The Journal of Systems & Software 144 (2018) 84–105

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Self-adaptation of service compositions through product line reconfiguration

Mahdi Bashari[a], Ebrahim Bagheri[*,b], Weichang Du[a]

[a] Faculty of Computer Science, University of New Brunswick ,Canada
[b] Department of Electrical and Computer Engineering, Ryerson University, Canada

ARTICLE INFO

Keywords:
Service composition
Feature model
Software product lines
Self adaptation

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 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.

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 (Lemos et al., 2016). 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* (Pohl et al., 2005). In 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* (Lee et al., 2002). 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 (Bashari et al., 2016), we have proposed a method to facilitate the composition of services 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 composition, 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-adaption. 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 multi-agent (Jiao and Sun, 2016), cyber-physical (Gerostathopoulos et al., 2016; Chen et al., 2017), and even industrial software systems (CÃ!‘mara et al., 2016). We propose enabling self-adaption in order to allow a service composition to *self-heal* (Kephart and Chess, 2003) in response to such failures. Although researchers have been working on various methods for enabling self-adaptation in the BPEL domain, our work is still timely considering that BPEL is currently extensively used for defining business processes in industry and many recent work are focused on enabling adaption for BPEL processes (ai Sun et al., 2018; Alfrez and Pelechano, 2017; Margaris et al., 2016; 2015). The proposed work is also relevant considering that it addresses limitations in existing work by enabling service compositions to *autonomously* adapt at runtime to recover from failure. More specifically, our work addresses the

* Corresponding author.
E-mail addresses: mbashari@unb.ca (M. Bashari), bagheri@ryerson.ca (E. Bagheri), wdu@unb.ca (W. Du).

https://doi.org/10.1016/j.jss.2018.05.069
Received 25 October 2017; Received in revised form 2 April 2018; Accepted 31 May 2018
Available online 06 June 2018
0164-1212/

following challenges in the state of the art:

- In some of the existing self-healing methods (Cetina et al., 2009; Subramanian et al., 2008), 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 (Lemos et al., 2013).
- 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 (Hristoskova et al., 2013; Angarita et al., 2016).
- Some of the methods only focus on maintaining either functional or non-functional requirements (Hristoskova et al., 2013; Canfora et al., 2008). 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 (Tan et al., 2014; Angarita et al., 2016).

A real-life example which can show the shortcomings of existing approaches is a flight booking website. Such websites often provide additional features such as hotel and car rental services, which provide discounted price based on the destination and the selected airline. These additional features are not critical although being desirable. These features are typically implemented as a holistic process, which will break if the services realizing the desired but not critical features do not perform properly. To address such scenarios, existing self-adaptive approaches focus either on finding alternate ways to realize these features, which is not always possible or leave the task of devising appropriate mitigation strategies to the developer, which could be complex and labour-intensive. In our work, we address failure by removing the feature(s) which caused the failure if they are non-critical (such as hotel booking or car rental) while ensuring integrity of the whole process. As a further example, it might be expected that the ticketing process finishes in less than a specific amount of time in order to guarantee customer satisfaction. Existing approaches focus on optimizing the time-to-completion of the process through alternative services or processes which may or may not result in meeting the specified constraint. In our proposed approach, we satisfy such constraints by removing the minimal set of non-critical features which guarantee the time-to-completion constraint of the system being satisfied (e.g., automatically removing hotel booking and/or car rental when they take much longer than expected).

More concretely, we propose a *Dynamic Software Product Line (DSPL)* engineering-based method (Hallsteinsen et al., 2008; Bosch and Capilla, 2012), 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 or maintain requirements (Montalvillo and DÃaz, 2016). 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 development suite, called *Magus*. The tool suite allows for the specification of the product line representation of the domain, as well as the modelling 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 (Bashari et al., 2017b) 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 characteristics, for finding linearly independent feature subsets within a software product line feature model. Linearly independent feature sets are important since a non-functional property of a service composition can be estimated as a unique linear equation over the availability of one of the features in each of these subsets; (3) We extend the formal representation of constraints within the context of psuedo-boolean optimization to cover three distinct types of constraints, namely the constraints defined over non-functional properties of service composition by the user, the constraints defined over combination of features which describe a valid service composition, and the assumptions made over input data by the failed service; 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: Section 2 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 Section 3, which provides an overview of the proposed approach. The details of this approach is presented in two sections. In the first section (Section 4), we discuss how features and non-functional properties are related to each other while the adaptation mechanism is proposed and discussed in a subsequent section (Section 5). In Section 6 the architecture used to implement the proposed approach has been discussed. This section is followed by Section 7 in which we go through the functionality provided by our tool suite. In Section 8, the design details of the experiments for evaluating the proposed approach have been presented and our findings have been reported. In Section 9, 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 discussion of lessons learnt, threats to validity as well as a summary of the findings and directions for future work.

