is assumed that there are l features in F' in iteration k. The fact that the loop in Algorithm 5 did not exit at iteration k means that there exists an assignment $A =< a_1, ..., a_l >$ where $A \neq< 0, ..., 0 >$ and the equation $a_1 c_1 + ... + a_l c_l = 0$ holds for all

valid configurations. Assuming that in iteration k, the j^{th} feature has been removed to break a linear dependency, since the j^{th} feature has been removed, the value of a_j cannot be zero. Therefore, it can be said that:

$$c_j = \frac{\sum_{i \in \{1,..,l\}, i \neq j} a_i c_i}{a_j} \tag{4.7}$$

for all valid feature model configurations using equation $a_1 c_1 + ... + a_l c_l = 0$. Having an assignment to P_k such that $Q(C) = \sum_{i \in \{1,..,l\}} p_{ki} c_i$ based on the induction assumption, c_j can be replaced with its equivalent term in Equation 4.7. Consequently, the following holds:

$$Q(C) = \sum_{i \in \{1,..,l\}, i \neq j} (p_{ki} + \frac{a_i}{a_j}) c_i \tag{4.8}$$

The value for the right side of this is not affected by the value of c_j which no longer exists in F' in iteration $k+1$. Therefore, the coefficient values in Equation 4.8 can be used in P_{k+1} such that $Q(C) = Q_{k+1}(C_{k+1})$. □

It should be noted that Lemma 2 is defined over one non-functional property. However, the same proof can be used for more than one non-functional property considering that the process of finding linearly independent features only depends on the feature model constraints and does not depend on the non-functional property values.

The process of creating and updating this estimation function works as follows: m feature model configurations are randomly selected and their corresponding service compositions are built once. Having implemented service compositions for all those feature model configurations, the values for the non-functional property for all m feature model configurations are calculated using the individual services' non-functional property values. Using the set of feature model configurations and their corresponding non-functional property values, function $Q(C)$ is estimated using the ordinary least squares method. For example, Figure 4.6-B shows reliability values for the services in the upload image service family. Having these reliability values for the services, Figure 4.6-C shows an example dataset which can be used for estimating the contribution values for features using linear regression. This dataset is built by randomly selecting a feature model configuration and calculating the reliability of the corresponding service compositions by building the compositions and calculating its reliability using individual services' reliability values. Using this dataset, the contribution of each feature is calculated using ordinary least squares as shown in Figure 4.6-A.

Given this approach for calculating the non-functional properties of features as well as service compositions, it is possible to monitor compliance with non-functional requirements. In the next section, the proposed approach for adapting service compositions in reaction to the violation of functional and non-functional requirements is introduced.

4.4 Feature Adaptation Management

The goal of self-healing is to enable service compositions to self adapt in such a way that they fully recover or degrade gracefully instead of failing completely when functional and non-functional requirements are violated. Self-healing enables the service composition to continue its service provisioning with complete, limited or alternative features. Before discussion on how such self-adaptation is realized, it is specified what kind of failures are in focus in this thesis:

- **Functional Failures** , which happen when one or more services become unavailable. Such failures may result in not being able to provide one or more features or a specific combination of features.
- **Non-functional Failures** are violations of constraints, which are set for values of the service composition's non-functional properties as a result of changes in the non-functional properties of the services, for instance, when a non-functional requirement such as keeping the response time of the service composition less than $50ms$ is violated. Our method will be able to address those non-functional failures which are quantifiable and their values can be collected at runtime.

In order to be able to react to failure, the adaptation mechanism should have context-awareness where functional and non-functional failures trigger

adaptation. It is assumed that a special model called *context state model* exists, which contains functional availability and non-functional property monitoring information. A monitoring mechanism updates this model as changes happen in the context of the service composition. Certain changes in this model trigger the adaptation mechanism. An example of such a model can be seen in Figure 4.7, which relates to the service family in Figure 4.2.

As mentioned before, it is assumed that the user has specified those functional and non-functional properties which are *critical* for the delivery of the service composition. When a failure occurs such that it affects these critical requirements, the service composition should decide on the selection of features, which can restore the critical requirements. The decision making process should minimize the effect of such adaptation on the currently selected features of the composition. Such adaptation can be accomplished by finding an alternate feature model configuration. Finding an alternate feature model configuration ensures that the set of features in the alternate service composition represents a valid combination of features. The alternate feature model configuration should not rely on the failed services for its realization, and should rather satisfy the critical functional and non-functional requirements and should have the minimum difference in terms of features compared to the failed feature model configuration. This problem can be seen as an optimization problem with the presence of certain constraints.

User requirements outlined in the Initial State of Figure 4.7 shows an example of the input a user needs to specify for service composition. It specifies that Upload Image, Storage, and Tagging are critical and Metadata-based Tagging, Filtering, Profanity, Editing, and Face Blur are desired. It is also critical for the user that the composed service has a reliability higher than 0.9. Based on the availability and reliability measures of the services reported in the context state model of the initial state, the features shown in Figure 4.5 can be selected to offer a reliability value of 0.93, which satisfies the constraint specified by the user. In the following, a functional and non-functional failure is described and is used as a running example in the rest of this section. An example of functional failure is when the Object Detection service in Figure 4.5 becomes unavailable. The context state model and the satisfaction of user requirements in this situation has been shown as a failed state in Figure 4.7. In this state none of the user's requirements is being satisfied. An example of non-functional failure would be when the reliability of Object Detection service decreases from 0.991 to 0.941 as shown in the Failed state of Figure 4.9. Such a reduction in reliability will result in reduction of the overall reliability of the service composition to 0.883 which is less that 0.9 that is critical for the user.

It is necessary to quantify the difference between the new feature model configuration and the existing feature model configuration as a distance measure. This distance measure can be represented as a function over the presence or absence of features from the two feature model configurations. In

case of functional or non-functional violations, our objective would be to find an alternate feature model configuration that would satisfy the critical requirements and have the minimum value for this distance measure. In order to find such an alternate feature model configuration, this work defines a *pseudo-boolean optimization (PBO)* problem [26], which can subsequently be reduced to a boolean satisfiability (SAT) problem [48, 91] and efficiently solved using existing SAT solvers. Existing methods for finding an optimum for a pseudo-boolean function also allow for including constraints over the input variables, which the optimum should respect. In the following, a formal representation of what is meant by a pseudo-boolean optimization problem is presented and then it is shown how the problem of finding our desirable feature model configuration can be reduced to it.

Pseudo-boolean Optimization. Assuming an array of variables $X =< x_1, ..., x_n >$, $x_i \in \{0, 1\}$, a pseudo-boolean optimization problem can be formally defined as:

$$\begin{aligned} &\textbf{Minimize: } f(x_1, ..., x_n) \quad f : \mathbb{B}^n \rightarrow \mathbb{R} \\ &\textbf{Subject to: } R_1, ..., R_m \quad R_i : a_{i1}x_1 + ... + a_{in}x_n > a_i \end{aligned} \tag{4.9}$$

In this definition, f is a function whose domain is a set of variables $x_1, ..., x_n$ with values that are in the set $\mathbb{B} = \{0, 1\}$ and its range is $\mathbb{R}$ which is the set of real numbers. The possible minimum value for this function should be found with respect to a set of constraints $R_1, ..., R_m$ where each

constraint is an inequality relation defined over the input variables of function f. To be more specific, R_i is an inequality defined as the weighted sum of variables $x_1, ..., x_n$ where a_{ij} is the weight of x_i and the value of this weighted sum is greater than a threshold value a_i.

A feature model configuration C can be represented as an array of binary variables $\mathcal{C} =< c_1, ..., c_n >$ where c_i would be one if $f_i \in C$ and zero otherwise. Using this representation of the feature model configuration, the difference between two feature model configurations can be represented as a pseudo-boolean function. Before defining such a function, it is required to specify how the difference between the alternate feature model configuration and the current one can be quantified. One possible way to quantify this difference is using the loss in the utility for the user when an alternate feature model configuration is used. The utility is a numerical value used in the requirement engineering process to capture the relative usefulness of a product for a user [24]. Therefore, loss of utility as a result of feature model reconfiguration can be used as a measure that needs to be minimized. The problem of finding an alternate feature model configuration C' with the minimum loss of utility from C can be formally defined as a pseudo Boolean function as follows:

$$\textbf{Minimize: } loss(\mathcal{C}, \mathcal{C}') \tag{4.10}$$
$$\textbf{Subject to: } S, E, I, N$$

In this definition, *loss* is a function which takes two configurations as input and returns a numerical value representing the loss of utility when the feature model configuration represented with $\mathcal{C}$ is replaced with the feature model configuration represented with $\mathcal{C}'$. In this function, the parameter $\mathcal{C}$ is constant and the goal is to find an assignment to $\mathcal{C}'$ which minimizes the loss of utility. Furthermore, S, E, I, N represent different sets of constraints which an assignment to $\mathcal{C}'$ should satisfy. These constraints are as follows:

- Constraint S is a set of constraints representing the feature model's structural and integrity constraints,
- Constraint E is a set of constraints that make sure the new configuration makes valid assumptions about the input data of the service composition,
- Constraint I is a set of constraints which enforce the selection of those feature model configurations that do not require the inclusion of the failed services for their realization, and
- Constraint N enforces the selection of those combinations of features which satisfy critical non-functional constraints.

In the following, it is formally shown how the utility function and the set of constraints can be represented.

4.4.1 Loss of Utility

The underlying idea behind loss of utility is that in an ideal situation when neither functional nor non-functional requirement violations have occurred, the current service composition is the best possible service composition that satisfies the requested requirements. Now, if some of the services fail or underperform, a new feature model configuration needs to be created through which a new service composition would be generated. In order to calculate *loss of utility*, one would need to first calculate or estimate the utility of the underlying feature model configurations. Some existing methods [17, 13] assume the utility of each feature is independent of the other features and propose ways for eliciting a numerical value to represent the utility of each feature. Assuming that such values are available through a function $U : F \rightarrow \mathbb{R}$, the loss of utility incurred by replacing feature model configuration $\mathcal{C}$ with feature model configuration $\mathcal{C}'$ can be represented as a function $loss(\mathcal{C}, \mathcal{C}')$ where:

$$loss(\mathcal{C}, \mathcal{C}') = \sum_{i \in \{1,..,n\}} c_i(1 - c'_i)U(f_i) - (1 - c_i)c'_iU(f_i) \tag{4.11}$$

In this equation, i is the index for features assuming that there are n features where f_i maps to the ith feature. With this, $U(f_i)$ would be the utility of the ith feature. This equation iterates over all features in the feature

model and calculates the amount of utility lost by using $\mathcal{C}'$. For each feature, if the feature is selected or unselected in both feature model configurations, the value inside the sum operator would be zero. If the feature was selected in $\mathcal{C}$ but not available in $\mathcal{C}'$, the utility of the unselected feature is added to the sum of the lost utility. If a feature exists in $\mathcal{C}'$ while it does not exist in $\mathcal{C}$, its utility is deducted from the sum of the lost utility.

The drawback of this approach is that it is not applicable in all situations since the utility values of features are not always available. Therefore, a restricted version of the $loss(\mathcal{C}, \mathcal{C}')$ function for situations when utility values of features are not available is suggested. This function is referred to as $distance(\mathcal{C}, \mathcal{C}')$ that uses only the number of features which are different between the two feature model configurations. The *distance* function can be used instead of the *loss* function:

$$distance(\mathcal{C}, \mathcal{C}') = \sum_{i \in \{1,..,n\}} (1 - c_i)c'_i + c_i(1 - c'_i) \tag{4.12}$$

The expression in the sum operator of the optimization function evaluates to zero when c_i and c'_i are both one or zero which means both of those features are selected or unselected. In case one of c or c' is one and the other is zero, this expression evaluates to one. Therefore, this function will be zero when the two feature model configurations are identical.

4.4.2 Feature Model Structural and Integrity Constraints (Constraint S)

The set of features in the alternate feature model configuration should respect the constraints defined in the feature model. For example, the storage feature cannot be unselected in an alternate feature model configuration since it is a mandatory child of the root feature and a service composition is not valid without it. Therefore, the set of linear constraints S should be satisfied when an assignment to $\mathcal{C}'$ represents a valid feature model configuration in terms of structural and integrity constraints. All types of structural and integrity constraints can be represented using linear constraints over features. For example, the optional relationship between the Upload Image feature and the Editing feature can be represented as $c_{UploadImage} - c_{Editing} > -1$. There exist methods for translating a feature model structure and its constraints into a set of linear constraints. This work uses the translation proposed in [111].

4.4.3 Service Composition Precondition Constraints (Constraint E)

The new service composition which replaces the old service composition cannot make new assumptions about the preconditions of its execution and needs to have the same preconditions. Therefore, a feature model configuration that

is realized by a service composition which has new preconditions cannot replace the failed feature model configuration. For example, adding the watermark feature to the feature model configuration containing features marked with (•) in Figure 4.2 will require the condition *WatermarkRequested*(im,wt) to be true before execution, which might not necessarily be true. The linear constraint in the set E should ensure that the alternate feature model configuration does not change the preconditions of the service composition.

Considering that each feature f in the feature model is annotated with the set $\mathcal{P}(f)$ containing the preconditions that the selection of the feature will add to the preconditions of the service composition, the new configuration cannot include a feature whose $\mathcal{P}(f)$ is not a subset of the old feature model configuration's preconditions. Therefore, the linear constraints set E will have one member, which can be defined as:

$$\left(\sum_{i \in \{1..n\} \ s.t. \ \mathcal{P}(f_i) \not\subseteq Pre(C)} c_i' \right) < 1 \tag{4.13}$$

Assuming that $Pre(C)$ returns all of the preconditions required by the service composition realizing feature model configuration C as outlined in [15], the sum operator in this linear constraint makes sure that the alternate feature model configuration will not have those features that add preconditions not present in the precondition set of the initial service composition.

This condition will make sure that the alternate feature model configuration has the same or looser preconditions compared to the failed configuration. In the case of the example feature model configuration, this condition would be $c_{watermark} < 1$ considering that the precondition for the current configuration is an empty set along with the only feature whose precondition is not non-empty. This equation means that the watermark feature cannot be selected in the alternative feature model configuration considering that it requires a text to be watermarked on the image.

A user may want to remove a feature from the alternative feature model configuration. Removing a feature means making sure that the feature will never get selected in an alternative feature model configuration. In the case that the user has such requirements, similar constraints can be used. Assuming that set R contains the indices of the features which have been removed by the user, a constraint represented as $\sum_{i \in R} c'_i < 1$ will ensure those features will never be selected in the alternative feature model configuration.

4.4.4 Service Independence Constraints (Constraint I)

These constraints prevent the selection of those feature model configurations which rely on the failed services for their realization. Since there is no direct mapping between features and services in our method and multiple features can be realized by multiple services, it is not easy to find those feature model

configurations that cannot be realized after a service failure without trying to compose them.

