Self-Adaptation of Service Compositions through Product Line Reconfiguration

Mahdi Bashari\textsuperscript{a}, Ebrahim Bagheri\textsuperscript{b}, Weichang Du\textsuperscript{a}

\textsuperscript{a}Faculty of Computer Science, University of New Brunswick
\textsuperscript{b}Department of Electrical and Computer Engineering, Ryerson University

Abstract

The large number of published services has motivated the development of tools for creating customized composite services known as service compositions. While service compositions provide high agility and development flexibility, they can also pose challenges when it comes to delivering guaranteed functional and non-functional requirements. This is primarily due to the highly dynamic environment in which services operate. In this paper, we focus on maintaining functional and non-functional requirements of service compositions within high variability domains, which are often modeled through software product line techniques. One of the main characteristics of high variability application domains is that they are rich in functional features whose configuration results in an instantiated functional application. We propose adaptation mechanisms that are able to effectively maintain functional and non-functional quality requirements in service compositions derived from software product lines. Unlike many existing work, the proposed adaptation mechanism does not require explicit user-defined adaptation strategies. We adopt concepts from the software product line engineering paradigm where service compositions are viewed as a collection of features and adaptation happens through product line reconfiguration. We have practically implemented the proposed mechanism in our Magus tool suite and performed extensive experiments, which show that our work is both practical and efficient for automatically adapting service compositions once violations of functional or non-functional requirements are observed.

Keywords: Service Composition, Feature Model, Software Product Lines, Self Adaptation

1. Introduction

Many service and API sharing platforms such as ProgrammableWeb index thousands of web services which are readily available to be used by developers. The growing number of such services and their relatively easy utilization has motivated researchers to create methods and supporting tools to build composite services. Given the high variability of services and the abundance of their variations, researchers have proposed that the development of service compositions can, among other ways, be performed through Software Product Line (SPL) engineering techniques in two lifecycle phases, namely domain engineering and application engineering [1].
the domain engineering phase, a domain expert would organize the functional aspects of the domain through an SPL variability modeling mechanism such as a feature model [2]. This will include the definition of the domain functionality and the services that can implement it. In the application engineering phase, the user specifies her requirements by selecting a valid subset of the features from the variability model. On this basis, in our previous work [3], we have proposed a method to facilitate the composition of service compositions by enabling the user to express her requirements in terms of software product line features. Our method would then automatically build service compositions represented in the form of executable BPEL code based on the selected features.

In this paper, we are focusing on another aspect of service compositions, which deals directly with the practical execution of service compositions at runtime. Considering that service compositions often rely on open API and online services, their functional availability and non-functional guarantees are highly dependent on the availability and performance of the services that were used to build them. Therefore, changes or failures in the constituent services can affect the functional and non-functional guarantees of the service composition. In order to handle such situations, we suggest enabling self-adaptation. Self-adaptation is the ability of a system to react to changes in its environment to maintain service and is used in different problem domains such as multi-agent [4], cyber-physical [5, 6], and even industrial software system [7]. We propose enabling self-adaptation in order to allow a service composition to self-heal [8] in response to such failures. There are existing methods that perform self-healing on a service composition and allow service compositions to autonomously adapt at runtime to recover from failure. However, such approaches often have one or more of the following limitations:

- In some of the existing self-healing methods, it is the developer’s responsibility to design adaptation strategies. Designing adaptation strategies tend to be complicated and prone to error given the diversity of ways or circumstances under which a service or a collection of services can fail.
- Many of the existing work adopt an all-or-none approach in healing functional failures where they try to fully recover functionality and fail when full recovery is not possible.
- Some of the methods only focus on maintaining either functional or non-functional requirements. However, an effective mechanism should be able to consider both types of requirements at the same time when performing adaptation since both of these types of properties can be critical for the system’s operations.

Considering these shortcomings, we propose a Dynamic Software Product Line (DSPL) engineering-based method [9, 10], which enables a service composition to adapt automatically and recover from violations of functional and/or non-functional requirements without the need for the adaptation strategies to be explicitly defined by the experts. Dynamic software product line engineering methods use software product line models and methods at runtime to satisfy our maintain requirements [11]. We propose a method based on feature model reconfiguration techniques in software product line engineering to enable automated re-selection of features such that all critical functional and non-functional requirements are subsequently recovered after failure. The concrete contributions of our work are enumerated as follows:

- We propose 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, we propose 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.

- We have implemented the proposed method and added it to our existing service develop-
ment 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,
continuously monitor the execution of the generated service composition and adapt it as
necessary.

It should be noted that the work in this paper is an extension of our earlier work [12] and
extends it in the following ways that are not addressed earlier: (1) In this paper, we propose
a systematic approach for calculating how software product line features can impact the non-
functional properties of a service composition and how they can be continuously monitored,
estimated and maintained; (2) We present an algorithm, and formally prove its desirable charac-
teristics, for finding linearly independent feature subsets within a software product line feature
model; (3) We extend the formal representation of constraints within the context of psuedo-
boolean optimization to cover three distinct types of constraints; and (4) We introduce our fully
functional publicly available online tool suite and also provide an extensive comparative analysis
of the literature beyond what was covered earlier.