2. Background

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 (Benavides et al., 2010). 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. Mandatory children of a feature must be selected when their

parent is selected while optional features may or may not be selected when their parent is selected. One or more features that are grouped with an *Or* relation can be selected when their parent is selected while exactly one feature from a group of *Alternative* features can be selected when their parent is selected. 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.

Fig. 1shows a feature model for a family of services, which process and upload an incoming image on a website. An application instance in this family needs to have the mandatory feature of storage and can provide optional features of tagging, filtering, and editing. The tagging feature processes an image and finds keywords that describe the image to be used in different operations such as image search. There are two types of tagging in this product line, namely metadata-based and external. In meta-data tagging, information about objects and text in the image is used to create the tags for the image. In the external tagging feature, an external service is used to create the tags for the image. The filtering feature provides mechanisms for detecting nudity or profanity within the image. Similar to the filtering feature, the editing feature provides watermark or face blur capabilities. The watermark feature watermarks an arbitrary text on an image and the face blur feature obscures faces in the image. The marks on the bottom right corner of the features in Fig. 1 are not part of the feature model notation and are used here to refer to different feature model configurations. For example, the features marked with (●) in Fig. 1 show the set of features that belong to a valid feature model configuration.

2.2. Modeling service compositions

In previous work (Bashari et al., 2016; Cheng et al., 2017; Im et al., 2013), service compositions have already been represented using the Business Process Execution Language (BPEL) (OASIS, 0000). 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 (Wang et al., 2004), which contains a set of entity and fact types and instances. Context modeling (Perera et al., 2014) is widely used in context-aware systems (Alegre et al., 2016) 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.

Fig. 2shows 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 (Θ, Φ, $\mathscr{F}$) where Θ includes the set of all entity types, which can be used in the service family, Φ defines the set of fact types where a fact is a relation that can be true between entities, and $\mathscr{F}$ specifies the type of entities that are related to each other for each fact type. Example members of Θ are *Image* and *TagList*, which are the types of entities that are processed by the services of the service family. An example member of Φ is *HasTags*, which relates an image to a tag list. An example member of $\mathscr{F}$ is (*HasTags*(*Image, TagList*)) which defines that *HasTags* relates entities of type *Image* to entities of type *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 ($\mathscr{P}_I$), (2) those facts that will be true after invocation of the service ($\mathscr{Q}_I$), and (3) those facts that will become false after invocation of the service ($\mathscr{R}_I$). Fig. 3 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 $\mathscr{P}_I$ includes two facts of type *HasObject* and *HasText*. The set $\mathscr{Q}_I$ 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 $\mathscr{R}_I$ 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, $\mathscr{P}_f$, $\mathscr{E}_f$) where E_f is the set of entities that are needed for the realization of the service composition when this feature is included, $\mathscr{P}_f$ is the set of facts that must be true before the execution of the service composition with this feature, and $\mathscr{E}_f$ is the set of facts that will become true after executing a service composition that includes this feature. The dashed boxes in Fig. 1 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 $\mathscr{P}_f$ 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 $\mathscr{E}_f$ 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 union of $\mathscr{P}_f$ for the selected features can be considered to be the initial state of the planning problem and union of $\mathscr{E}_f$ for the selected feature can be considered as the goal state of the planning problem. Considering the invocation of different services to be actions, the problem of finding a sequence of service invocations which satisfy the requirements of the feature model configuration can be reduced into a planning problem represented using PDDL which is supported by many off-the-shelf planners. Such a planner will return a sequence of service invocations which can move us from the initial state to the goal state. This sequence of service invocations is converted to an invocation dependency graph using an optimization algorithm which removes precedence relations between service invocations. This graph is then converted into executable BPEL code.