In order to address this issue, an incremental approach is taken. A failed feature model configuration set is defined which consists of those feature model configurations that rely on the failed services for realization and make sure that members of this set are not selected as an alternate feature model configuration. In the beginning, this set is initialized with the feature model configuration whose corresponding service composition has failed. Consequently, as alternative feature model configurations are found, the adaptation mechanism tries to generate their corresponding service composition using existing services. If this approach is not able to generate a service composition for the feature model configuration, that feature model configuration is added to the failed feature model configuration set and the process of looking for an alternate feature model configuration continues. The set I is made of a set of linear constraints where each of those constraints correspond to a feature model configuration in the failed feature model configuration set. Assuming that C is a feature configuration in this set, its corresponding linear constraint is defined as:

$$\left(\sum_{i \in \{1..n\} \ s.t. \ f_i \in C} (1 - c'_i) + \sum_{i \in \{1..n\} \ s.t. \ f_i \notin C} c'_i \right) > 1 \tag{4.14}$$

This constraint would not be satisfied if the feature model configuration corresponding to the current assignment is the same as the feature model

configuration C and true otherwise. For the example configuration, the constraint set I after a functional or non-functional failure - such as the ones mentioned above - initially includes only the following constraint:

$$8 - c'_{uploadImage} - c'_{tagging} - c'_{metadataBased} - c'_{filtering} - c'_{profanity} - c'_{storage} - c'_{editing} - c'_{faceBlur} + c'_{external} + c'_{nudity} + c'_{watermark} > 0 \quad (4.15)$$

The number 8 comes from simplifying the Equation 4.14 for the example configuration. Considering $c' \in \{0, 1\}$ and only 8 features are being subtracted from 8, this equation can never be lower than zero and the only case it can be zero is when all features corresponding to the current configuration are selected which will cause all subtracted variables to neutralize the effect of 8 and none of the other features be selected considering that they increase the value of the left side of this inequation. This selection of features corresponds to the current configuration. Therefore, this constraint prevents the currently failed configuration to be selected again. As the method looks for an alternative configuration for the current failure, similar constraints corresponding to those configurations which cannot be realized using existing services are added to I.

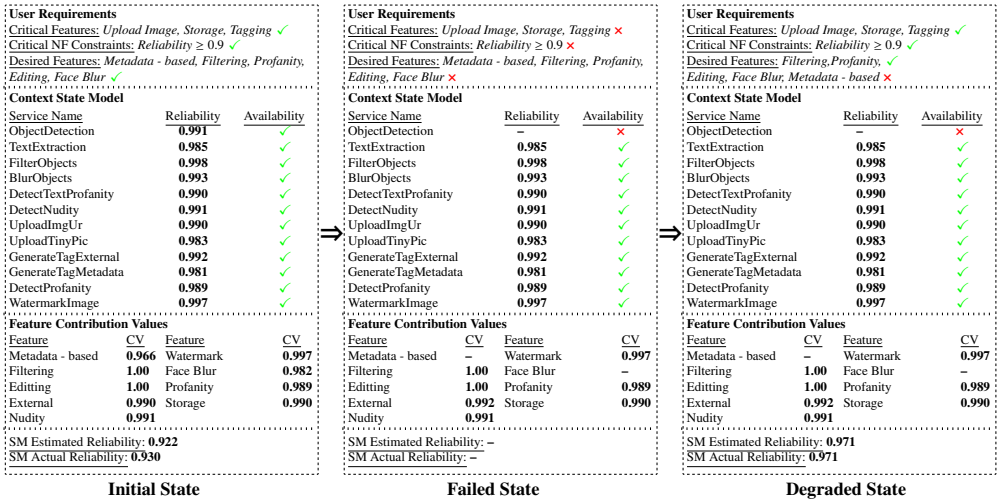


Figure 4.7: Initial, failed, and degraded state of the service composition in Figure 4.5 after failure of the *Object Detection* service.

4.4.5 Non-functional Constraints (Constraint N)

The goal of these constraints is to make sure that the service composition, which is built based on the alternate feature model configuration, satisfies the non-functional constraint set for the service composition. It has been already discussed that the non-functional properties of a service composition can be estimated using a linear function over the features using Equation 4.1. Therefore, the constraint for a non-functional property can be represented as:

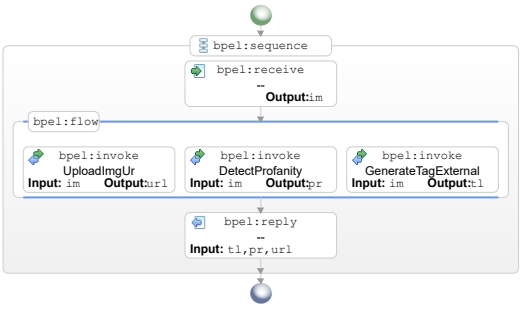


Figure 4.8: BPEL process visualization for an alternate solution for BPEL process in Figure 4.5 after adaptation by feature model reconfiguration as a result of the *ObjectDetection* failure.

$$\sum_{i \in \{1,..,n\}} p_i c_i' < q \tag{4.16}$$

where p_i is the contribution value of the ith feature and q is the threshold for that non-functional property. This linear constraint ensures that the value for the non-functional property of the alternate feature model configuration is less than the threshold value when the user has specified that the non-functional property value must be lower than the threshold. If the user has specified that the value must be greater than a threshold, the lower than operator should be switched to the greater than operator.

In the previous section, a regression method for estimating the contribution value of each feature to a non-functional property is proposed. This method is used in order to estimate all p_i values. The Initial state in Figure

4.9 shows the calculated contribution value for each feature for the example case study as well as the updated contribution value of the features after the example non-functional failure as a result of decline in the reliability of the object detection service. It can be seen that the contribution value of metadata-based tagging and face blur show decline, which is intuitive considering that it can be seen in Figure 4.5 that this service is used in realizing the functionality of both of those features.

Considering that the product operator is an appropriate aggregation operator for reliability, the constraint in Equation 4.16 can be defined over the natural logarithm reliability values as shown below:

$$ln(0.957)c'_{metadataBased} + ln(1)c'_{filtering} + ln(1)c'_{editing} + ln(0.992)c'_{external} + ln(0.991)c'_{nudity} + \\ + ln(0.997)c'_{watermark} ln(0.967)c'_{faceBlur} + ln(0.989)c'_{profanity} + ln(0.990)c'_{storage} > ln(0.9) \tag{4.17}$$

This inequation is linearly defined over the presence of features and will act as a constraint which assures that the alternative feature model configuration will satisfy the reliability constraint.

It should be noted that these values are estimated rather than being exactly specified; there is no guarantee that the actual non-functional property of the service composition is the same as the estimated value. In order to address this issue, a probabilistic approach is used. Assuming e is the error of estimation, it can be assumed that it has a normal distribution $\mathcal{N}(0, \sigma)$ with a mean of 0 and a standard deviation of σ. The value of σ can be calculated using the feature model configuration in the dataset and their corresponding

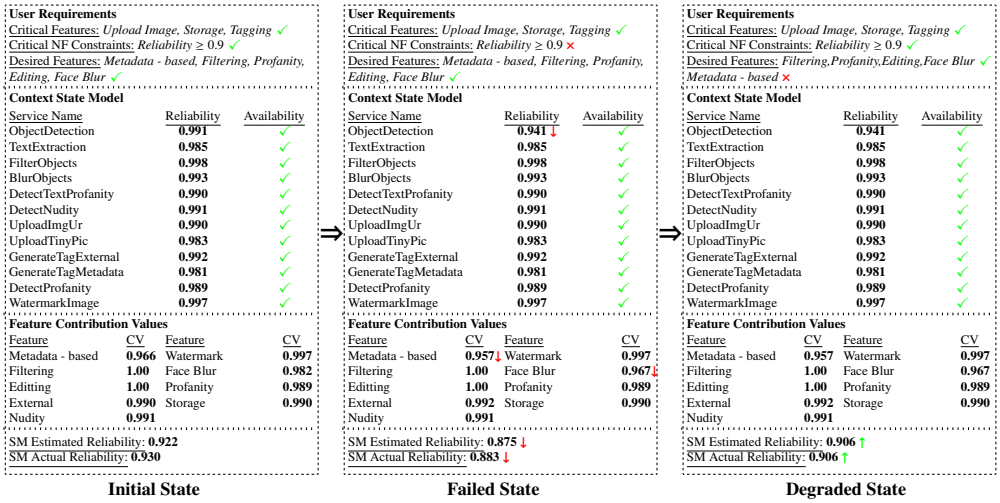


Figure 4.9: Initial, failed, and degraded state of a service composition in Figure 4.5 after change in the reliability of the *Object Detection* service.

compositions. Having the distribution of error e, the value for threshold q can be set in such a way that there is a specific confidence, e.g., 95%, that the generated service composition satisfies.

In summary, based on the formal definitions provided for the loss of utility and the four constraints required in Equation 10, it is possible to automatically derive a new feature model configuration that alleviates the functional and non-functional requirement violations as well as respecting the four types of constraints. The adaptation happens by optimizing the minimization problem of Equation 10 in the context of Equations 12, 13, 14 and 15. The outcome of this optimization problem is an alternate feature

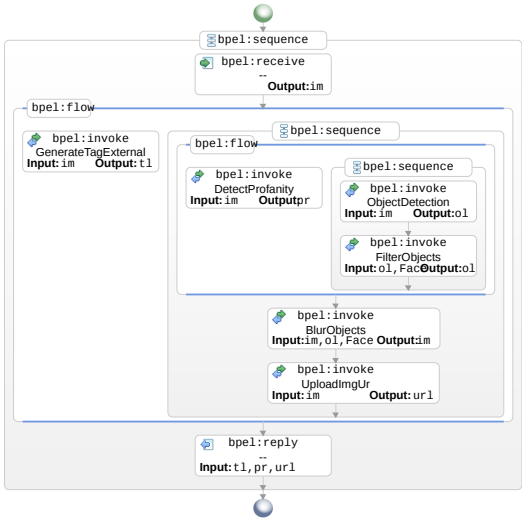


Figure 4.10: BPEL process visualization for an alternate solution for BPEL process in Figure 4.5 after adaptation by feature model reconfiguration as a result of the decrease in the reliability of the *ObjectDetection* service.

model configuration that would be used for generating an alternate service composition.

For the case of the sample functional failure, the above method results in a feature model configuration which does not provide metadata-based tagging, editing and face blur features but provides external tagging. The new feature model configuration is marked with (○) in Figure 4.2. The alternative feature model configuration represents a degraded state where service composition still provides upload image, storage, and tagging which are critical for the user but does not provide some non-critical features. Figure 4.8 shows the alternate service composition built based on this configuration. This BPEL process uses the same service for realizing storage but uses alter-

native services for tagging and detecting profanity considering that previous logic for realizing them is no longer possible after the failure of the object detection service.

For the case of the sample non-functional failure, the adaptation mechanism results in a feature model configuration which does not provide metadata-based tagging and uses external tagging instead. This feature model has been represented in Figure 4.2 with features marked with (■). Again, this configuration provides the critical features and the corresponding service composition has a reliability of 0.906 which is higher than the user specified threshold. Figure 4.10 shows the alternative service composition built based on the alternative feature model configuration. This BPEL process still uses degraded object detection service considering that this service is required for realizing three of the desired features (i.e., tagging, editing, and face blur) but uses different logic for realizing tagging and detecting profanity features which results in higher reliability than the logic in Figure 4.5.

4.5 Summary

resThis chapter proposes the idea that it is possible for a service composition to recover from both functional and non-functional failures by losing some of its non-critical features automatically without external intervention. In order to investigate the feasibility of this idea, a self-healing service composition

reconfiguration method was proposed. In the proposed method, the features of a family of service composition are represented through software product line feature models, which allow for defining constraints on features, which guarantee that a set of features that satisfy those constraints represent a valid service composition. This allows the service composition to undergo adaptation by selecting another subset of features which are valid with respect to those constraints. The proposed method is composed of two sub-processes where the first sub-process calculates the contribution of each feature on the non-functional properties of the service composition and feeds it to the second sub-process which realizes the adaptation by deciding which features should be included in the adapted service composition. The first process works by performing linear regression and the second process works by reducing the problem of selecting features to a pseudo-boolean optimization problem.

4.6 Related Publications

- Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Self-healing in Service Mashups Through Feature Adaptation." *International Systems and Software Product Line Conference*, ACM, 2017 (Refereed Paper)

- Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Self-Adaptation of Service Compositions through Product Line Reconfiguration", *Jour-*

nal of Systems and Software 144 (2018) 84-105

Chapter 5

Tooling Support

The proposed automated composition method, the proposed self-healing method as well as tools for creating domain models have been implemented in an online tool suite called *MAGUS* which is available online[1] and supports the whole lifecycle of service composition from design to managing its adaptation. Our tool provides the following benefits:

- The tool supports domain experts in the process of designing the domain models for a service composition family by providing user interfaces for designing different models such as feature models and context models.

[1] Available at: `http://magus.online/`

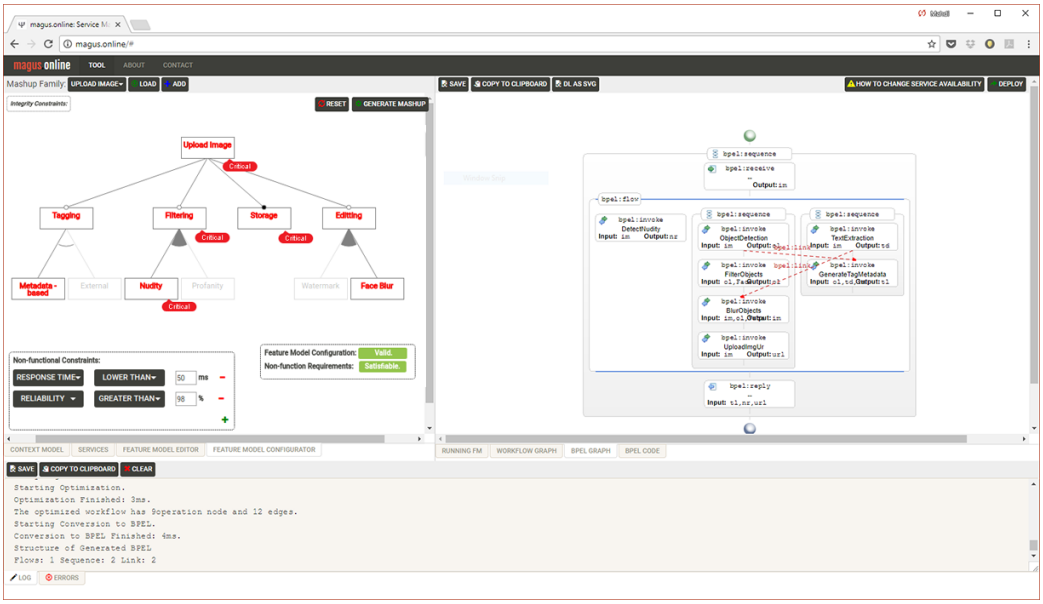


Figure 5.1: A screenshot of the Magus.Online tool suite.

- The tool enables users to generate their desired service composition by providing an interface for specifying requirements and generating the service composition automatically based on the specified requirements.

- The tool supports users by enable self-adaptation on a service composition by providing an automated adaptation management engine which recommends an alternate service composition when adaptation is required.