The rest of this paper is organized as follows: The background section provides a general
overview of the techniques that are used in this paper along with an introduction to the running
case study. This is followed by the details of our proposed approach, which is presented in two
subsections. In the first subsection, we discuss how features and non-functional properties are
related to each other while the adaptation mechanism is proposed and discussed in the second
subsection. In the experiments section, the design details of the experiments for evaluating the
proposed approach have been presented and our findings have been reported. In the related work,
the proposed work is compared with existing works in both self-healing software systems and
dynamic software product line engineering. The paper is finally concluded with a summary of
the findings and directions for future work.

2. Background

The work in this paper revolves around the fundamental notion of feature models where the
expected requirements of a service composition are specified by a feature model configuration,
which is used to automatically generate a service composition. On this basis, the goal of our work
is to dynamically adapt a service composition at runtime when a functional or non-functional
violation occurs. In the following, background on feature models, how service compositions can
be contextually modeled, and how automated composition of services can be performed will be
provided.
2.1. Feature Models

Feature models are among the more popular models used in the SPL community for representing the variability of the problem domain [13]. Feature models allow for hierarchical representation of features that are related to each other through structural and/or integrity constraints. The structural constraints relate features to their parents through Mandatory, Optional, Alternative, and Or relations. Integrity constraints represent dependencies between features which are not hierarchically related. Having a feature model, users can define their desired variants by selecting features from the feature model resulting in a specialized model known as feature model configuration. A feature model configuration consists of a subset of features from the feature model that respects structural and integrity constraints and can be used as the reference model for composing an application.

Figure 1-A 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 features marked with (●) in Figure 1-A denote a valid feature model configuration.

2.2. Modeling Service Compositions

In previous work [3, 14, 15], service compositions have already been represented using the Business Process Execution Language (BPEL) [16]. In this language, a service composition is defined as a process which works with a set of partner services from a service repository. In order to relate services and features together, a model known as the context model is often used [17], which contains a set of entity and fact types and instances. Context modeling [18] is widely used in context-aware systems [19] in order to capture relevant properties of the operating environment for decision-making. It is possible to define how both features and services can affect the context model. Using these information, the problem of finding a service composition for a feature model configuration can be easily reduced to a planning problem and solved using an AI planner.

Figure 1-B shows a context model for the upload image product family. Here, the context model is a triple \( (c_T, c_E, S) \) where \( c_T \) defines the entity and fact types, \( c_E \) defines entity instances, and \( S \) defines fact instances. To be more elaborate, \( c_T \) is a triple \( (\Theta, \Phi, F) \) where \( \Theta \) includes the set of all entity types, which can be used in the service family, \( \Phi \) defines the set of fact types where a fact is a relation that can be true between entities, and \( F \) specifies the type of entities that are related to each other for each fact type. Example members of \( \Theta \) are \textit{Image} and \textit{TagList}, which are the types of entities that are processed by the services of the service family. An example member of \( \Phi \) is \textit{HasTags}, which relates an image to a tag list. An example member of \( F \) is \( \text{(HasTags(Image, TagList))} \) which defines that \textit{HasTags} relates entities of type \textit{Image} to entities of type \textit{TagList}.

Using such a context model, the services in the service repository can be annotated. Each service would have two annotation sets: \( I \) and \( O \), which define the service’s inputs and outputs.
The members of these two sets would be entities with types from the context model. Each service can also be annotated with three sets of facts defined over entities from \( I \) and \( O \), namely (1) those facts that need to be true to invoke the service \( (P_I) \), (2) those facts that will be true after invocation of the service \( (Q_I) \), and (3) those facts that will become false after invocation of the service \( (R_I) \).

Figure 1-C shows some sample parts of the service repository for the example service family. It additionally shows their inputs, outputs, and their annotations using the context model. For example, the service GenerateTagMetadata has three inputs of type Image, InImageObjectList, and InImageTextList. This service has an output of type TagList. The set \( P_I \) includes two facts of type HasObject and HasText. The set \( Q_I \) has a fact of type HasMetadataTags, which specifies that the tag list entity in the output would represent the tags generated from the objects and texts in the image. The set \( R_I \) is empty for this service.

Likewise and in addition to the services, the features in the feature model can be annotated using the context model. The annotation of each feature would be a triple \( (E_f, P_f, E_f) \) where \( E_f \) is the set of entities that are needed for the realization of the service composition when this feature is included, \( P_f \) is the set of facts that should be true before the execution of the service composition with this feature, and \( E_f \) is the set of facts that will become true after executing a service composition that includes this feature. The dashed boxes in Figure 1-A show the annotations for the features in the feature model of the example service family. For instance, the set \( E_f \) for the watermark feature has an entity of type Text that is used to hold the text that needs to be watermarked on the image. The set \( P_f \) has the fact WatermarkRequested, which means that the service composition that includes this feature requires a text entity that has been requested to...
be watermarked in the image. The set $E_f$ has a fact Watermarked that denotes after executing a service composition with this feature, the image will be watermarked with the requested text.