For example, Fig. 4 shows the visualization of the BPEL process for the feature model configuration containing features marked with (●) in Fig. 1. Service invocations in this BPEL process have been organized using three type of structures: flow, sequence, and links. The invocations in the flow structure can be invoked in any order while the invocations in a sequence structure should be executed in order. The link structure is used when there is precedence relation between two invocations which cannot be represented using a sequence. In this example, after an image is received, two sequences of activities are performed in parallel in a flow structure. The first sequence calls a service for detecting the objects in the image, then calls a service for filtering objects based on the type only keeping the list of face objects in the image, and blurring faces by calling a face blur service and finally uploading the image on an image upload server. The second sequence calls text extraction service and using the text extracted from the image, it calls two services in parallel in a flow structure. One detects profanity in the image and the other generates meta-data based on the text and objects detected in the image. The are two precedence relationships

between the activities in the main flow which are enforced with links. First, the invocation of the meta-data generation service should be done after the invocation of the object detection service since it is dependent on the data returned by the first service. Second, the invocation of the text-extraction service should happen before blurring the image since it may affect the text extraction service performance. After performing this sequence, the service composition will return the generated tag list, detected profanity and the url of the image that has been uploaded.

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 (Christos et al., 2008; Carzaniga et al., 2015) while others find an alternate composition for satisfying the requirements through realtime replanning (Huang et al., 2015; Hristoskova et al., 2013). 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 an alternative feature set that satisfies user's critical 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, which can also cause a violation of the non-functional constraints.

Fig. 5provides an overview of the approach that is proposed for recovering from such failures. The only input that the user should provide to enable adaptation is the set of critical features and non-functional constraints. Adaptation uses these inputs in addition to the domain model and the feature model configuration, which specifies desirable features, to decide about the best adaptation strategy in response to a failure. The top box in Fig. 5 shows the activities performed to enable self-healing for a service composition. The process is an iterative process (as shown by a dashed line going from the last activity in the process to the first one) which starts by an activity which monitors the functional and non-functional requirements specified by the user and possibly detects the occurrence of any failures. In case of a failure, the process switches to those activities which are responsible for finding an alternative service composition which replaces the failed service composition. We briefly discuss how the monitoring activity can be realized in Section 6; however, we assume mechanisms for measuring service availability and quality of service metrics are already available and as such methods for measuring them would be outside the scope of this paper.

In our proposed approach, the first activity that is performed is estimating the effect of features on non-functional properties of the service composition. This step is required since the adaptation mechanism decides about features when performing adaptation and needs to know how selecting a feature affects non-functional properties of the service composition. The red box in the bottom of Fig. 5 shows the steps that should be taken in this activity. The first step of this activity is selecting independent feature sets. An independent feature set is a set of features in the feature model where the selection status of all features in the service composition can be determined by knowing the presence of such features in a feature model and there is no feature in this feature set whose presence can be determined by knowing the presence of other features in this set using a linear relation. In the next step, we generate a dataset of feature model configurations and their corresponding service compositions. Based on this dataset, we estimate the extent to which each of the features in the independent feature set contributes to non-functional properties of the final service composition. It can be shown that knowing the contribution value of features is enough for estimating the non-functional properties of service compositions and also linear regression converges to a single value for contribution values of features. The first two steps of this activities only need to be executed once and the next two steps need to be executed in each iteration of the adaptation process. These steps have been discussed in detail in Section 4.

The next activity in the adaptation process is to find an alternative feature set to replace the failed feature set by reducing this problem into a *pseudo-boolean optimization* problem that can be solved using an optimization solver. A solution to this optimization problem delivers critical features and satisfies non-functional constraints, if and when possible, while minimizing loss of desired features. The details of this step of the process is discussed in detail in Section 5. After finding an alternative feature model configuration as a result of the pseudo-boolean optimization problem, the replacement service composition is composed. For composing a service composition based on a feature model composition, we use a method which uses AI planning to generate the alternative. This method has been discussed in detail in Bashari et al. (2016). After composing the alternative service composition, it will be deployed in the execution environment and new requests are redirected to it. We discuss how such mechanism can be implemented in Section 6.