A screenshot of this tool suite has been shown in Figure 5.1. This tool suite provides three tools which focus on different phases of the service composition lifecycle: *Domain Design Tool*, *Service Composition Configuration Tool*, and

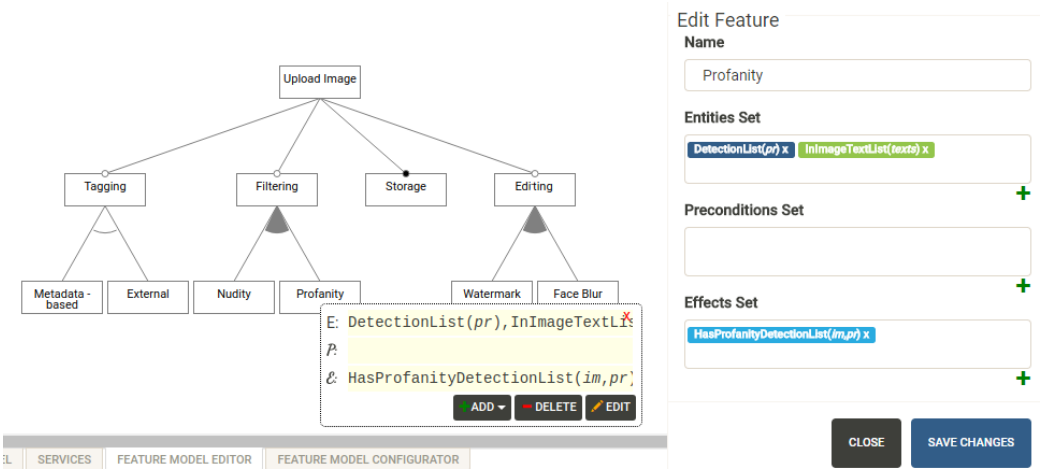


Figure 5.2: The Domain Design Tool interface for annotating features in the feature model.

Adaptation Management Engine & API. Details on the functionality of each of these tools and how they are used in the service composition lifecycle are provided in the following.

5.1 Domain Design Tool

This tool allows a software developer to design domain models for a service composition family. These domain models are base models for automated service composition and adaptation. Specifically, this tool provides the following functionalities:

- It allows for creating and editing a context model and its entities and fact types and saving and loading it as an OWL ontology.

- It allows importing and modifying OWL-S definitions of existing services into the domain model.
- It provides a graphical user interface for creating, editing, and serializing feature models as well as annotating features using the context model.

In order create a service composition family, the software developer extracts possible variability in the problem domain and designs a feature model capturing those using the feature model designer in the tool. The developer then uses the context model design feature to design the context model of the service composition, which reflects the properties of interest within the context where the service composition operates. This context model is used to annotate the features in the feature model in order to specify what is the expected behavior of the service composition that has that feature. Figure 5.2 shows a screenshot of the Domain Design Tool GUI used for editing and annotating a feature model. Having the annotated feature model in the tool, the user can provide services that realize features of the feature model by importing existing OWL-S definitions or by developing new services and annotating them using the service annotation GUI provided in the Domain Design Tool.

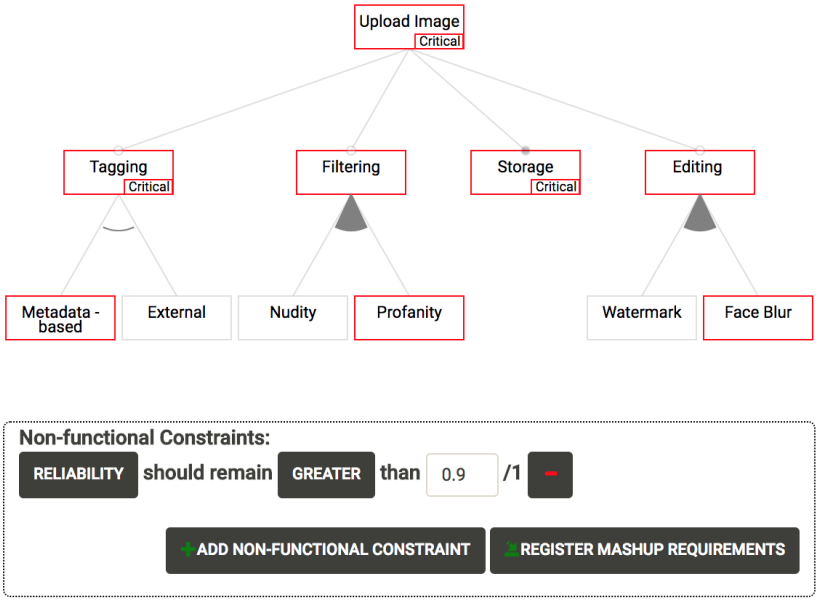


Figure 5.3: The Service Composition Configuration Tool interface for specifying user's requirements.

5.2 Service Composition Configuration Tool

A non-expert user can load an existing domain model corresponding to a service composition family in the Service Composition Configuration Tool and specify the requirements of the desired service composition using this tool. The requirement specification is used for composing desired service compositions as well deciding on the adaptation when a failure takes place. Specifically, the Service Composition Configuration Tool provides the following functionalities:

- It allows loading an existing service composition family using the URL to its configuration.
- It provides a user interface for interactively configuring a feature model.
- It verifies the satisfaction of feature model structural and integrity constraints as the user configures the feature model.
- It allows defining constraints over the values of the different non-functional properties.
- It can create a service composition satisfying the specified requirements and generate its associated BPEL code as well as representing it visually.

In order to define the requirements of a service composition, the user first loads the service composition family. When a service composition family is loaded in the Service Composition Configuration Tool, the feature model for that service composition family is viewed where the user can mark a feature as selected or unselected. The user can additionally specify if that feature is critical for the service composition or not. The Service Composition Configuration Tool checks the structural and integrity constraints of the feature model and notifies the user if the current selection of features violates any of them. The user can also define non-functional constraints by specifying the type of the non-functional property, the threshold value and if the non-functional property should be lower or higher than this value.

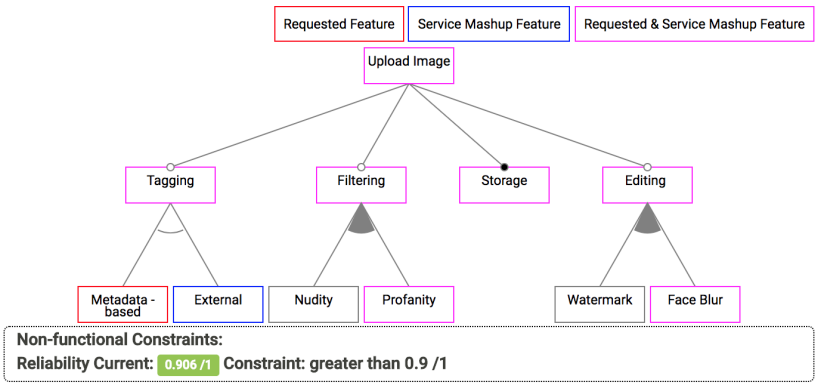


Figure 5.4: The Adaptation Management Engine interface for monitoring the current state of the service composition.

The Service Composition Configuration Tool investigates the possibility of satisfying those constraints and notifies the user if those constraints can be satisfied using current services. Figure 5.3 shows the interface for the composition configuration tool when the user has selected a set of features and specified two non-functional constraints. The user can request for the service composition to be generated, which would produce the corresponding BPEL code for the composition.

5.3 Adaptation Management Engine and API

Rather than implementing this tool as an extension of an existing BPEL execution engine which limits the application of the proposed method to that specific engine, an API is developed which can work with an execution engine. This API simply sends the current state of the system to the Adaptation

Management Engine where it detects failure and recommends replacement BPEL processes when necessary. Specifically, the Adaptation Management Engine & API provides the following functionalities:

- It allows integration with different BPEL execution environments through a well-defined REST API.
- It provides a visual representation of the current health and performance of the service composition.
- It enables feature adaptation in response to service failure or non-functional violation using the REST API.
- It provides a notification service which notifies the user of failures and adaptation strategy taken to address them.

After a service composition is generated, it can be deployed on a BPEL execution environment. An execution environment can be extended to be managed using the Adaptation Management Engine through its API. This API is well-documented and is available online[2]. Using an API makes adaptation management agnostic to the execution environment and therefore can be used for different BPEL execution environments. The execution environment sends the current context state model to the adaptation management engine through the API where satisfaction of user requirements is examined. In the case of failure, an adaptation takes place and the updated service composition

[2]Swagger documentation available at: `http://magus.online/swagger-api`

is sent back to the execution environment. The Adaptation Management Engine also updates the online console with updated information of the service composition. Figure 5.4 shows the console for a running service composition. It shows the user's desired features, those features that have been provided, and those features which cannot be provided in the current state as well as the health of the service composition in terms of non-functional properties.

5.4 Summary

In this chapter, a tool suite which supports practical use of the proposed work is introduced. This tool suite is made of three tools. The first tool is used by an engineer and allows developing families of service composition. The second tool can be used by end users to configure their desired service composition family based on their requirements. The third tool provides a monitoring capability, as well as an API which BPEL execution engines can use for managing self-adaptation when needed. All these tools are available online and readily usable.

Chapter 6

Experiments and Evaluation

Our proposed approach has been implemented using the FF planner [69] for AI planning and NaPS solver [127] for pseudo-boolean optimization. Using this implementation, seven experiments were performed in order to investigate the practicality of the proposed method. The experiments were performed on generated services and feature models considering that actual feature/service repositories that would have different sizes are not publicly available. These experiments were performed on a machine with Intel Core i5 2.5 GHz CPU, 6GB of RAM, Ubuntu 16.04 and Java Runtime Environment v1.8. In the following two sections, experiments performed for evaluation service composition and for the adaptation approach are discussed respectively.

6.1 Evaluating the Composition Approach

The service composition approach can be divided into three main parts, which are: workflow generation, workflow optimization, and BPEL code generation. In the following, these three parts are reviewed and the experiments which have been performed in order to evaluate them are described.

6.1.1 Workflow Generation

The soundness and correctness of the proposed techniques have been already shown formally; therefore, the main focus of the experiments with regards to workflow generation is the assessment of the scalability of the proposed method in terms of its running time. The efficiency of the method is evaluated from two perspectives:

- **Experiment 1.1 (Scalability in terms of the service repository size):** How does the workflow generation time increase as the number of services in the repository grows?
- **Experiment 1.2 (Scalability in terms of the feature model configuration size):** How does the workflow generation time increase as as the size of the feature model configuration grows?

In order to run the experiments, three models were required: context model, services and their annotations, feature model and its annotations.

Context Model: An ontology was developed for the context model with 30 entity types and 600 fact types. We could not find an existing work in the literature to report on the number entity type and fact types in practical domains. Therefore, these numbers were chosen arbitrarily. This context model is used to annotate the services and the feature model.

Services and their annotations: In order to generate the services and their annotations, a random OWL-S service description generator was developed which creates service descriptions with inputs, outputs, preconditions, and effects randomly picked from our context model. This OWL-S service description generator is highly customizable with different service model characteristics (e.g., number of inputs, outputs, precondition, and effects). Three service repository sets have been created where services in the repositories of different sets have different numbers of preconditions and effects. Since we could not find a work in the literature that reports the typical number of annotations per service, we used the average number of annotations for the services in our case studies which was 3. We also generated services with 6 and 9 annotations to capture how the increase of annotations affects the execution time of this method. Therefore, the numbers 3, 6, and 9 are used as the number of preconditions and effects. Each of these repository sets has 10 different repositories of sizes between 1,000 to 10,000. In total, 30 different service repositories have been created.

Feature model and its annotations: The SPLOT feature model generator [99] is used to generate a feature model with 1,000 features which represents a large feature model in industry. In order to annotate this feature model, a customized feature model annotation generator was developed which randomly picks annotations from the context model and assigns them to the features of the feature model. Using this annotation generator, three different annotation sets were created for the feature model. Considering that feature model annotation is a new model proposed in this thesis, we generated an annotation set with 2 annotations per feature. This number is the average number of feature annotations in our case studies. Furthermore, we also generated annotation sets with 8 and 16 per feature to investigate the effect of increase in the number of annotations on the execution time the method. In the first experiment, a feature model configuration with 50 features is selected and the time to generate the workflow using service repositories of different sizes is measured. This operation is done repeatedly 20 times with different feature model configurations of the same size and the average time for generating the workflow is calculated. This experiment is repeated for all three repository sets. Figure 6.1 shows how the workflow generation time increases with the increase in the size of the service repository. As it can be seen from the figure, the workflow generation time remains manageable (around 2.4 seconds for 9,000 services).

In the second experiment, the service repository with 1,000 services and an average sum of preconditions and effects of 6 is selected. In this setting,

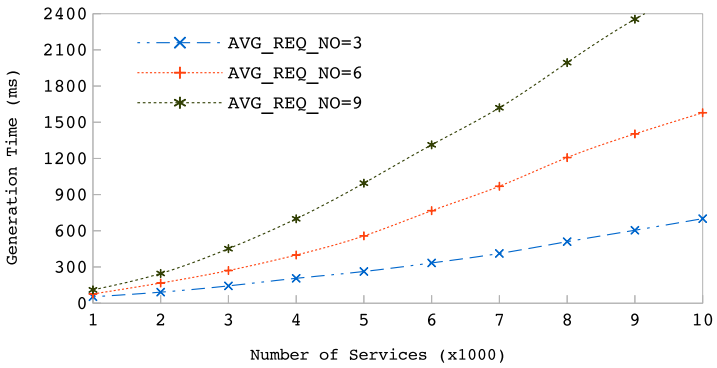


Figure 6.1: Workflow generation time in terms of service repository size.

the time for generating workflows for feature model configurations of different sizes is measured. The feature model configuration is generated by randomly selecting features using a tool based on the FaMa suite [19] which receives a feature model and the desired number of features in the configuration and returns a random valid feature model configuration with that size. For each configuration size, 20 different configurations are generated. For each number of annotations, the average time required for generating the workflow is calculated for different configurations. Figure 6.2 shows the average workflow generation time with different feature model configuration sizes for different number of annotations. As seen in the figure, the generation time remains manageable when the number of annotations are 2 and 8 per feature. However, when the number of annotations is increased to 16, the generation time becomes exponential and shows rapid increase. It is important to note that i) even with the increase, the time is manageable for practical purposes, i.e.,

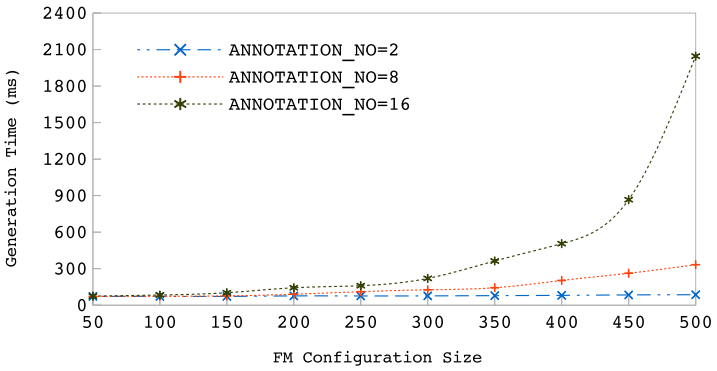


Figure 6.2: Workflow generation time in terms of feature model configuration size.