The collection of a feature model, a service repository, context model, service annotations, and feature annotations would be referred to as a domain model for the service family. In order to express his/her requirements, the user can configure the feature model. Having the domain model and a feature model configuration, the problem of finding a sequence of service invocations which satisfy the requirements of that feature model configuration can be reduced into a planning problem that can be solved using an existing planner. Such a planner will return a sequence of service invocations, which can be converted into executable BPEL code. For example, Figure 2 shows the visualization of the BPEL process code for the feature model configuration containing features marked with (●) in Figure 1-A.

3. Approach Overview

The main objective of this paper is to develop an automated service composition reconfiguration method that can react to violations in functional or non-functional requirements. Some existing failure recovery methods introduce redundancy into the service composition [20, 21] while others find an alternate composition for satisfying the requirements through realtime re-planning [22, 23]. 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 user’s critical functional and non-functional requirements while operating in a degraded state.

We perform adaptation by automatically choosing alternative configurations that satisfy user requirements and at the same time circumvent those features that caused the failure. A feature can cause a functional failure by using a faulty or unresponsive service in its realization or can cause a violation of the non-functional requirements when the services required for its realization do not uphold the service level agreements. In order to be able to reconfigure the service composition such that it runs in a degraded state, the set of critical functional and non-functional requirements need to be specified. For this reason, we assume that the user has specified both critical as well as desired but non-critical features of the service composition. We also assume that the user would specify critical non-functional requirements. Based on this information, we perform adaptation in response to functional and/or non-functional requirement violations by re-configuring the feature model of the service composition such that the new configuration satisfies all the critical functional and non-functional requirements.

Figure 3 shows an overview of the proposed approach for service composition reconfiguration. There are two main processes in this approach: (1) feature adaptation management: a process for adapting the service composition using feature model reconfiguration in order to remedy requirement violations; and (2) feature non-functional property calculation: a process for determining the contribution of each feature to the non-functional properties of the service composition. This process is important as we need to determine how much each feature contributes to the non-functional properties of the overall composition to be able to perform adaptation at the feature level when non-functional properties are violated.

The feature adaptation management process is the core of our proposed approach and works by monitoring the functional and non-functional properties of the services and their availability. In case of a violation of a requirement, the adaptation process begins. The adaptation is formalized as a pseudo-boolean optimization problem and 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. After finding an alternate feature model configuration as a result of the pseudo-boolean optimization problem, the replacement service composition is composed and forthcoming requests are processed using the new service composition.
The feature adaptation management process relies on the possibility of being able to determine the status of the functional and non-functional properties of the service composition. While functional properties are quite straightforward to monitor, as they are either satisfied or not, it is harder to determine non-functional property values at the feature level. For this reason, we also propose a process for calculating and maintaining how much each feature contributes to the overall non-functional properties of the service composition. This process enables us to make decisions about non-functional constraints when deciding which features to include or exclude in the reconfiguration process. We use linear regression for calculating the contribution of each feature to a non-functional property in real-time.

In the following section, we provide the details of the proposed method for determining the contribution of each feature towards the non-functional properties of a service composition. On this basis and in a subsequent section, we then describe our proposed method for service composition adaptation as a response to violations of functional and non-functional requirements.

4. Impact of Features on Non-functional Properties

Although features represent functional properties of a system [24], 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 to estimate the non-functional requirements of a feature within the context of a service composition. Such 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, we propose 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 [25] 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 [26] which aggregates the values of the non-functional properties of the individual features.
In our work, while we do not assume that non-functional property values of features are constant, we make a similar assumption about how features’ non-functional properties can be aggregated. For example, Figure 4-A shows that the features in the model shown in Figure 1 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.

In this work, we employ a sum aggregation function for aggregating non-functional properties. Those non-functional properties whose aggregation function is a product, such as reliability, can be converted to summation by working with their natural logarithm. As such, the function for estimating the value of a non-functional property of a service composition can be represented as:

\[ Q(C) = \sum_{i=1}^{n} p_i c_i \]

where \( c_i \) can be zero or one based on the presence of the \( i \)-th feature \( f_i \) in the feature model configuration \( C \). Similarly, \( p_i \) is the contribution value of the \( i \)-th 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 be defined as below:

\[
y = \sum_{j \in \{1, \ldots, n\}} \beta_i x_i + \alpha
\]  

(2)

where \( x_i \) is an independent variable, \( y \) is the dependent variable, \( \beta_i \) is a regression parameter, and \( \alpha \) is the intercept. Having a dataset \( D = \{d_1, \ldots, d_m\} \) and \( d_j = \langle \bar{y}_j, x_{j1}, \ldots, x_{jn} \rangle \) where \( \bar{y}_j \) is the actual value for the dependent variable when dependent variable values are \( \langle x_{j1}, \ldots, x_{jn} \rangle \), linear regression is able to estimate the values for the \( \beta_i \) and \( \alpha \) parameters.