4. Impact of features on non-functional properties

Although features represent functional properties of a system (Bosch, 2000), 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 Benavides et al. (2005) 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* (Mohabbati et al., 2011) 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, Fig. 6-A shows that the features in the model shown in Fig. 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 the product operator, such as reliability, can be converted to summation by working with their natural logarithm. It should be noted that the relation between features and non-functional properties might be more complex than the relation assumed in this paper. However, we suggest that this relation can be estimated using a linear function over feature presence with sufficient performance for our purpose. As such, the function for estimating the value of a non-functional property of a service composition can be represented as:

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

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

Considering a linear relationship between the presence of the features and the value of a non-functional property, linear regression is a suitable option for finding the p_i values for the $Q(C)$ function. The linear regression approach is used for modeling the relationship between a dependent variable and a set of independent variables affecting the value of that dependent variable having a linear contribution. It allows estimating the contribution value for each of the independent variables to the value of the dependent variable using a limited number of dependent variable and independent variable set values. The relation between independent variables and the dependent variable can defined as below:

$$y = \sum_{i\in\{1,..,n\}} \beta_i x_i + \alpha \tag{2}$$

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

In Eq. (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 Eq. (1). Hence, linear regression can be used in order to find $P = <p_1, ...,p_n>$ which represents contribution values of all features where p_i is the contribution of the ith feature. This can be done assuming that we have a set $D = \{(C_1, q_1), ...,(C_m, q_m)\}$ where pair (C_i, q_i) shows a feature model configuration and its corresponding value for a non-functional property.

We employ the *Ordinary Least Squares (OLS)* regression method (Späth, 2014), which is widely used for estimating regression parameters. Assuming that there is a dataset, the ordinary least squares method is able to find an assignment to the regression parameters, which minimizes the squared errors in that dataset. Having dataset $D = \{d_1, ...,d_m\}$, the error for estimating the jth instance of the data would be $e_j = |y_j - \overline{y_j}|$. As such, ordinary least squares will be able find an assignment to the regression parameters such that the following is minimized:

$$\sum_{j\in\{1,..,m\}} e_j^2 \tag{3}$$

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

As a requirement for OLS to converge to a single assignment to $P = <p_1, ...,p_n>$, which minimizes Eq. 3, there should not be a linear dependence between values of independent variables in the training set. Having a dataset $D = \{d_1, ...,d_m\}$ where $d_j = <\overline{y_j}, x_{j1}, ...,x_{jn}>$, this means for the following system of linear equations:

$$\begin{aligned} &a_1x_{11} + a_2x_{12} + \cdots + a_nx_{1n} = 0 \\ &\quad\vdots \\ &a_1x_{m1} + a_2x_{m2} + \cdots + a_nx_{mn} = 0 \end{aligned} \tag{4}$$

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

There can be more complex situations where feature model constraints do not allow for creating a dataset in which a linear dependence does not hold. In order to address this issue, 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):

$$\begin{aligned} \neg\, \exists\, A \in \mathbb{R}^{n'} - \{<0, ...,0>\}\ s.\ t.\ \ \forall\, C \subseteq F\ s.\ t.\ \ Valid(C, fm) \\ \Rightarrow a_1c'_1 + \cdots + a_{n'}c'_{n'} = 0 \end{aligned} \tag{5}$$

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

$$\begin{aligned} \forall\, P \in \mathbb{R}^n \Rightarrow \exists\, P' \in \mathbb{R}^{n'}\ s.\ t.\ \ \forall\, C \subseteq F\ s.\ t.\ \ Valid(C, fm) \Rightarrow Q(C) \\ = Q'(C') \end{aligned} \tag{6}$$

holds.

In Eq. (5), A represents possible coefficient values for features in feature subset F' and $Valid(C, fm)$ examines if a configuration C is valid in terms of the constraints represented in *fm*. Therefore, the equation expresses that there is no assignment to $A = <a_1, ...,a_{n'}>$ except all zeros such that for all valid configurations of the feature model, linear dependency holds for this subset of features. Eq. (6) denotes that for all

```
1: function FindIndependentFeatureSet(FeatureModel fm)
2: D ← {}, F′ ← fm.F
3: while GetNumberOfSolutions(D) ≠ 1 do
4:    A ← Solve(D)
5:    C ← FindConfiguration(A, fm)
6:    if C ≠ {} then
7:       D ← D ∪ {C}
8:    else
9:       F_R ← {f_i s.t. f_i ∈ F′ and a_i ≠ 0}
10:      f ← SelectAMemberRandomly(F_R)
11:      F′ ← F′ − {f}
12:   end if
13: end while
14: return F′
```

Algorithm 1. The algorithm for finding a subset of features F' for which linearly dependency does not hold.

assignments to contribution values P, there also exists an assignment to P' such that $Q'(C')$, which has been defined on a the subset of features F', has the same value as $Q(C)$ for all valid feature model configurations.