2 seconds for 1,000 services and 500 requirements. ii) Literature suggests that the number of annotations is typically in the range of 5-6 annotations per feature [12], in which case, the performance of the generation algorithm is linear.

6.1.2 Workflow Optimization

The focus of the second set of experiments is on the investigation of the scalability of the optimization method. The optimization method scalability is explored when the size of workflow increases. In addition, it is explored whether the optimization method is able to decrease the time-to-completion of the workflow.

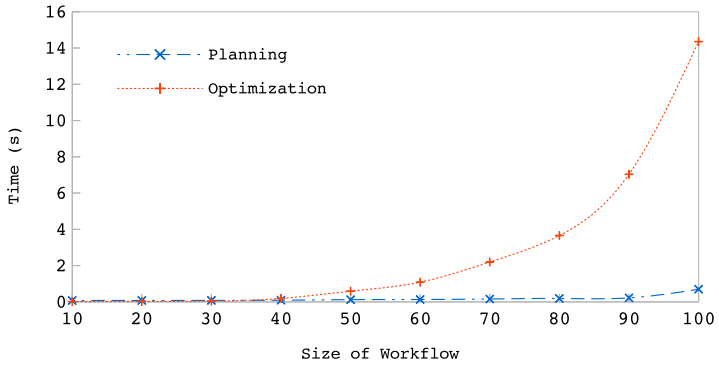


Figure 6.3: Workflow optimization time in terms of workflow size.

- **Experiment 2.1 (Optimization scalability in terms of workflow size):** How does the workflow optimization time increase with the increase in the size of workflow (in terms of growth in the number of workflow nodes)?

- **Experiment 2.2 (Effectiveness of the optimization in terms of workflow time-to-completion):** How much does the time-to-completion of a workflow decrease as a result of the optimization?

For the sake of the experiments, the models from the previous experiment were reused. The service repository with 1,000 services and an average number of preconditions and effects of 6 were employed. The services were annotated with random time-to-completion with a normal distribution of $\mathcal{N}(200ms, 50)$.

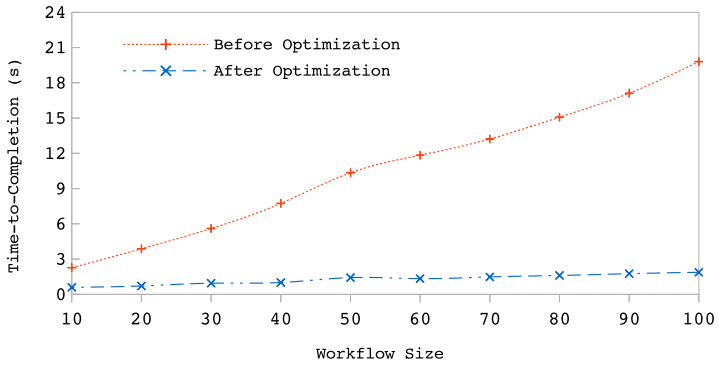


Figure 6.4: Time to completion in terms of workflow size.

For the first experiment, 20 different configurations in each workflow size category were randomly selected and the average time for workflow generation and optimization is calculated. Figure 6.3 shows how workflow generation and optimization time increases as the size of workflow grows. This shows that the workflow optimization method is considerably slower than the planning method. However, given the fact that the optimization method is only a one-time task, its benefits in terms of reducing the time-to-completion is noticeable.

In the second experiment, the objective was to measure whether the optimization method is able to generate workflows that have a lower time-to-completion compared to the original non-optimized workflows. For this purpose, the times-to-completion of the generated workflow were calculated both before and after the optimization. Figure 6.3 shows the result of the optimization. As seen in the figure, the time-to-completion of a non-optimized

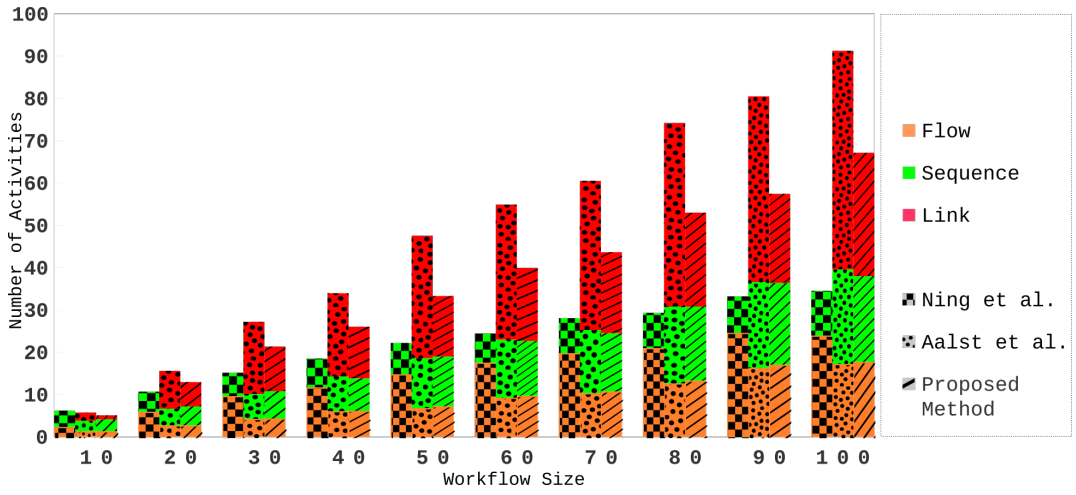


Figure 6.5: Comparison between different BPEL generation methods in terms of element distribution.

workflow increases as the size of the workflow increases. However, the optimization method has been able to maximize parallelism in the workflow such that there is very little growth with the increase in the workflow size. For instance, for a workflow with 100 activities, which on average take 20 seconds prior to optimization, the optimization method has been able to reduce the time-to-completion to 1 second.

6.1.3 BPEL Process Generation

The focus in the experiments in the BPEL generation part is on comparing the proposed method with two other BPEL generation methods in terms of structuredness of the resulting workflow. In addition, the time required for

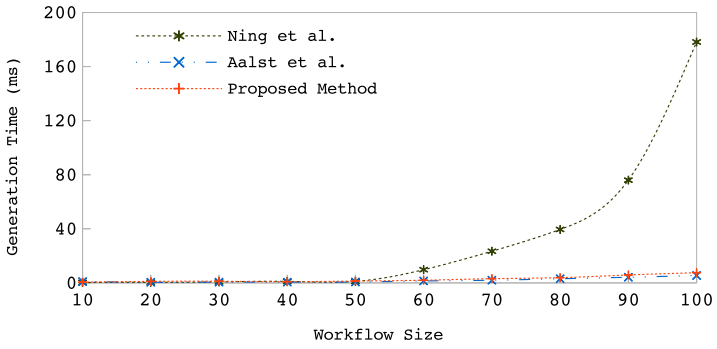


Figure 6.6: Comparison between different BPEL generation methods in terms of generation time.

the generation of BPEL in the proposed method is compared with two other methods.

- **Experiment 3.1 (Process structuredness in terms of the number of block-structure activities):** How well does the proposed method take advantage of block-structured activities in the generated BPEL code compared to other methods of BPEL generation?

- **Experiment 3.2 (BPEL process generation scalability in terms of process size):** How does the BPEL process generation time increase with the increase in the size of input workflow (in terms of growth in the number of workflow nodes)?

In order to run these experiments, the workflow models from previous experiments were re-used. In order to compare our proposed method with other existing methods, the method proposed in [110] and a method based on the

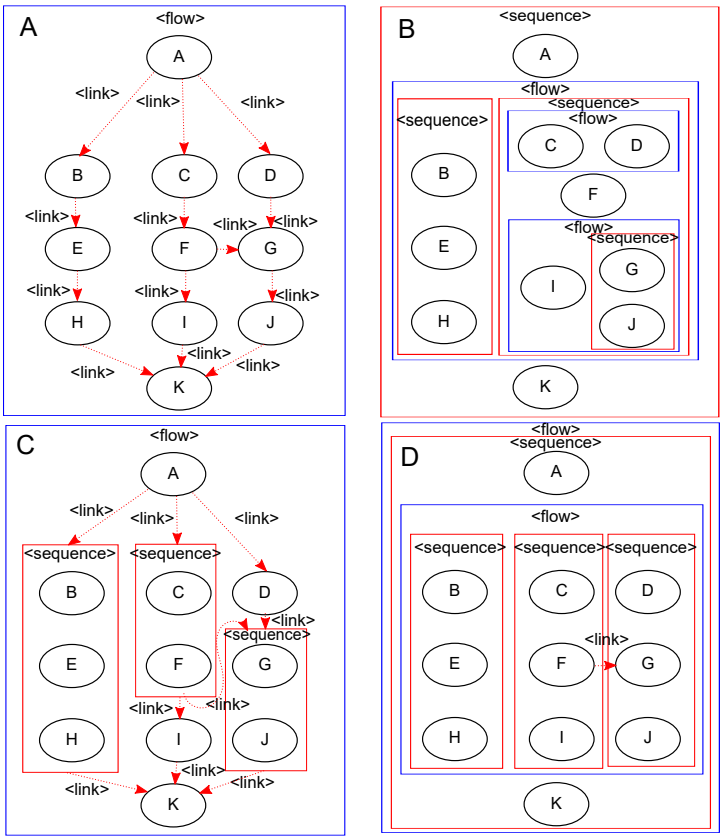


Figure 6.7: Possible BPEL representation of workflow in Figure 3.4.

idea in [149], were implemented. In [110], an algorithm, known as BPEL-GEN, for converting a service workflow represented by a directed acyclic graph to BPEL is proposed. The resulting process is a fully block-structured process built only using flow and sequence activities. However, this method adds some execution constraints since a graph-based structure cannot be fully captured using block-based structure. For example in Figure 3.4, the workflow graph generated by our approach for the example feature model

configuration shown in Figure 3.2 is presented. Figure 6.7-B shows the resulting BPEL process generated by executing the BPELGEN algorithm. Although this BPEL process is fully block-structured, it imposes some new constraints such as enforcing the execution of operation I after D or F after D which is not enforced in the workflow graph. In [149], a method for converting workflow-net, which is a petri-net based language, to a BPEL process has been proposed. In this method, workflow-net is converted into a BPEL process by consecutive folding of structures of workflow-net into BPEL activities. However, this method requires human intervention when it could not find any well-structured set of activities for folding as discussed before. Figure 6.7-C shows the result of this method for the example in Figure 3.4 before requiring human intervention. Figure 6.7-D shows the resulting BPEL process generated by our proposed approach.

In order to perform the two experiments of this section, 50 different workflows are randomly selected and converted to BPEL using the three different methods. For the first experiment, the number of flow activities, sequence activities and link tags are collected and for the second experiment, the time to generate that BPEL is collected. For the method proposed in [149], the BPEL code right before requiring human intervention was considered to be the result.

Figure 6.5 shows the average number of flows, sequence activities and link tags for BPEL code generated by the different methods. It can be seen

that the number of links and total number of elements are highest for the method proposed in [149], which is expected considering the fact that this method cannot proceed when there is no well-structured set of activities in the BPEL code. Our proposed method creates processes with a significantly smaller number of activities and smaller number of links compared to this method. Additionally, the BPELGEN method creates no links and fewer activities compared to our proposed method. This is because BPELGEN method merges sequence tags and flow tags together whose are not merged in the proposed method to maintain the precedence relationship defined in the input graph. Furthermore, BPELGEN method imposes dependencies which are not in the workflow graph which can negatively affect understandability of the workflow as well as some quality of service attributes such as time-to-completion.

Figure 6.6 shows the average time required for generating BPEL processes using the different algorithms. Our proposed method and the method in [149] are able to generate the BPEL process in a very fast time (<10ms). The BPELGEN algorithm shows exponential increase in BPEL generation time, which is expected considering that it requires depth-first traversal over the workflow graph in each iteration. Still, the time required for generating BPEL using this method remains acceptable (<200 ms) for the process sizes tested in our experiments.

6.1.4 Threats to Validity

In this subsection, we discuss the validity of the observations that were made in the experiments by discussing issues related to internal validity and also discuss threats that can affect the generalization of the observations by analyzing external validity.

6.1.4.1 Internal Validity

In the following, we highlight some of the threats related to the internal validity of our experiments:

- All the models which are used in these experiments are synthetically generated. While the models are synthetic, we have been rigorous in capturing and reporting all of the statistical distributions that were used in our tools for generating these models. Despite this, there can still be some unconfigurable parameters in these tools which could result in bias when creating the synthetic models, hence indirectly impacting the experimental results.

- In Experiment 2.1, planning was performed using the FF planner, which is a highly optimized planner written in C; however, for the sake of consistency, we have implemented the optimization logic using the Java programming language. The programming language as well

as the optimization method can affect time-to-completion. Therefore, the reported values could change based on a different implementations.

- The implementation for the two methods used as baseline for BPEL process generation has not been publicly released by the authors. Therefore, we have implemented these two methods based on the algorithms provided in their corresponding papers. These implementations may have minor differences with the actual implementation of the authors, which may result in slightly different outputs. We have made the implementation of the baselines as well as the implementation of our proposed method available on the magus.online source repository for replication purposes.

6.1.4.2 External Validity

In the following, we report on the issues which threaten the generalization of the observations made in the experiments.

- The results of our experiments may be different when applied on real-world domains considering the fact that we used synthetic domain models in our experiments, which may not truly capture the properties of a real problem domain. Therefore, evaluating the proposed method by addressing real-world problems has to be considered as a next step for our work.

- Experiment 2.2 assumes that the time-to-completion of all services has the same distribution. However, this may not be the case in real-world scenarios, which may affect the efficiency of the optimization method. Similar to the previous issue, evaluating the optimization method on a workflow defined over actual services may be able to better capture the efficiency of this method. Furthermore, the optimization method is only evaluated for optimizing time-to-completion that may not have the same efficiency for other properties of the workflow such as parallelism.

- In Experiment 3.1, it has been shown that the proposed method results in more structured BPEL processes compared to other methods. Considering that structuredness is used as a measure for understandability of a process, we concluded that our method results in more understandable processes. This cannot always be the case considering that the process can be structured in a way which is confusing for the user. This conclusion can be further examined through a user-study, which evaluates the understandability of the generated processes by actual users.

6.2 Evaluating the Adaptation Approach

In this section, four different experiments are performed to evaluate the practicality of the proposed approach. In the first two experiments, the effec-

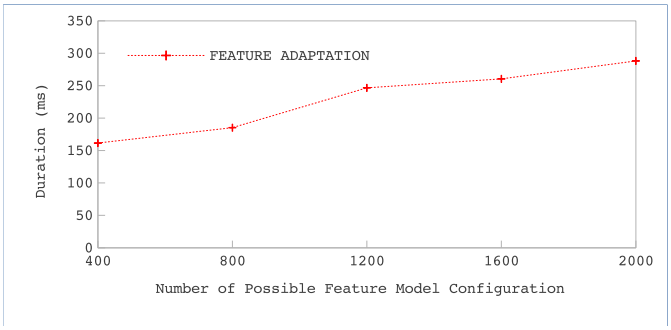


Figure 6.8: Feature adaptation execution time in terms of feature model configuration sizes.

tiveness and robustness of the proposed approach for functional failure are evaluated. In the third experiment, the effectiveness of the proposed regression method estimating non-functional properties of a service composition based on its features is evaluated. Finally, the effectiveness of the proposed adaptation approach in response to non-functional requirements violations is evaluated.