In Equation 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 1. Hence, linear regression can be used in order to find the values for \( p_i = \langle p_1, \ldots, p_n \rangle \) assuming that we have a set \( FC = \{(C_1, q_1), \ldots, (C_m, q_m)\} \) where pair \( (C_i, q_i) \) shows a feature model configuration and its corresponding value for a non-functional property.

We employ the Ordinary Least Squares (OLS) regression method [27], 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 \( D = \{d_1, \ldots, d_m\} \), the error for estimating the \( j^{th} \) instance of the data would be \( e_j = |y_j - \bar{y}_j| \). As such, ordinary least squares will be able find an assignment to the regression parameters such that the following is minimized:

\[
\sum_{j \in \{1, \ldots, m\}} e_j^2
\]

(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 \( P = \langle p_1, \ldots, p_n \rangle \) is found such that the sum of square of all errors is minimized.

As a requirement for OLS to converge to a single assignment to \( P = \langle p_1, \ldots, p_n \rangle \), which minimizes Equation 3, there should not be a linear dependence between values of independent variables in the training set. Having a dataset \( D = \{d_1, \ldots, d_m\} \) where \( d_j = \langle \bar{y}_j, x_{j1}, \ldots, x_{jn} \rangle \), this means for the following system of linear equations:
\[ a_1 x_{11} + a_2 x_{12} + ... + a_n x_{1n} = 0 \]  
\[ \vdots \]  
\[ a_1 x_{m1} + a_2 x_{m2} + ... + a_n x_{mn} = 0 \]  
(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 dataset of feature model configurations for a feature model may not be possible considering the fact that the relation 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 \( i^{th} \) feature is the mandatory child of the \( j^{th} \) 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 \( i^{th} \) feature is an alternative group and has two child features, which are the \( j^{th} \) and \( k^{th} \) 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, we propose 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 we have 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):

\[ \forall A \in \mathbb{R}^n - \{ < 0, ..., 0 > \} \text{ s.t. } \forall C \subseteq F \text{ s.t. } \text{Valid}(C, fm) \Rightarrow \sum_{i=1}^{n} a_i c'_i = 0 \]  
(5)

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

\[ \forall P \in \mathbb{R}^n \Rightarrow \exists P' \in \mathbb{R}^n \text{ s.t. } \forall C \subseteq F \text{ s.t. } \text{Valid}(C, fm) \Rightarrow Q(C) = Q'(C') \]  
(6)

holds.

In Equation 5, \( \text{Valid}(C, fm) \) 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 for features in \( F' \) such that for all valid configurations of the feature model, linear dependency holds for the presence of features with coefficient \( A \). Equation 6 denotes that for all assignments to contribution values \( P \), there also exists an assignment to \( P' \) such that \( Q'(C') \), which only works on a subset of features, has the same value as \( Q(C) \) for all valid feature model configurations.

In order to find the set \( F' \), Algorithm 1 has been proposed. This algorithm works on an input feature model and returns a set of features for which the conditions in Equations 5 and 6 hold.
function `FindIndependentFeatureSet(FeatureModel fm)`

1. `FC ← {}`, `F' ← fm.F`
2. `while GetNumberOfSolutions(FC) ≠ 1 do`
3. `A ← Solve(FC)`
4. `f ← FindConfiguration(A, fm)`
5. `if f ≠ {} then`
6. `FC ← FC ∪ {f}`
7. `else`
8. `R ← {f | s.t. f ∈ F' and a_i ≠ 0}`
9. `r ← SelectAMemberRandomly(R)`
10. `F' ← F' − {r}`
11. `end if`
12. `end while`
13. `return F'`

**Algorithm 1**: The algorithm for finding linearly independent feature subsets.

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 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 `FC` (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 in the dataset after exiting the loop. In Line 4 of the loop, function `Solve(FC)` is called, which finds one of these assignments 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 `FC`. Otherwise, there is a linear dependence between those features in the feature model whose corresponding `a_i` is non-zero. Therefore, those features are assigned to a set `R` in Line 9 and one of them is selected randomly and assigned to `r` in Line 10. Function `SelectAMemberRandomly` simply selects the member randomly from a set. In Line 10, the linear dependence is broken by removing a feature `r` from `F'`.

In the following, we formally prove that the set of features in `F'` after exiting the while loop will contain those features, which satisfy the requirements in Equations 5 and 6.

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

**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, we assume `FC` is the set of feature model configurations in Algorithm 1 when the algorithm exits the loop. Considering that...
A holds for all feature model configurations, it should hold for all feature model configurations in FC. Therefore, A can be considered as a solution for the system of linear equations, which is built using FC. This means that there exist at least two solutions for the system of linear equations built using FC (all zeros and A). Considering that we have used FC after the loop exited, it is in contradiction with the condition of the loop. □

**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 1 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.** We prove 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 whole 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 1. In the case of \( k = 1, F = F' \) and the same values for \( P \) can be used for \( P' \).