6.2.1 Effectiveness of Adaptation in Functional Failures

In this first set of experiments, answering two questions is in focus:

Experiment 4.1 - How does the execution time for performing an adaptation change as the possible feature model configuration size increases?

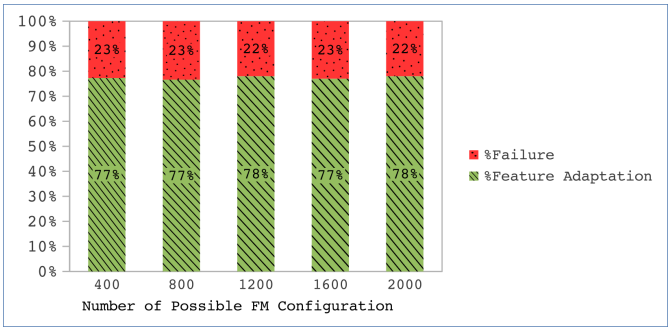


Figure 6.9: Distribution of success versus failure of the adaptation mechanism.

Experiment 4.2 - What is the success rate of the adaptation mechanism when addressing service failure for different feature model configuration sizes?

The goal of exploring the first research question is to evaluate if the adaptation process is performed in a reasonable amount of time. Furthermore, through this research question, the applicability of the proposed method in terms of adaptation time for larger feature models is investigated. The objective of the second research question is to first investigate if the proposed self-healing approach is capable of recovering from service failures and second to examine if the adaptation method remains feasible as the size of the feature model increases. The number of valid feature model configurations is used rather than the number of features since it is a more suitable metric for representing the size of the search space for the optimization problem and the method searches in this space for the solution.

The experiments were designed as follows: For different service families with the same service repository size but different feature model sizes, a feature model configuration were randomly selected, and the service composition satisfying it was automatically composed. Then a service was randomly selected and placed in the failed state and the time to perform adaptation and if the adaptation was successful were recorded.

The experiment was performed for five different groups of service families where feature models in the same group had the same number of features but different numbers of possible configurations. Each group had 10 different service families which were generated as follows: The Betty feature model generator [130] was used to generate the feature models with 30 features, which is the average number of features in the SPLOT repository[1] and with maximum branching factor of 10. Four groups consisting of 25 percent of the features each were allocated to the Or group, Alternative group, Optional and Mandatory categories. According to the survey that is performed in [146], these parameters reflect structural properties of a real feature model. The number of possible configurations for the generated feature models ranged between 400 and 2,000, which was divided into 5 groups. The feature models were annotated using the context model with 30 entity types and 600 fact types using a feature model annotation generator with parameters $\mathcal{N}(2,1)$, $\mathcal{N}(0.2,0.8)$, and $\mathcal{N}(1,1)$ as the number of entities, preconditions, and effects, respectively. $\mathcal{N}(\mu,\sigma)$ is a normal distribution with a

[1] http://www.splot-research.org/

mean of μ and a standard deviation of σ. These values were calculated based on the case studies that were implemented while investigating the practicality of the proposed method since there is no other real case study reported in the literature related to this. In the composition approach, OWL-S is used for annotating service preconditions-effects. Although OWL-S is widely used for service annotation in service composition, we were not able to find a dataset of OWL-S services with preconditions-effects annotations or a paper in the literature which reports on service preconditions-effects parameter distributions. Therefore, the services for the annotated feature models were generated by a customized random service generator with 250 services, and with $\mathcal{N}(1,1)$, and $\mathcal{N}(2,1)$ for the number of preconditions and effects[2].

For generating services for the feature model, a customized service model generator was implemented in order to make sure that all possible configurations of the feature model have corresponding service compositions. In this method, the generator iterates over all possible configurations of the input feature model and makes sure the required services for realizing that feature model configuration exist in the repository. For each feature model configuration, the planner is used to examine if that feature model configuration can be realized using existing services in the repository. If the feature model configuration cannot be realized using existing services, a planner is used to find a service composition whose effects have the largest intersection with

[2]All generated datasets are available for download at: `http://magus.online/experiments/1`

the feature model configuration's aggregated effects. Then, a random service generator is used to realize the remaining effects of the feature model configuration, which are not realized by the sequence generated by the planner. This service generator ensures that all feature model configurations in a service family can be realized using services in the service repository[3].

For each service family, the experiment was performed by selecting a random valid feature model configuration with 15±1 features from the feature model and generating its corresponding service compositions. Then, a random service from the generated compositions was selected and failed. The failure was handled through adaptation. This process was repeated 100 times for each service family. Having five groups and ten service families in each group, this activity was repeated 5,000 times.

Figure 6.8 shows the average time to perform feature adaptation for feature models with different numbers of possible configurations. It can be seen that the time for adaptation increases as the number of possible configurations increases since the problem becomes more complex to solve. However, it should be noted that the feature adaptation time remains fast for feature models with the size in this range (<300 ms), and increases *linearly*.

Figure 6.9 shows the distribution of successful versus failed adaptation efforts as service failure occurs. In our dataset of service families, the adaptation approach is able to recover from a failure in more than 77% of cases.

[3]The code for the tool and the service generator is available at `https://github.com/matba/magus`

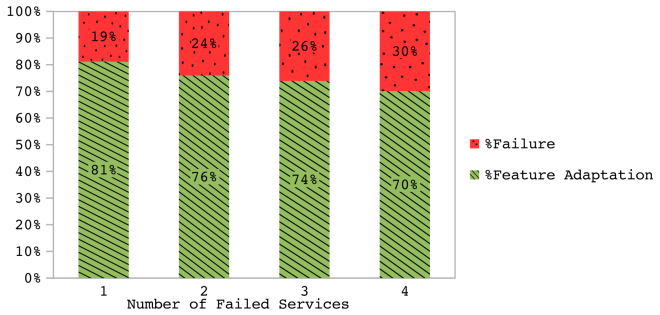


Figure 6.10: Distribution of success versus failure mechanisms after a service failure in terms of number of failed services.

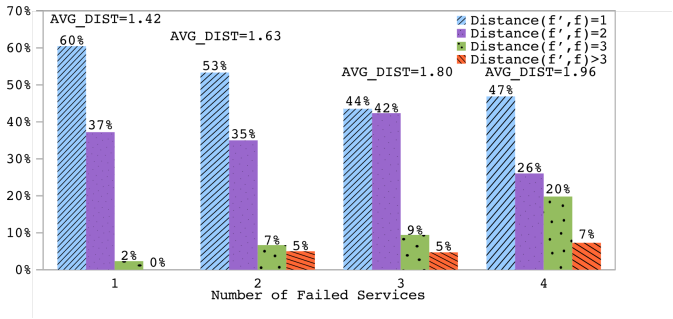


Figure 6.11: Distribution of the distance between the alternate feature model configuration and the failed one after service failure in terms of number of failed services.

However, it should be noted that the recovery rate for a service family can be influenced by different factors of how it has been realized through services and may not be the same for other service families. However, the results show that this approach can recover a failed service composition while the rate of success can differ between different service families. It can also be seen from this figure that there is no obvious relation between the size of the feature model and the success rate of the adaptation mechanism.

6.2.2 Robustness of Adaptation Towards Functional Failure

The focus of this set of experiments is on the effect of the number of failed services on the different aspects of adaptation in the service compositions to examine the robustness of the approach. In these experiments, the answer for the following research questions are looked for:

Experiment 5.1 - How successful is the adaptation mechanism in addressing a service failure as the number of failed services increases?

Experiment 5.2 - How extensively does the feature model configuration change as a result of feature model adaptation in terms of the distance between the new feature model configuration and the failed one as the number of failures increases?

The objective of the first research question is to investigate how well the proposed self-healing method performs in response to more severe failures when two or more services fail. The objective of the second research question is to determine if the changes in the adapted feature model configuration are limited enough to be used as a mitigation strategy.

For generating the dataset for this experiment, 10 service families with parameters used in the first experiment with 800 possible configurations and a service repository size of 150 was created.

In this experiment, for the number of failures between one and four, the following process was performed: a random valid feature configuration was selected from the service family and the corresponding service composition was generated. Then, a random number of services, between 1 and 4, was selected and set to failed status. The failures were then addressed by the proposed approach. This activity was repeated 250 times for each service family.

Figure 6.10 shows the distribution of successful versus failed adaptation as service failure occurs for a different number of failed services. As expected, the number of recovered service compositions decreases as the number of failed services increases. However, 70% of failures can still be addressed in the extreme case where four services fail simultaneously.

Figure 6.11 shows the distribution of the distance between the alternate feature model configuration and the failed feature model configuration for different numbers of failures. It can be seen that the average distance of the alternate feature model configuration increases as the number of failed services increases since it is more likely that more extensive changes are required as more services fail. However, the average distance between the failed feature model configuration and the alternate feature model configuration remains less than 2 features even when four services have failed.

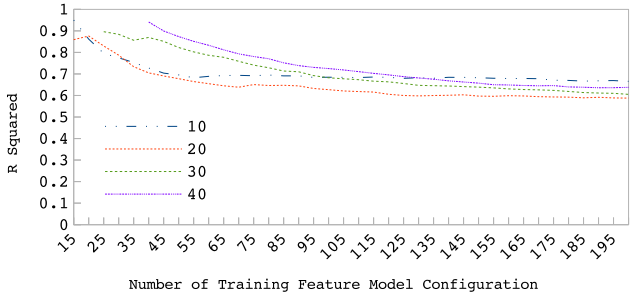


Figure 6.12: R Squared of regression model for estimation in terms of number of training feature model configurations.

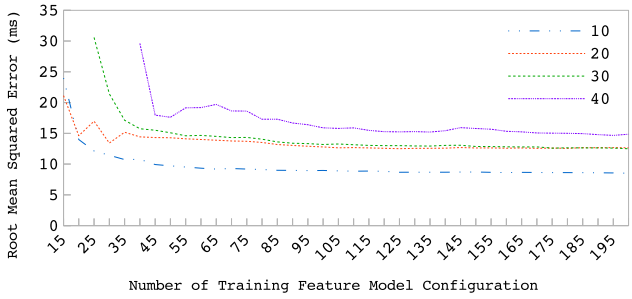


Figure 6.13: Root Mean Squared Error of the estimation method in terms of number of training feature model configurations.

6.2.3 Effectiveness of Regression For Estimating Non-functional Properties

In this set of experiments, the focus is on investigating if using a linear function is an effective way to accurately estimate the non-functional properties of a service composition. More specifically, an attempt is made to answer the two following research questions:

Experiment 6.1 - How fit is a linear function for estimating the values of non-functional properties in a service composition?

Experiment 6.2 - How accurate is the estimation approach in estimating non-functional properties for a service composition, which is not in the dataset and how does the size of the dataset affect this accuracy?

The objective of the first research question is to determine if a non-functional property of a service composition can effectively be modeled by a linear function. The goal of asking the second question is to see if an estimation function which is built based on a set of feature model configurations and their corresponding non-functional property values can be used for predicting the value of a non-functional property for a feature model configuration, which was not in the training dataset.

In order to answer the first question, *coefficient of determination* or R squared (R^2) is used, which is a metric for determining how well an estimation model has been built. R^2 is the proportion of variance in the dependent variable, which can be explained by the model for the dataset it has been built upon. The value of R^2 ranges between 0 and 1 where the value of 1 means it can perfectly model the variance and value of a zero means it cannot model the variance.

In order to perform the first experiment, the focus was on service composition response time as the non-functional property of interest. The first experiment was performed as follows: Four groups of service families were

generated in a similar way to the first experiment. Each group had five service families. The service compositions in different groups were different in terms of the number of independent features used for training the regression. The process started by annotating the services in the service families in a group with random response time with distribution $\mathcal{N}(200, 50)$. Then, the contribution values of the features were calculated using the smallest possible dataset of feature model configurations, converging to a single result. Using the contribution values, the value of R^2 was calculated. The dataset was augmented with five more feature model configurations. The response time pairs and the updated contribution values and the new R^2 value were then calculated. This process was performed three times for each service family for each group.

Figure 6.12 shows the average R^2 values for service families in each group for different dataset sizes. The value of R^2 is around 0.9 in the beginning for all groups and starts decreasing as the size of the dataset increases. The reason for this decrease is that the model built using smaller datasets overfits; therefore, it can model the data accurately while it will perform worse for new data predictions. The increase in the size of the dataset results in a more generalized model. The value of R^2 stops its descent around 0.7 for all groups. In general, having values closer to one for this metric shows that the function has modeled the relationship between the dependent variable and the independent variables more accurately and having values closer to zero shows that the model was less successful in capturing the relation. Having

this value for R^2 shows that the linear function in Equation 4.1 does model the relation between the presence of features in a feature model configuration and the response time of the service composition implementing that feature model configuration to a good extent. Furthermore, having the same value for R^2 for all groups shows that the size of the feature model does not affect the efficiency of the modeling approach for a non-functional property.

In order to measure the accuracy of our estimation model for answering the second research question, *root-mean-square error* used, which is a metric for evaluating the accuracy of regression. This metric calculates the standard deviation for error of estimation for a test dataset. Therefore, values closer to zero show that the estimation model has a better accuracy.

The experiment was designed based on the previous experiment dataset. For each service family in each group, 100 feature model configurations were selected as the test dataset. Then, starting from the smallest possible dataset, the contribution values of features were calculated using the proposed method and the value of root-mean-square error for the test data was calculated and recorded. This process was repeated three times.

Figure 6.13 shows the average root-mean-square error values for service families in each group for different dataset sizes. For all feature model sizes, the value for root-mean-square error decreases at the beginning but eventually stops decreasing. This observation confirms our speculation that the estimation model overfits data when the dataset is small. It can also be seen

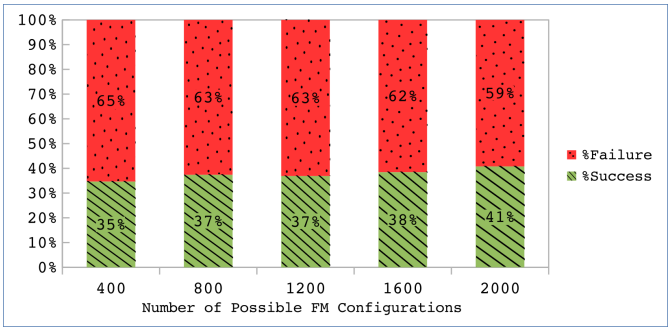


Figure 6.14: Distribution of success versus failure mechanisms after a non-functional constraint violation.

that although the value of the root-mean-square error is larger for larger feature models, the root-mean-square error remains less than 15ms for all groups. One other difference between groups with different feature model sizes is the number of instances in the dataset where adding more instances does not improve the estimation model. For smaller feature models, it happens sooner which means a smaller number of instances are needed for creating the estimation model. In other words, a larger dataset is needed for larger feature models. However, it can be seen that the estimation model for a feature model with 40 independent features, which can have up to 2^{40} feature model configurations, can be modeled using fewer than 200 feature model configurations.

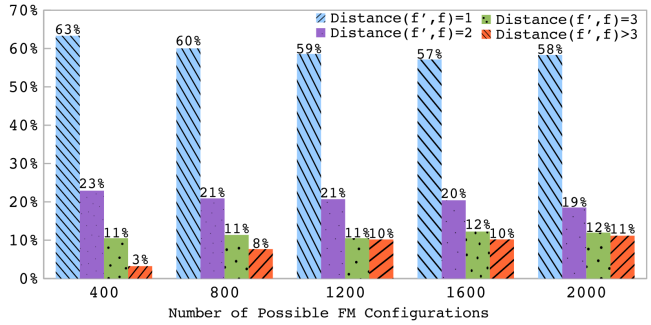


Figure 6.15: The distribution of the distance between the alternate feature model configuration and the failed one after a non-functional constraint violation.

6.2.4 Effectiveness of Feature Adaptation for Addressing Non-functional Violations

In this set of experiments, it is investigated if the proposed approach can effectively address non-functional violations of service composition. More specifically, an attempt is made to answer two research questions:

Experiment 7.1 - How successful is the adaptation mechanism in addressing non-functional failures for feature models with different sizes?

Experiment 7.2 - How extensively does the feature model configuration change as a result of feature model adaptation in response to a non-functional violation?

The objective of the first research question is to examine if the proposed feature adaptation mechanism can be used as an effective way to recover

from a failure. The goal of the second research question is to investigate if changes to the feature model configuration are small enough (minor) such that the alternate feature model configuration is still desirable for the user.

In order to answer these two questions, the experiment was designed as follows: The services in the dataset used for performing experiment 4.1 were annotated randomly with response times with the distribution of $\mathcal{N}(200, 50)$. Then, a random feature model configuration from the feature model was selected and the corresponding service composition was built. Then, the response time for the built service composition was calculated using service annotations. Then, a non-functional constraint was added having a threshold value which is equal to 110% of the calculated response time. Afterward, the services in the service composition were randomly selected and their response time was increased by 25% until the non-functional constraint no longer held. Then, the non-functional failure was addressed using the proposed self-healing approach and the result of the adaptation and the distance between the alternate feature model configuration and the failed one was recorded.

Figure 6.14 shows the success rate of adaptations for feature models with different sizes. Although the success rate of the proposed adaptation approach in addressing non-functional failure is less than its success rate in addressing functional failure, it can still address one-third of the non-functional failures. The other observation from this figure is that the growth in the size of the feature model slightly increases the chance of success.

Figure 6.15 shows the distribution of the distance between the alternate and the failed feature model configurations for feature models of different sizes. One observation from this figure is that the distance of alternate feature model configuration increases slightly as the size of the feature model increases. However, it can be seen that for all feature model sizes, the majority of failures are addressed by replacing the failed feature model configuration with an alternate one, which has a distance of 'one'. This means the alternate feature model configuration will have one feature added or removed compared to the original one.

6.2.5 Threats to Validity

We discuss the threats to the validity of our work in two parts. First we discuss the validity of the observations made in the experiments in internal validity and then we discuss potential issues which can threaten generalization of the observations in external validity.

6.2.5.1 Internal Validity

In the following, we discuss threats which can affect the validity of the observations we made in the experiments:

- In the experiments for evaluating the impact of features on non-functional properties, we assume that feature presence as an independent variable

determining the value of dependent variable which is the value of the non-functional property. However, there are other factors which can affect non-functional properties of a service considering that different BPEL processes with different non-functional properties can realize a feature model configuration. The design of the planner affects which of those processes are selected. The planner may have a bias on selecting a specific realization of a feature model configuration and therefore our observation may not be valid when a realization is selected randomly from the set of possible realizations. However, this does not affect the applicability of the proposed method used for estimating the non-functional properties of the process created using a planner.

- Considering that there are currently no real-world dataset available to perform experiments, the models used in these experiments were synthetically generated. All the parameters and distributions used in generating the models have been reported in the paper for replication studies. However, there could be unaccounted parameters that could affect the values observed in the experiments if and when such unaccounted parameters are found.

6.2.5.2 External Validity

In the following, we discuss threats which can affect the generalization of the observations made in the experiments:

- The values reported on the success rate of the adaptation in response to both functional and non-functional failures are on the domains which have been synthetic. The success rate of adaptation can change depending on the domain model as a result of change in the different properties of the domain model. The effect of this threat can be mitigated by applying the proposed method on real-world domains and reporting on the efficiency of the proposed approach on those domains.

- The way that a non-functional property for a service composition is calculated based on the value of individual services value differs for various types of non-functional properties. This can affect the accuracy and fitness of the method for estimating non-functionality of service compositions based on the selected features. This threat can be mitigated by extending Experiment 2 of the evaluation on larger sets of non-functional property types.

6.3 Summary

In this section, two series of experiments were performed to evaluate the service composition and adaptation approach, respectively. These two series of experiments investigated two different aspects of the proposed approach:

- The experiments on the service composition approach investigated the scalability of the three steps of service composition and the effectiveness

of optimization and BPEL generation. The following findings can be reported:

  - The experiments showed that the time required for service composition remains short as the size of the service composition family grows in terms of services and features. This observation supports the practicality of the proposed approach in terms of scalability.
  - The experiments showed that the optimization method remains fast for workflows of different sizes and can reduce time-to-completion for the service composition. This observation supports the practicality of the service composition method in terms of scalability and effectiveness.
  - The experiments showed that the proposed BPEL generation method remains fast for different workflow sizes and results in more structured BPEL code compared to other BPEL generation methods in the literature. This observation supports the practicality and usability of the proposed service composition method in generating BPEL code.

- The experiments on the service composition adaptation approach investigated the suitability of the regression method for estimating non-functional properties of service composition and effectiveness and robustness of the service composition adaptation in response to functional

and non-functional adaptation. From this investigation, the following findings can be reported:

- The experiments showed that a linear function over the presence of features is a good fit for estimating non-functional properties of a service composition and this function can be accurately estimated using linear regression. This observation supports the assumption that the non-functional properties of a service composition can be determined by features that are present in it.
- The experiments showed that the service composition can recover from service failure in more than 70% of the cases. This observation shows that the proposed method can be effectively used to recover from functional failures through adaptation.
- The experiments showed that service composition can recover from non-functional constraint violation in more than 35% of the cases. This observation supports the assumption that this thesis made that non-functional failure can be addressed in service composition through feature adaptation.

Chapter 7

Conclusions

This chapter concludes the thesis and provides an outlook on future work. First, a summary is presented and concluding remarks are made for the thesis. Next, open research problems for future work are described.

7.1 Concluding Remarks

This thesis first provided a comprehensive framework and tool suite based on dynamic software product line engineering, which allows a non-expert user to develop service composition to satisfy their requirements. The developed service composition can self-adapt in response to functional and non-functional changes in the services in order to provide critical functional and

non-functional requirements. In the following, a number of notes that can be concluded from the work done in this thesis is reviewed.

- *The proposed dynamic software product line engineering taxonomy can be used for comparing existing DSPL engineering approaches or developing a new one based on the adaptation requirements.* A taxonomy on design aspects of a DSPL is introduced and the effect of each of these design decisions on the properties of the adaptations the DSPL provides is discussed. This framework can be used for selecting an existing DSPL engineering approach or designing a new one based on the requirements of the problem domain.

- *The software product line engineering can be used to enable non-expert users to build their desired service composition which is readily executable.* In this thesis, a dynamic software product line engineering approach was introduced for service composition. The service composition method works on a feature model as variability model and builds an executable service composition represented as BPEL code. The process of composition is performed by reducing the problem of finding the workflow which provides requested features to a planning problem and solving it using an existing planner. This thesis also proposes a method for optimizing the generated workflow using an optimization method and finally generating BPEL code based on the workflow.

- *The automated composition of services based on the user's requirements results in optimized and structured service composition and can be performed in reasonable time.* This thesis evaluated the composition approach using different service composition families with a different number of features and services. The evaluation showed that the composition method finishes in reasonable time for problems of different size and results in a service composition which is optimized and structured.

- *DSPL models and methods can be used to enable service composition to adapt in response to functional and non-functional failures autonomously.* This thesis proposed a method which recovers from functional and non-functional failures using an alternative set of features which still provides the properties which are critical for the user. In order to do so, a method which estimates the contribution of each feature to the non-functional properties of the service composition using linear regression is proposed. Using this information, finding the subset of features which respect critical constraints and provide the maximum number of features desired by the user was represented as a pseudo-boolean optimization problem. Solving this problem results in a new set of features which satisfies critical functional and non-functional requirements of the user.

- *Linear regression is a fit and accurate method for estimating the contribution value of each feature to non-functional properties of the service*

composition. This thesis evaluated the linear regression method for estimating the contribution value of each feature to the non-functional property of a service composition for both goodness of fit (using the R-squared metric) and accuracy (using the root mean squared error metric) using service composition families with different sizes. Our evaluation showed that linear regression is a fit and accurate method for estimating the non-functional properties of a service composition based on its features.

- *The proposed service composition adaptation method can effectively be used for addressing functional and non-functional failures.* This thesis evaluated the proposed adaptation method for service composition families of different sizes. The adaptation method was able to recover from more than 70 percent of functional failures and 35 percent of non-functional failures.

- *The proposed method for composing self-adaptive service composition is available to be used for addressing real-world problems.* In order to investigate if the proposed method in this thesis can be offered as a tool to address real-world problems, the proposed method is implemented as an online tool which is publicly available to be applied on real problems by practitioners.

7.2 Future Work

The work presented in this thesis for developing self-adaptive service composition can be improved from a number of aspects with additional research. In the following, some of these aspects are discussed and directions for future research are provided:

- This thesis worked on sample case studies. Additionally, the evaluation of this thesis was performed on simulated services. Applying the proposed method to real-world problems can better evaluate the assumptions made in this thesis and the effectiveness of the proposed approach. It can also introduce new challenges that need to be addressed. There, the focus will be on solving a real-world problem and reporting on lessons learnt.

- This work allows a service composition to lose features or use alternative features to recover from functional and non-functional failures. This new approach allows service compositions to recover from failures which existing approaches cannot recover from. However, changing features as the first step for addressing a failure may not be desirable. In [16], we proposed a method for recovering from functional failure where the adaptation management mechanism tries to address a functional failure by using alternative services or alternative workflows and feature adaptation is performed as the last resort. Future work will

extend the approach proposed to address both functional and non-functional properties while keeping existing features and attempting feature adaptation when that is not possible.

- The feature adaptation method only prioritizes non-critical features for removal when their utility has been provided and otherwise, it assumes that all non-critical features have the same value for the user. Having an effective way to extract user preferences over features when deciding on feature removal can improve users' experiences. In [17], we proposed a method for eliciting users' preferences over features using a method which can estimate the utility of each feature with a series of simple questions. Future work focuses on integrating these two approaches in order to improve the desirability of adaptation decisions made by the proposed method.

- Currently, the focus on micro-services architecture [108] has been increasing significantly in software research [47]. Additionally, with the introduction of container-based platforms like Docker and micro-services frameworks like Apache Kafka, this approach is being used in enterprise system architectures in a wide-spread manner. Micro-services are small and isolated services which focus on accomplishing one functionality. Our approach can be also used to enable adaptation on systems with such architecture. Future work investigates on the application of the proposed approach on systems with micro-services architecture.

- The proposed approach in this thesis for self-adaptation focuses on satisfying non-functional constraints for the service composition. However, the proposed method does not guarantee that the adapted service composition provides the best non-functional properties possible with existing features. There is existing work in the literature which focuses on optimizing non-functional properties of the system [142, 64]. Future work will improve the proposed adaptation approach such that it guarantees the best non-functional property values.

- This work proves that the service composition generated based on the selected features provides the requested functionality. However, this proof relies on the assumption that features and services have been annotated correctly. This may not be always the case since it is possible that domain model designers make mistakes in capturing or modeling the relations between features and services. As a result of such mistakes, the alternate service composition generated in the process of adaptation may not be executable or it may behave in an unexpected way. Future work will focus on providing mechanisms to verify and validate the generated service composition on-the-fly before deploying.

List of Abbreviations

ADL	Architecture Description Language
AI	Artificial Intelligence
API	Application Programming Interface
BPEL	Business Process Execution Language
BPML	Business Process Modeling Language
BPMN	Business Process Model and Notation
CSP	Constraint Satisfaction Problem
DSPL	Dynamic Software Product Line
ECA	Event Condition Action
GUI	Graphical User Interface
HTN	Hierarchical Task Network
MAPE-K	Monitoring, Analysis, Planning, Execution, and Knowledge
MDD	Model-Driven Development
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
PBO	Pseudo-Boolean Optimization
PDDL	Planning Domain Definition Language
REST	Representational State Transfer
SAT	Satisfiability
SOA	Service-oriented Architecture
SPL	Software Product Line
SPLE	Software Product Line Engineering
UML	Unified Modeling Language
WS-BPEL	Web Services Business Process Execution Language

Index

adaptation management engine, 167
aggregation function, 127
application engineering phase, 21

BPEL, 108
business process execution language, 108

context model, 83

domain design tool, 163
domain engineering phase, 21
domain models, 77
DSPL, 23
dynamic software product line, 23

feature model, 22, 79
feature model annotation, 85
feature model configuration, 23, 80
flow, 108
functional failure, 140

independent feature set, 120

linear regression, 129
link, 108
loss of utility, 146

MAGUS, 161
MAPE-K loop, 19

non-functional constraints, 153
non-functional failure, 140

OLS, 131
ordinary least squares, 131

PBO, 143
PDDL, 89
planning domain, 89
planning domain definition language, 89
planning problem, 89
pseudo-boolean optimization, 143

runtime adaptation, 18

safety, 100
self-adaptive, 18
self-configuring, 36
self-healing, 37
self-optimizing, 37
sequence, 108
service, 81
service composition configuration tool, 165
service model annotation, 87
software product line engineering, 21
SPLE, 21

threat, 103

workflow, 82
WS-BPEL, 108

Bibliography

[1] Komi S. Abotsi, S. Tonny Kurniadi, Hamad I. Alsawalqah, and Danhyung Lee, *A software product line-based self-healing strategy for web-based applications*, Proceedings of the 15th International Software Product Line Conference, Volume 2, ACM, 2011, p. 31.

[2] Mohammad Abu-Matar and Hassan Gomaa, *Variability modeling for service oriented product line architectures*, International Software Product Line Conference (SPLC), IEEE, 2011, pp. 110–119.

[3] M. Acher, P. Collet, P. Lahire, S. Moisan, and J. P Rigault, *Modeling variability from requirements to runtime*, 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), april 2011, p. 77.

[4] Akamai, *Akamai online retail performance report: Milliseconds are critical*, `https://www.akamai.com/uk/en/about/news/press/2017-press`, Accessed: 2017-12-26.

[5] German H. Alferez and Vicente Pelechano, *Context-aware autonomous web services in software product lines*, International Software Product Line Conference (SPLC), IEEE, 2011, pp. 100–109.