Now, we prove that if for iteration \( k \) there exists a \( P_k \) such that \( Q(C) = Q_k(C_i) \) 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. We assume that there are \( l \) features in \( F' \) in iteration \( k \). The fact that the loop in Algorithm 1 did not exit at iteration \( k \) means that there exists an assignment \( A = < a_1, ..., a_l > \) where \( A =< 0, ..., 0 > \) 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:

\[
c_j = \frac{\sum_{i \in [1...n], i \neq j} a_i c_i}{a_j} \quad (7)
\]

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 \( Q(C) = \sum_{i \in [1...l]} p_k c_i \) based on the induction assumption, we can replace \( c_j \) with its equivalent term in Equation 7. Consequently, we will have:

\[
Q(C) = \sum_{i \in [1...l], i \neq j} (p_k + \frac{a_i}{a_j}) c_i \quad (8)
\]

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 8 can be used in \( P_{k+1} \) such that \( Q(C) = Q_{k+1}(C_{k+1}) \). □

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-B shows reliability values for the services in the upload image service family. Having these reliability values for the services, Figure 4-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-A.

Given this approach for calculating the non-functional properties of features as well as service compositions, we are able to monitor compliance with non-functional requirements. In the next section, we will introduce our proposed approach for adapting service compositions in reaction to the violation of functional and non-functional requirements.

5. 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, we specify what kind of failures are in focus in this paper:

- **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 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 50 ms is violated.

In order to be able to react to failure, the adaptation mechanism should have context-awareness where functional and non-functional failures trigger adaptation. We assume 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 model can be seen in Figure 5, which relates to the service family in Figure 1.

As mentioned before, we assume 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 term of features compared to the failed feature model configuration. This problem can be seen as an optimization problem with the presence of certain constraints.

We need 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, we define a pseudo-boolean optimization (PBO) problem [28], which can subsequently be reduced to a boolean satisfiability (SAT) problem [29, 30] 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 we show how the problem of finding our desirable feature model configuration can be reduced to it.

**Pseudo-boolean Optimization.** Assuming an array of variables \( X = \langle x_1, \ldots, x_n \rangle \), \( x_i \in \{0, 1\} \), a pseudo-boolean optimization problem can be formally defined as:

\[
\begin{align*}
\text{Minimize:} & \quad f(x_1, \ldots, x_n) \quad f : \mathbb{B}^n \to \mathbb{R} \\
\text{Subject to:} & \quad R_1, \ldots, R_m \\
& \quad R_i : a_1 x_1 + \ldots + a_n x_n > a_i
\end{align*}
\] (9)

In this definition, \( f \) is a function whose domain is a set of variables with values that are either zero or one and their range is a real value. The possible minimum value for this function should be found with respect to a set of constraints \( R_1, \ldots, R_m \) where each constraint is an inequality relation defined over the input variables of function \( f \).

A feature model configuration \( C \) can be represented as an array of binary variables \( C = \langle c_1, \ldots, c_n \rangle \) 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 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 requirement engineering process to capture relative usefulness of a product for a user [31]. 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:

\[
\begin{align*}
\text{Minimize:} & \quad \text{loss}(C, C') \\
\text{Subject to:} & \quad S, E, I, N
\end{align*}
\] (10)

In this definition, \( \text{loss} \) 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 \( C \) is replaced with the feature model configuration represented with \( C' \). In this function, the parameter \( C \) is constant and the goal is to find an assignment to \( C' \) which minimize the loss of utility. Furthermore, \( S, E, I, N \) represent different sets of constraints which an assignment to \( C' \) 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, we formally show how the utility function and the set of constraints can be represented.

5.1. 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 [32, 33] 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 \( C \) with feature model configuration \( C' \) can be represented as a function \( \text{loss}(C, C') \) where:

\[
\text{loss}(C, C') = \sum_{i \in \{1, \ldots, n\}} c_i (1 - c'_i) U(f_i) - (1 - c_i) c'_i U(f_i) \tag{11}
\]

This equation iterates over all features in the feature model and calculates the amount of utility lost by using \( C' \). 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 \( C \) but not available in \( C' \), the utility of the unselected feature is added to the sum of the lost utility. If a feature exists in \( C' \) while it does not exist in \( C \), 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, we suggest a restricted version of the \( \text{loss}(C, C') \) function for situations when utility values of features are not available. We refer to this function as \( \text{distance}(C, C') \) that uses only the number of features which are different between the two feature model configurations. The \( \text{distance} \) function can be used instead of the \( \text{loss} \) function:

\[
\text{distance}(C, C') = \sum_{i \in \{1, \ldots, n\}} (1 - c_i) c'_i + c_i (1 - c'_i) \tag{12}
\]

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 \( c' \) are 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.
5.2. 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 $C'$ 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 $c_{\text{UploadImage}} - c_{\text{Editing}} > -1$. There exist methods for translating a feature model structure and its constraints into a set of linear constraints. In our work, we have used the translation proposed in [34].

5.3. 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 1-A will require the condition $\text{WatermarkRequested}(\text{im,wt})$ to be true before execution, which might not necessarily be true. The linear constraints 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 $P(f)$ 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 $P(f)$ 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:

$$\sum_{i \in \{1..n\} \text{ s.t. } P(f_i) \not\subseteq \text{Pre}(C)} c'_i < 1 \quad (13)$$

Assuming that $\text{Pre}(C)$ returns all of the preconditions required by the service composition realizing feature model configuration $C$ as outlined in [3], 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.

5.4. 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. We define a failed feature model configuration set, 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 compose their corresponding service composition using existing services. If this approach is not able to compose 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:

$$
\sum_{i \in \{1, \ldots, n\}} (1 - c_i') + \sum_{i \in \{1, \ldots, n\} \setminus f_i \in C} c_i' > 1 \quad (14)
$$

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.

5.5. 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 constraints 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 1. Therefore, the constraint for a non-functional property can be represented as:

$$
\sum_{i \in \{1, \ldots, n\}} p_i c_i < q \quad (15)
$$

where $p_i$ is the contribution value of the $i^{th}$ 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
In summary, based on the formal definitions provided for the loss of utility and the four constraints required in Equation 10, we are now able to automatically derive a new feature model configuration as a result of the ObjectDetection failure.

alternate feature model configuration is less than the threshold value. In the previous subsection, we proposed a regression method for estimating the contribution value of each features to a non-functional property. This method is used in order to estimate all $p_i$ values. Considering 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, we use a probabilistic approach. Assuming $e$ is the error of estimation, it can be assumed that it has a normal distribution $N(0, \sigma)$ with a mean of 0 and a standard deviation of $\sigma$. The value of $\sigma$ can be calculated using the feature model configuration in the dataset and their corresponding 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.

Figure 7: Initial, failed, and degraded state of a service composition in Figure 2 after change in the reliability of the Object Detection service.
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 model configuration that would be used for generating an alternate service composition.

As an example, let us assume that a user has specified her requirements as shown in Figure 5 where some features are critical while some other features are desired but not critical. The service composition in Figure 2 will provide both critical and desired features for this user. However, if the service ObjectDetection that extracts objects in the image fails, the service composition cannot fully recover using existing services in the service repository. The above method results in a feature model configuration with features marked with (x) in Figure 1 as the alternate feature model configuration, which clearly does not rely on the failed service. Figure 6 shows the alternate service composition built based on this configuration. In the alternate feature model configuration, the Metadata-based Tagging, Editing, and Face Blur features are removed and External Tagging is added.

For this example, let us further assume the user has also specified a critical constraint that the reliability of the service composition should be greater than or equal to 0.9. In the current context state model as shown in Figure 7, the reliability of the service composition shown in Figure 2 is 0.93, which is higher than the threshold. However, the reliability value of service composition will fall to 0.88 in the case that the reliability of the ObjectDetection service becomes 0.941. This is a violation of the non-functional constraint and therefore results in the need for an adaptation. The adaptation mechanism selects features marked with (■) in Figure 1 as the alternate feature model configuration whose corresponding service composition has a reliability of 0.906 which is higher than the user specified threshold. Figure 8 shows the alternate service composition built based on the alternate feature model configuration. In the alternate feature model configuration, the Metadata-based tagging feature has been removed and External tagging feature has been
6. Tooling Support

The ideas from this paper have been implemented in an online tool suite called Magus.Online which is publicly available and supports the whole lifecycle of service composition from design to managing its adaptation. A screen shot of this tool suite has been shown in Figure 9. This tool suite provides three tools which focus on different phases of the service composition lifecycle: Domain Design Tool, Configuration Tool, and Adaptation Management Engine & API. In the following, we provide details on the functionality of each of these tools and how they are used in the service composition lifecycle.

6.1. Domain Design Tool

This tool allows a software developer to design domain models for a service family. These domain models are base models for automated 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.

1Available at: http://magus.online/
It provides a graphical user interface for creating, editing, and serializing feature models as well as annotating features using the context model.

In order to create a service 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 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 in. 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 10 shows a screen shot 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.

6.2. Configuration Tool

A non-expert user can load an existing domain model corresponding to a service family in the 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 as deciding on the adaptation when a failure takes place. Specifically, the Configuration Tool provides the following functionalities:

- It allows loading an existing service 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 family. When a service family is loaded in the Configuration Tool, the feature model for that service 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 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. The Configuration Tool investigates the possibility of satisfying those constraints and notifies the user if those constraints can be satisfied using current services. Figure 11 shows the interface for the 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.