[6] Germán H Alférez, Vicente Pelechano, Raúl Mazo, Camille Salinesi, and Daniel Diaz, *Dynamic adaptation of service compositions with variability models*, Journal of Systems and Software **91** (2014), 24–47.

[7] Manar S. Ali and Stephan Reiff-Marganiec, *Autonomous failure-handling mechanism for WF long running transactions*, Services Computing (SCC), 2012 IEEE Ninth International Conference on, IEEE, 2012, pp. 562–569.

[8] V. Alves, D. Schneider, M. Becker, N. Bencomo, and P. Grace, *Comparitive study of variability management in software product lines and runtime adaptable systems*, Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2009), 2009, pp. 9–17.

[9] J. Andersson, R. De Lemos, S. Malek, and D. Weyns, *Modeling dimensions of self-adaptive software systems*, Software Engineering for Self-adaptive Systems (SEAMS), Springer, 2009, pp. 27–47.

[10] Rafael Angarita, Marta Rukoz, and Yudith Cardinale, *Modeling dynamic recovery strategy for composite web services execution*, World Wide Web **19** (2016), no. 1, 89–109.

[11] Sven Apel and Christian Kstner, *An overview of feature-oriented software development.*, Journal of Object Technology **8** (2009), no. 5, 49–84.

[12] Mohsen Asadi, Bardia Mohabbati, Gerd Groner, and Dragan Gasevic, *Development and validation of customized process models*, Journal of Systems and Software **96** (2014), 73–92.

[13] Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Samaneh Soltani, *Stratified analytic hierarchy process: Prioritization and selection of software features*, International Conference on Software Product Lines (SPLC 2010), Springer, 2010, pp. 300–315.

[14] L. Baresi, S. Guinea, and L. Pasquale, *Service-oriented dynamic software product lines*, Computer **45** (2012), no. 10, 42.

[15] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, *Automated composition of service mashups through software product line engineering*, International Conference on Software Reuse, Springer, 2016, pp. 20–38.

[16] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, *Self-healing in service mashups through feature adaptation*, Proceedings of the 21st International Systems and Software Product Line Conference - Volume A (New York, NY, USA), SPLC '17, ACM, 2017, pp. 94–103.

[17] Mahdi Bashari, Mahdi Noorian, and Ebrahim Bagheri, *Product line stakeholder preference elicitation via decision processes*, International

Journal of Knowledge and Systems Science (IJKSS) **5** (2014), no. 4, 35–51.

[18] D. Benavides, P. Trinidad, and A. Ruiz-Cortes, *Automated reasoning on feature models*, Advanced Information Systems Engineering, Springer, 2005, pp. 381–390.

[19] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortes, *FAMA: Tooling a framework for the automated analysis of feature models*, Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS), 2007, pp. 129–134.

[20] N. Bencomo, S. Hallsteinsen, and E. Almeida, *A view of the landscape of dynamic software product lines*, Computer **45** (2012), no. 10, 36–41.

[21] N. Bencomo, P. Sawyer, G. Blair, and P. Grace, *Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems*, 2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), Limerick, Ireland, vol. 38, 2008, p. 40.

[22] Nelly Bencomo, Gordon S. Blair, Carlos A. Flores-Cortes, and Peter Sawyer, *Reflective component-based technologies to support dynamic variability.*, Workshop on Variability Modeling of Software-intensive Systems (VaMoS), 2008, pp. 141–150.

[23] Djamal Benslimane, Schahram Dustdar, and Amit Sheth, *Services mashups: The new generation of web applications*, IEEE Internet Computing **5** (2008), 13–15.

[24] James O Berger, *Statistical decision theory and bayesian analysis*, Springer Science & Business Media, 2013.

[25] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso, *Automated composition of web services via planning in asynchronous domains*, Artificial Intelligence **174** (2010), no. 3, 316–361.

[26] Endre Boros and Peter L. Hammer, *Pseudo-boolean optimization*, Discrete applied mathematics **123** (2002), no. 1, 155–225.

[27] Jan Bosch, *Design and use of software architectures: Adopting and evolving a product-line approach*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[28] Boudjemaa Boudaa, Slimane Hammoudi, Leila Amel Mebarki, Abdelkader Bouguessa, and Mohammed Amine Chikh, *An aspect-oriented model-driven approach for building adaptable context-aware service-based applications*, Science of Computer Programming **136** (2017), 17–42.

[29] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani, *The Fractal component model and its sup-*

port in Java, Software: Practice and Experience **36** (2006), no. 11-12, 1257–1284.

[30] Cinzia Cappiello, Maristella Matera, Matteo Picozzi, Gabriele Sprega, Donato Barbagallo, and Chiara Francalanci, *Dashmash: a mashup environment for end user development*, Web Engineering, pp. 152–166, Springer, 2011.

[31] Antonio Carzaniga, Alessandra Gorla, Nicolo Perino, and Mauro Pezze, *Automatic workarounds: Exploiting the intrinsic redundancy of web applications*, ACM Transactions on Software Engineering and Methodology (TOSEM) **24** (2015), no. 3, 16.

[32] C. Cetina, P. Giner, J. Fons, and V. Pelechano, *Designing and prototyping dynamic software product lines: techniques and guidelines*, Software Product Lines: Going Beyond (2010), 331–345.

[33] Carlos Cetina, Joan Fons, and Vicente Pelechano, *Applying software product lines to build autonomic pervasive systems*, Software Product Line Conference (SPLC), IEEE, 2008, pp. 117–126.

[34] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano, *Autonomic computing through reuse of variability models at runtime: The case of smart homes*, Computer **42** (2009), no. 10, 37–43.

[35] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, and et al., *Software*

engineering for self-adaptive systems: A research roadmap, Software Engineering for Self-Adaptive Systems (2009), 1–26.

[36] Kareliotis Christos, Costas Vassilakis, Efstathios Rouvas, and Panayiotis Georgiadis, *Exception resolution for BPEL processes: a middleware-based framework and performance evaluation*, Proceedings of the 10th International Conference on Information Integration and Web-based Applications and Services, ACM, 2008, pp. 248–256.

[37] Paul C. Clements, *A survey of architecture description languages*, Proceedings of the 8th International Workshop on Software Specification and Design (Washington, DC, USA), IWSSD '96, IEEE Computer Society, 1996, p. 16.

[38] Geoff Coulson, Gordon S. Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama, *A component model for building systems software*, SEA 2004: IASTED Software Engineering and Applications, 2004.

[39] Javier Cubo, Nadia Gamez, Lidia Fuentes, and Ernesto Pimentel, *Composition and self-adaptation of service-based systems with feature models*, Safe and Secure Software Reuse, pp. 326–342, Springer, 2013.

[40] Krzysztof Czarnecki and Michal Antkiewicz, *Mapping features to models: A template approach based on superimposed variants*, Generative Programming and Component Engineering, Springer, 2005, pp. 422–437.

[41] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glck, David Vandevoorde, and Todd Veldhuizen, *Generative programming and active libraries*, Generic Programming (2000), 25–39.

[42] Nelly Delgado, Ann Q. Gates, and Steve Roach, *A taxonomy and catalog of runtime software-fault monitoring tools*, IEEE Transactions on Software Engineering **30** (2004), no. 12, 859–872.

[43] Mahdi Derakhshanmanesh, Mazeiar Salehie, and Jurgen Ebert, *Towards model-centric engineering of a dynamic access control product line*, Proceedings of the 16th International Software Product Line Conference-Volume 2, ACM, 2012, pp. 151–155.

[44] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini, *A dynamic software product line approach using aspect models at runtime*, 5th Domain-Specific Aspect Languages Workshop, 2010.

[45] Glen Dobson, *Using WS-BPEL to implement software fault tolerance for web services*, 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06), IEEE, 2006, pp. 126–133.

[46] Christoph Dorn, Daniel Schall, and Schahram Dustdar, *Context-aware adaptive service mashups*, Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific, IEEE, 2009, pp. 301–306.

[47] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina, *Microservices: yesterday, today, and tomorrow*, Present and Ulterior Software Engineering, Springer, 2017, pp. 195–216.

[48] Niklas Een and Niklas Sorensson, *Translating pseudo-boolean constraints into SAT*, Journal on Satisfiability, Boolean Modeling and Computation **2** (2006), 1–26.

[49] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, *A survey of rollback-recovery protocols in message-passing systems*, ACM Computing Surveys (CSUR) **34** (2002), no. 3, 375–408.

[50] Richard E. Fikes and Nils J. Nilsson, *Strips: A new approach to the application of theorem proving to problem solving*, Artificial Intelligence **2** (1972), no. 3, 189–208.

[51] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh, *A generic approach for automatic model composition*, Models in software engineering, pp. 7–15, Springer, 2008.

[52] Franck Fleurey and Arnor Solberg, *A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems*, Model Driven Engineering Languages and Systems, pp. 606–621, Springer, 2009.

[53] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, *Using architecture models for runtime adaptability*, Software, IEEE **23** (2006), no. 2, 62–70.

[54] Jacqueline Floch, Cristina Fr, Rolf Fricke, Kurt Geihs, Michael Wagner, J. Lorenzo, E. Soladana, Stephan Mehlhase, Nearchos Paspallis, and Hossein Rahnama, *Playing music: building context-aware and self-adaptive mobile applications*, Software: Practice and Experience **43** (2012), no. 3, 359–388.

[55] Keita Fujii and Tatsuya Suda, *Semantics-based dynamic web service composition*, International Journal of Cooperative Information Systems **15** (2006), no. 03, 293–324.

[56] Kurt Geihs, Paolo Barone, Frank Eliassen, Jacqueline Floch, Rolf Fricke, Eli Gjorven, Svein Hallsteinsen, Geir Horn, Mohammad Ullah Khan, Alessandro Mamelli, et al., *A comprehensive solution for application-level adaptation*, Software: Practice and Experience **39** (2009), no. 4, 385–422.

[57] Yaser Ghanam and Frank Maurer, *Linking feature models to code artifacts using executable acceptance tests*, Software Product Lines: Going Beyond, pp. 211–225, Springer, 2010.

[58] Brian K. Gibb and Suresh Damodaran, *ebXML: Concepts and application*, John Wiley and Sons, Inc., 2002.

[59] Michael Goedicke, Carsten Kollmann, and Uwe Zdun, *Designing runtime variation points in product line architectures: three cases*, Science of Computer Programming **53** (2004), no. 3, 353–380.

[60] Hassan Gomaa and Koji Hashimoto, *Dynamic software adaptation for service-oriented product lines*, Proceedings of the 15th International Software Product Line Conference, Volume 2, ACM, 2011, p. 35.

[61] Hassan Gomaa and Mohamed Hussein, *Dynamic software reconfiguration in software product families*, Software Product-Family Engineering, pp. 435–444, Springer, 2004.

[62] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg, *On the notion of variability in software product lines*, Proceedings of Working IEEE/IFIP Conference on Software Architecture, IEEE, 2001, pp. 45–54.

[63] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, *Dynamic software product lines*, Computer **41** (2008), no. 4, 93–95.

[64] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch, *Using product line techniques to build adaptive systems*, Software Product Line Conference (SPLC), IEEE, 2006, pp. 10 pp.–150.

[65] Ourania Hatzi, Dimitris Vrakas, Nick Bassiliades, Dimosthenis Anagnostopoulos, and Ioannis Vlahavas, *The PORSCE II framework: Using AI planning for automated semantic web service composition*, The Knowledge Engineering Review **28** (2013), no. 02, 137–156.

[66] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza, Ana Moreira, and Awais Rashid, *Relating feature models to other models of a software product line*, Transactions on aspect-oriented software development VII, pp. 69–114, Springer, 2010.

[67] Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, and Wim Van Betsbrugge, *Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines*, Proceedings of the Third International Workshop on Dynamic Software Product Lines (DSPL@ SPLC 2009), 2009, pp. 18–27.

[68] Mike Hinchey, Sooyong Park, and Klaus Schmid, *Building dynamic software product lines*, Computer **45** (2012), no. 10, 22–26.

[69] Jorg Hoffmann and Bernhard Nebel, *The FF planning system: Fast plan generation through heuristic search*, Journal of Artificial Intelligence Research (2001), 253–302.

[70] Anna Hristoskova, Bruno Volckaert, and Filip De Turck, *The WTE framework: automated construction and runtime adaptation of service mashups*, Automated Software Engineering **20** (2013), no. 4, 499–542.

[71] Hui Huang, Xueguang Chen, and Zhiwu Wang, *Failure recovery in distributed model composition with intelligent assistance*, Information Systems Frontiers **17** (2015), no. 3, 673–689.

[72] M. C. Huebscher and J. A. McCann, *A survey of autonomic computing:degrees, models, and applications*, ACM Comput.Surv **40** (2008), no. 3, 1–28.

[73] Azlan Ismail and Valeria Cardellini, *Towards self-adaptation planning for complex service-based systems*, International Conference on Service-Oriented Computing, Springer, 2013, pp. 432–444.

[74] Paul Istoan, Gregory Nain, Gilles Perrouin, and J-M Jezequel, *Dynamic software product lines for service-based systems*, Ninth IEEE International Conference on Computer and Information Technology (CIT), vol. 2, IEEE, 2009, pp. 193–198.

[75] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, *FORM: A feature-oriented reuse method with domain-specific reference architectures*, Annals of Software Engineering **5** (1998), no. 1, 143–168.

[76] Christian Kastner, Sven Apel, and Don Batory, *A case study implementing features using AspectJ*, International Software Product Line Conference (2007), IEEE, 2007, pp. 223–232.

[77] J. O. Kephart and D. M. Chess, *The vision of autonomic computing*, Computer **36** (2003), no. 1, 41–50.

[78] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, *An overview of AspectJ*, ECOOP 2001-Object-Oriented Programming, pp. 327–354, Springer, 2001.

[79] Matthias Klusch, Andreas Gerber, and Marcus Schmidt, *Semantic web service composition planning with OWLS-XPLAN*, AAAI Fall Symposium on Semantic Web and Agents, 2005.

[80] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, and Brad Myers, *The state of the art in end-user software engineering*, ACM Computing Surveys (CSUR) **43** (2011), no. 3, 21.

[81] Michiel Koning, Chang ai Sun, Marco Sinnema, and Paris Avgeriou, *VxBPEL: Supporting variability for web services in BPEL*, Information and Software Technology **51** (2009), no. 2, 258–269.

[82] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais, and Jean-Marc Jezequel, *Introducing variability into aspect-oriented modeling approaches*, Model Driven Engineering Languages and Systems, pp. 498–513, Springer, 2007.

[83] Kung-Kiu Lau and Zheng Wang, *Software component models*, Software Engineering, IEEE Transactions on **33** (2007), no. 10, 709–724.

[84] J. Lee, G. Kotonya, and D. Robinson, *Engineering service-based dynamic software product lines*, Computer **45** (2012), no. 10, 49–55.

[85] Jaejoon Lee and Gerald Kotonya, *Combining service-orientation with product line engineering*, Software, IEEE **27** (2010), no. 3, 35–41.