6.3. Adaptation Management Engine & API

The user can integrate a BPEL execution environment to the Adaptation Management Engine using its REST API. This allows for the monitoring of the health and performance of the generated composition as well as enabling adaptation 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.
Figure 12: The Adaptation Management Engine interface for monitoring current state 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. 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 is sent back to the execution environment. The Adaptation Management Engine also updates the online console with updated information of the service composition. Figure 12 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.

7. Experiments

Our proposed approach has been implemented using the FF planner [35] for AI planning and NaPS solver [36] for pseudo-boolean optimization. Using this implementation, we performed four different sets of experiments 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

---

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

7.1. Effectiveness of Adaptation in Addressing Functional Failures

In this first set of experiments, we focus on answering two Research Questions (RQ):

**RQ 1.1** - How does the execution time for performing an adaptation change as the possible feature model configuration size increases?

**RQ 1.2** - 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 we investigate the applicability of the proposed method in terms of adaptation time for larger feature models. 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. We used the number of valid feature model configurations 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, and a possible 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: We used the Betty feature model generator [37] to generate the feature models with 30 features which is the average number of features in the SPLOT repository and with maximum branching factor of 10. Four groups consisting of 25 percent of the features each were allocated to Or group, Alternative group, Optional and Mandatory categories. According to the survey that is performed in [38], 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 \( N(2, 1) \), \( N(0.2, 0.8) \), and \( N(1, 1) \) as the number of entities, preconditions, and effects, respectively. \( N(\mu, \sigma) \) is a normal distribution with a mean of \( \mu \) and a standard deviation of \( \sigma \). These values were calculated based on the case studies that we 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, \( N(1, 1) \), and \( N(2, 1) \) for the number of preconditions and effects.

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 compositions whose effects have the largest intersection with the feature model configuration 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.

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

\(^{3}\)http://www.splot-research.org/

\(^{4}\)All generated dataset are available for download at: http://magus.online/experiments/1

\(^{5}\)The code for the tool and the service generator is available at https://github.com/matba/magus
each service family. Having five groups and ten service families in each group, this activity was repeated 5,000 times.

Figure 13 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 14 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. 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 compositions 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.

7.2. Robustness of Adaptation towards Functional Failure

In this set of experiments, we focus 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, we are looking to answer the following research questions:

**RQ 2.1** - How successful is the adaptation mechanism in addressing a service failure as the number of failed services increases?

**RQ 2.2** - How extensive 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, we created 10 service families with parameters used in the first experiment with 800 possible configurations and service repository size of 150.

In this experiment, for the number of failures between one to 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 15 shows the distribution of successful versus failed adaptation as service failure occurs for 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 16 shows the distribution of the distance between the alternate feature model configuration and the failed feature model configuration for different number 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.

7.3. Effectiveness of Regression For Estimating Non-functional Properties

In this set of experiments, we focus on investigating if using a linear function is an effective way to accurately estimate the non-functional properties of a service composition. More specifically, we are trying to answer the two following research questions:
Figure 18: Root Mean Squared Error of the estimation method in terms of number of training feature model configuration.

**RQ 3.1 -** How fit is a linear function for estimating the values of non-functional properties in a service composition?

**RQ 3.2 -** 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, we used coefficient of determination or R squared ($R^2$), which is a metric for determining how good 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 zero means it cannot model the variance.

In order to perform the first experiment, we focused 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 $N(200, 50)$. 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 17 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, 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 dependant 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 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, we used root-mean-square error, 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 18 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 estimation model overfits data when the dataset is small. It can also be seen 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 less 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 less than 200 feature model configurations.
7.4. Effectiveness of Feature Adaptation for Addressing Non-functional Violations

In this set of experiments, we are investigating if the proposed approach can effectively address non-functional violations of service composition. More specifically, we are trying to answer two research questions:

RQ 4.1 - How successful is the adaptation mechanism in addressing non-functional failures for feature models with different sizes?

RQ 4.2 - How extensive 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 answering RQ 1.1 were annotated randomly with response times with the distribution of $N(200, 50)$. 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 did no longer hold. 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 19 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 20 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 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.

8. Related Work

This section first compares our work with existing self-healing approaches in service-based systems in general and then compares our work with existing works which use feature models as the main model for managing adaptation.

8.1. Self-healing 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 [42, 43, 44]. For example in [43], 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, we are focusing 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 1 shows some of the more recent works in this area (≥2013) compared to the proposed method. In the following, we go through the different aspects of self-healing mechanisms covered in Table 1 and discuss similarities and differences between existing methods and our proposed method.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Adaptation Cause</th>
<th>Adaptation Goal</th>
<th>Adaptation Realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hristoskova et al. [23]</td>
<td>Service Failure –</td>
<td>Recovery –</td>
<td>Substitution / Re-</td>
</tr>
<tr>
<td></td>
<td>Functional</td>
<td>Non-functional</td>
<td>organization</td>
</tr>
<tr>
<td></td>
<td>Functional</td>
<td>Non-functional</td>
<td>Genetic Algorithms</td>
</tr>
<tr>
<td>Angarita et al. [40]</td>
<td>Service Failure –</td>
<td>Recovery –</td>
<td>Retry/Substitution</td>
</tr>
<tr>
<td></td>
<td>Constraint Violation</td>
<td>Recovery –</td>
<td>Rule-based</td>
</tr>
<tr>
<td>Nallur et al. [41]</td>
<td></td>
<td>Recovery –</td>
<td>Substitution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Non-functional</td>
<td>Market-based Heuristics</td>
</tr>
<tr>
<td>Carzaniga et al. [21]</td>
<td>Service Failure –</td>
<td>Recovery –</td>
<td>Reorganization</td>
</tr>
<tr>
<td></td>
<td>Constraint Violation</td>
<td>Non-functional</td>
<td>Rule-based</td>
</tr>
<tr>
<td>Our Approach</td>
<td>Service Failure –</td>
<td>Recovery –</td>
<td>Reduction</td>
</tr>
<tr>
<td></td>
<td>Constraint Violation</td>
<td>Recovery –</td>
<td>Pseudo-boolean</td>
</tr>
<tr>
<td></td>
<td>Mitigation</td>
<td>Reorganization</td>
<td>Optimization</td>
</tr>
</tbody>
</table>

Table 1: Comparison of recent approaches for self-healing in service compositions.
Most of the work in the literature have 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. [23] 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 QoS for service composition is not in the acceptable range to restore it. As an example, Nallur et al. [41] 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 QoS constraints. Although providing both functional and non-functional requirements can play a critical role in the usability of the system, most existing methods in the literature are designed such that they can only respond to one of these failure types. In the proposed method, we address this shortcoming by proposing a decision-making mechanism which considers both functional and non-functional requirements at the same time in planning for addressing a failure.

Although a system usually 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 while deciding on adaptation, we refer to the work by Angarita et al. [40] that 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 usually 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. [39] 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, we used feature models 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 represent 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.

Strategies for realizing adaptation can be classified into three categories [22]: 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 [45, 46]. 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 [21, 22, 47]. One example of such methods is the approach proposed by [20] 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 usually use adaptation rules [48, 49], AI planning [22, 23], and other methods [50, 51] to recover from failure. For example, the failure recovery method proposed in [22] looks for a service with the same functionality when a failure occurs; in cases that such 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.

8.2. Using Feature Models for Managing Adaptation

Feature models have been effectively used in many of the dynamic software product line approaches for managing variability of the adaptive software [52]. These methods mostly work by monitoring execution context where certain changes in the context trigger a change in the current feature model configuration. Consequently, the change in the feature model configuration is reflected in the running service composition [53, 54]. For example in [54], a method is used for developing self-healing smart homes where devices are likely to fail. In this method, rules for changes in the feature model have been defined to address failures. For example, if the service for the siren device fails, it selects a feature which uses house lights for alarming. The changes in the feature model configuration are reflected on the running system by enabling/disabling its corresponding services and connectors. In the Refresh approach [55] for self-healing service composition, feature models are used as the model for capturing different ways for realizing a service composition. When a service failure occurs, the feature corresponding to the service is marked as unselectable and the problem of finding an alternate feature model configuration is represented as a Constraint Satisfaction Problem (CSP) and solved using existing solvers. When an alternate feature model configuration is found, the system shuts down those components corresponding to the unselected features and launches those corresponding to the selected features using a method called micro-booting. The work in [56] proposes using multi-objective evolutionary algorithms to select features from mobile applications’ feature model in different contexts in order to optimize its performance in respect to multiple non-functional requirements. In general, our work is similar to this work in the sense that it uses feature models as the main artifact for representing variability but there are three aspects that distinguish our work: first, none of the existing methods work on the relation of a feature and non-functional properties nor take those requirements into account. Second, our work does not require any predefined rules for adaptation, which have been shown to be prone to rule inconsistencies [57]. Third, existing approaches assume that there is a direct relationship between features and services, which is not always true in practice [58]. In our work, we do not assume any direct relationship between features and services, instead, we use planning for finding either an atomic or composite service that can collectively create the service composition.

9. Concluding Remarks

This paper proposes that 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, we proposed a self-healing service composition reconfiguration method. 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 constraint 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 a regression method and the second process works by reducing the problem of selecting features to pseudo-boolean optimization. Our experiments showed that the first process can effectively estimate the contribution value of features and the second process can recover a service composition in more than 70% and 35% of the time for functional and non-functional failures, respectively. This results show that the proposed approach can effectively improve the robustness of service compositions in response to both functional and non-functional failures.

References

URL http://doi.acm.org/10.1145/3106195.3106215
35


[16] OASIS, Web services business process execution language version 2.0. URL http://docs.oasis-open.org/wsbadl/2.0/wsbadl-specification-draft.html


URL https://doi.org/10.1007/978-3-642-25535-9_29


URL https://doi.org/10.1007/978-3-642-25535-9_29


[54] R. Angarita, M. Rukoz, Y. Cardinale, Modeling dynamic recovery strategy for composite web services execution,