[86] Kwanwoo Lee, Kyo Kang, and Jaejoon Lee, *Concepts and guidelines of feature modeling for product line software engineering*, Software Reuse: Methods, Techniques, and Tools (2002), 62–77.

[87] Leonardo AF Leite, Gustavo Ansaldi Oliva, Guilherme M. Nogueira, Marco Aurélio Gerosa, Fabio Kon, and Dejan S. Milojicic, *A systematic literature review of service choreography adaptation*, Service Oriented Computing and Applications (2012), 1–18.

[88] F. Leymann, *Web services flow language (WSFL 1.0)*, Tech. report, IBM Corporation, 2001.

[89] Xiaoli Lian, Li Zhang, Jing Jiang, and William Goss, *An approach for optimized feature selection in large-scale software product lines*, Journal of Systems and Software **137** (2018), 636 – 651.

[90] An Liu, Qing Li, and Mingjun Xiao, *A declarative approach to enhancing the reliability of BPEL processes*, IEEE International Conference on Web Services (ICWS 2007), IEEE, 2007, pp. 272–279.

[91] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes, *Algorithms for weighted boolean optimization*, International conference on theory and applications of satisfiability testing, Springer, 2009, pp. 495–508.

[92] Dave Marples and Peter Kriens, *The open services gateway initiative: An introductory overview*, Communications Magazine, IEEE **39** (2001), no. 12, 110–114.

[93] David McAllester and David Rosenblatt, *Systematic nonlinear planning*, Tech. Report AIM-1339, MIT, 1991.

[94] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins, *PDDL - the planning domain definition language*, Tech. report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[95] Drew V. McDermott, *Estimated-regression planning for interactions with web services.*, AIPS **2** (2002), 204–211.

[96] Deborah L. McGuinness and Frank Van Harmelen, *OWL web ontology language overview*, W3C recommendation **10** (2004), no. 2004-03, 10.

[97] Flávio Mota Medeiros, Eduardo Santana De Almeida, and Silvio Romero de Lemos Meira, *Towards an approach for service-oriented product line architectures*, Proceedings of the Workshop on Service-oriented Architectures and Software Product Lines, 2009, pp. 1–7.

[98] Daniel A. Menasce, Emiliano Casalicchio, and Vinod Dubey, *A heuristic approach to optimal service selection in service oriented architectures*, Proceedings of the 7th international workshop on Software and performance, ACM, 2008, pp. 13–24.

[99] Marcilio Mendonca, Moises Branco, and Donald Cowan, *SPLOT: software product lines online tools*, Proceedings of the 24th ACM SIG-

PLAN conference companion on Object oriented programming systems languages and applications, ACM, 2009, pp. 761–762.

[100] Microsoft, *Windows workflow foundation*.

[101] Bardia Mohabbati, Dragan Gasevic, Marek Hatala, Mohsen Asadi, Ebrahim Bagheri, and Marko Boskovic, *A quality aggregation model for service-oriented software product lines based on variability and composition patterns*, Service-Oriented Computing - 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5-8, 2011 Proceedings, 2011, pp. 436–451.

[102] Bardia Mohabbati, Marek Hatala, Dragan Gasevic, Mohsen Asadi, and Marko Boskovic, *Development and configuration of service-oriented systems families*, Proceedings of the 2011 ACM Symposium on Applied Computing, ACM, 2011, pp. 1606–1613.

[103] B. Morin, O. Barais, J. M. Jezequel, F. Fleurey, and A. Solberg, *Models@Runtime to support dynamic adaptation*, Computer **42** (2009), no. 10, 44–51.

[104] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel, *Taming dynamically adaptive systems using models and aspects*, Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, 2009, pp. 122–132.

[105] Oliver Moser, Florian Rosenberg, and Schahram Dustdar, *Non-intrusive monitoring and service adaptation for WS-BPEL*, Proceedings of the 17th international conference on World Wide Web, ACM, 2008, pp. 815–824.

[106] Javier Munoz and Vicente Pelechano, *Applying software factories to pervasive systems: A platform specific framework.*, ICEIS (3), Citeseer, 2006, pp. 337–342.

[107] Vivek Nallur and Rami Bahsoon, *A decentralized self-adaptation mechanism for service-based applications in the cloud*, IEEE Transactions on Software Engineering **39** (2013), no. 5, 591–612.

[108] Sam Newman, *Building microservices: designing fine-grained systems*, O'Reilly Media, Inc., 2015.

[109] A. H. H. Ngu, M. P. Carlson, Q. Z. Sheng, and Hye young Paik, *Semantic-based mashup of composite applications*, Services Computing, IEEE Transactions on **3** (2010), no. 1, 2–15, ID: 1.

[110] Gu Ning, Yiwen Zhu, Tun Lu, and Fang Wang, *BPELGEN: an algorithm of automatically converting from web services composition plan to BPEL4WS*, ICPCA 2007, IEEE, 2007, pp. 600–605.

[111] Mahdi Noorian, Ebrahim Bagheri, and Weichang Du, *Toward automated quality-centric product line configuration using intentional vari-*

ability, Journal of Software: Evolution and Process **29** (2017), no. 9, e1870.

[112] OMG, *Business process modeling notation specification, version 1.2*, Tech. Report formal/09-01-03, Object Management Group, January 2009.

[113] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, *An architecture-based approach to self-adaptive software*, Intelligent Systems and Their Applications, IEEE **14** (1999), no. 3, 54–62.

[114] P. Oreizy, N. Medvidovic, and R. N. Taylor, *Architecture-based runtime software evolution*, Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, 1998, pp. 177–186.

[115] Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien, *Unifying design and runtime software adaptation using aspect models*, Science of Computer Programming **76** (2011), no. 12, 1247–1260.

[116] Carlos Parra, Xavier Blanc, and Laurence Duchien, *Context awareness for dynamic service-oriented product lines*, Proceedings of the 13th International Software Product Line Conference, Carnegie Mellon University, 2009, pp. 131–140.

[117] Carlos Parra, Daniel Romero, Sebastien Mosser, Romain Rouvoy, Laurence Duchien, and Lionel Seinturier, *Using constraint-based optimiza-*

tion and variability to support continuous self-adaptation, Proceedings of the 27th Annual ACM Symposium on Applied Computing, ACM, 2012, pp. 486–491.

[118] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and J-M Jezequel, *Reconciling automation and flexibility in product derivation*, International Software Product Line Conference (SPLC), IEEE, 2008, pp. 339–348.

[119] K. Pohl, G. Bockle, and F. Van Der Linden, *Software product line engineering: Foundations, principles and techniques*, vol. 10, Springer, 2005.

[120] Andrei Popovici, Thomas Gross, and Gustavo Alonso, *Dynamic weaving for aspect-oriented programming*, Proceedings of the 1st international conference on Aspect-oriented software development, ACM, 2002, pp. 141–147.

[121] Jinghai Rao, Peep Kungas, and Mihhail Matskin, *Logic-based web services composition: From service description to process model*, Web Services, 2004. Proceedings. IEEE International Conference on, IEEE, 2004, pp. 446–453.

[122] Hajo A. Reijers and Jan Mendling, *A study into the factors that influence the understandability of business process models*, Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on **41** (2011), no. 3, 449–462.

[123] Pablo Rodriguez-Mier, Manuel Mucientes, and Manuel Lama, *Automatic web service composition with a heuristic-based search algorithm*, ICWS 2011, IEEE, 2011, pp. 81–88.

[124] Juan F. Ingles Romero, Cristina Vicente Chicote, Brice Morin, and Olivier Barais, *Using models@ Runtime for designing adaptive robotics software: an experience report*, Tech. report, Laurent Riox, 2010.

[125] M. Rosenmller, N. Siegmund, M. Pukall, and S. Apel, *Tailoring dynamic software product lines*, Proceedings of the 10th ACM international conference on Generative programming and component engineering, ACM, 2011, pp. 3–12.

[126] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, *Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments*, Software Engineering for Self-adaptive Systems (SEAMS), Springer, 2009, pp. 164–182.

[127] Masahiko Sakai and Hidetomo Nabeshima, *Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers*, IEICE Transactions on Information and Systems **98** (2015), no. 6, 1121–1127.

[128] M. Salehie and L. Tahvildari, *Self-adaptive software: Landscape and research challenges*, ACM Transactions on Autonomous and Adaptive Systems (TAAS) **4** (2009), no. 2, 14.

[129] Ina Schaefer, *Variability modelling for model-driven development of software product lines.*, Proceeding of the 4th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS), 2010, pp. 85–92.

[130] Sergio Segura, Jose A. Galindo, David Benavides, Jose A. Parejo, and Antonio Ruiz-Cortes, *Betty: benchmarking and testing on the automated analysis of feature models*, Workshop on Variability Modeling of Software-Intensive Systems, ACM, 2012, pp. 63–71.

[131] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu, *Web services composition: A decades overview*, Information Sciences **280** (2014), 218–238.

[132] Amit P. Sheth, Karthik Gomadam, and Jon Lathem, *SA-REST: semantically interoperable and easier-to-use services and mashups*, IEEE Internet Computing **11** (2007), no. 6, 91–94.

[133] Hesham Shokry and M. Ali Babar, *Dynamic software product line architectures using service-based computing for automotive systems.*, Proceedings of the Second International Workshop on Dynamic Software Product Lines (DSPL@ SPLC 2008), 2008.

[134] Evren Sirin, Bijan Parsia, Dan Wu, James Hendler, and Dana Nau, *HTN planning for web service composition using SHOP2*, Web Semantics: Science, Services and Agents on the World Wide Web **1** (2004), no. 4, 377–396.

[135] Brian Cantwell Smith, *Reflection and semantics in a procedural language (ph. d. thesis)*, Technical Report (1982), 1.

[136] Helmuth Späth, *Mathematical algorithms for linear regression*, Academic Press, 2014.

[137] Sattanathan Subramanian, Philippe Thiran, Nanjangud C. Narendra, Ghita Kouadri Mostefaoui, and Zakaria Maamar, *On the enhancement of BPEL engines for self-healing composite web services*, Applications and the Internet, 2008. SAINT 2008. International Symposium on, IEEE, 2008, pp. 33–39.

[138] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch, *A taxonomy of variability realization techniques*, Software: Practice and Experience **35** (2005), no. 8, 705–754.

[139] Yang Syu, Shang-Pin Ma, Jong-Yih Kuo, and Yong-Yi FanJiang, *A survey on automated service composition methods and related techniques*, 2012 IEEE Ninth International Conference on Services Computing (SCC), IEEE, 2012, pp. 290–297.

[140] Yang Syu, Shang-Pin Ma, Jong-Yin Kuo, and Yong-Yi FanJiang, *A survey on automated service composition methods and related techniques*, Services Computing (SCC), 2012 IEEE Ninth International Conference on, 2012, ID: 1, pp. 290–297.

[141] Clemens Szyperski, *Component software: beyond object-oriented programming*, Pearson Education, 2002.

[142] Tian Huat Tan, Manman Chen, Etienne Andre, Jun Sun, Yang Liu, and Jin Song Dong, *Automated runtime recovery for QoS-based service composition*, Proceedings of the 23rd international conference on World wide web, ACM, 2014, pp. 563–574.

[143] R. N. Taylor, N. Medvidovic, and P. Oreizy, *Architectural styles for runtime software adaptation*, European Conference on Software Architecture (WICSA/ECSA), IEEE, 2009, pp. 171–180.

[144] Satish Thatte, *XLANG: Web services for business process design*, Microsoft Corporation **2001** (2001), 1.

[145] Rajesh K. Thiagarajan, Amit K. Srivastava, Ashis K. Pujari, and Visweswar K. Bulusu, *BPML: a process modeling language for dynamic business models*, Advanced Issues of E-Commerce and Web-Based Information Systems, 2002.(WECWIS 2002). Proceedings. Fourth IEEE International Workshop on, IEEE, 2002, pp. 222–224.

[146] Thomas Thum, Don Batory, and Christian Kastner, *Reasoning about edits to feature models*, ICSE, IEEE, 2009, pp. 254–264.

[147] Paolo Traverso and Marco Pistore, *Automated composition of semantic web services into executable processes*, The Semantic WebISWC 2004, pp. 380–394, Springer, 2004.

[148] P. Trinidad, A. Ruiz-Cortes, J. Pena, and D. Benavides, *Mapping feature models onto component models to build dynamic software product lines*, International Workshop on Dynamic Software Product Line, DSPL, 2007.

[149] Wil MP Van Der Aalst and Kristian Bisgaard Lassen, *Translating unstructured workflow processes to readable bpel: Theory and implementation*, Information and Software Technology **50** (2008), no. 3, 131–159.

[150] Markus Voelter and Iris Groher, *Product line implementation using aspect-oriented and model-driven software development*, International Software Product Line Conference (SPLC), IEEE, 2007, pp. 233–242.

[151] Danny Weyns, M. Usman Iftikhar, Sam Malek, and Jesper Andersson, *Claims and supporting evidence for self-adaptive systems: A literature study*, ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), IEEE, 2012, pp. 89–98.

[152] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt, *Improving domain-specific language*

reuse with software product line techniques, Software, IEEE **26** (2009), no. 4, 47–53.

[153] R. Wolfinger, S. Reiter, D. Dhungana, P. Grunbacher, and H. Prahofer, *Supporting runtime system adaptation through product line engineering and plug-in techniques*, International Conference on Composition-Based Software Systems (ICCBSS), IEEE, 2008, pp. 21–30.

[154] Ji Zhang and Betty HC Cheng, *Model-based development of dynamically adaptive software*, Proceedings of the 28th international conference on Software engineering, ACM, 2006, pp. 371–380.

[155] Tewfik Ziadi and Jean-Marc Jezequel, *Software product line engineering with the UML: Deriving products*, Software Product Lines, pp. 557–588, Springer, 2006.

Vita

Candidate's full name: Mahdi Bashari
University attended (with dates and degrees obtained):

Sharif University of Technology, Tehran, Iran
Master of Computer Engineering (Software), 2009

Azad University of Mashhad, Mashhad, Iran
Bachelor of Computer Engineering (Software), 2006

Publications:

Journal Publications:

Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Dynamic software product line engineering: A reference framework", *International Journal of Software Engineering and Knowledge Engineering* 27 (1) (2017) 144

Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Self-Adaptation of Service Compositions through Product Line Reconfiguration", *Journal of Systems and Software* 144 (2018) 84-105

Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Automated Composition and Optimization of Services for Variability-Intensive Domains",

Journal of Systems and Software (Accepted)

Conference Presentations:

Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Automated Composition of Service Mashups Through Software Product Line Engineering." *International Conference on Software Reuse*, Springer, Cham, 2016. (Refereed Paper)

Mahdi Bashari, Ebrahim Bagheri, and Weichang Du, "Self-healing in Service Mashups Through Feature Adaptation." *International Systems and Software Product Line Conference*, ACM, 2017 (Refereed Paper)

Mahdi Bashari, Ebrahim Bagheri, "Engineering self-adaptive systems and dynamic software product line." Proceedings of the 17th International Software Product Line Conference. ACM, 2013 (Tutorial)

Mahdi Bashari, Ebrahim Bagheri, "Dynamic, Variable and Self-Adaptive Systems" *Canadian Conference on Electrical and Computer Engineering*, 2014 (Tutorial)