In order to find the set F', Algorithm 1 has been proposed. This algorithm works on an input feature model and returns a set of features for which the conditions in Eqs. (5) and (6) hold. The function works by incrementally building a dataset for which no linear dependency holds. In the process of building this dataset, the linear dependencies in the current dataset are considered as candidates for the linear dependencies that holds for all configurations of the dataset. If this is not the case, a feature model configuration which does not hold the candidate linear dependency is added to the dataset. Otherwise, one of the features involved in the linear dependency is randomly removed from the feature set and the process continues.

The algorithm starts by assigning an empty set of feature model configurations as the dataset D where linear dependency does not hold (Line 2) and continues by incrementally adding new instances of feature model configurations to it in the while loop. Conversely, the set of all features F is assigned to the result feature set F' and features are incrementally removed from it in the while loop. This while loop continues until there is only one solution for the system of linear equations which is built using the existing feature model configurations in the dataset. Therefore, there will be no linear dependence between the selected features F' in the dataset after exiting the loop. In Line 4 of the loop, function *Solve*(D) is called, which finds one of these 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 D. Otherwise, there is a linear dependence between those features whose corresponding a_i in A is non-zero. Therefore, those features are assigned to a set F_R in Line 9. Therefore, F_R will include those features in F' which have linear dependency. This linear dependency is broken by selecting a feature f randomly in Line 10 and removing it from F' in Line 11. Function *SelectAMemberRandomly* simply selects the member randomly from a set.

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

Lemma 1. *(No Linear Dependency) There exists no linear dependency between the set of features F' 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 D 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 D. Therefore, A can be considered as a solution for the system of linear equations, which is built using D. This means that there exist at least two solutions for the system of linear equations built using D (all zeros and A). Considering that we have used D 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 while loop. Therefore, there exists such P' for the result set F'. This is proven by mathematical induction over the validity of the lemma in the k^{th} iteration of the loop in Algorithm 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 $Q(C) = Q_k(C_k)$ for all valid configurations, there also exists a P_{k+1} for which $Q(C) = Q_{k+1}(C_{k+1})$ holds for all valid configurations. 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 $A \neq <0, ...,0>$ and the equation $a_1c_1 + \cdots + a_lc_l = 0$ holds for all valid configurations. Assuming that in iteration k, the j^{th} feature has been removed to break a linear dependency, since the j^{th} feature has been removed, the value of a_j cannot be zero. Therefore, it can be said that:

$$c_j = \frac{\sum_{i\in\{1,..,l\},i\neq j} a_i c_i}{a_j} \tag{7}$$

for all valid feature model configurations using equation $a_1c_1 + \cdots + a_lc_l = 0$. Having an assignment to P_k such that $Q(C) = \sum_{i\in\{1,..,l\}} p_{ki} c_i$ based on the induction assumption, we can replace c_j with its equivalent term in Eq. (7). Consequently, we will have:

$$Q(C) = \sum_{i\in\{1,..,l\},i\neq j} \left(p_{ki} + \frac{a_i}{a_j}\right) c_i \tag{8}$$

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

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

The process of creating and updating this estimation function works as follows: m feature model configurations are randomly selected and their corresponding service compositions are built once. Having implemented service compositions for all those feature model configurations, the values for the non-functional property for all m feature model configurations are calculated using the individual services' non-functional property values. Using the set of feature model configurations and their corresponding non-functional property values, function $Q(C)$ is estimated using the ordinary least squares method. For example, Fig. 6-B shows reliability values for the services in the upload image service family. Having these reliability values for the services, Fig. 6-C shows an example dataset which can be used for estimating the contribution values for features using linear regression. This dataset is built by randomly selecting a feature model configuration and calculating the reliability of the corresponding service compositions by building the

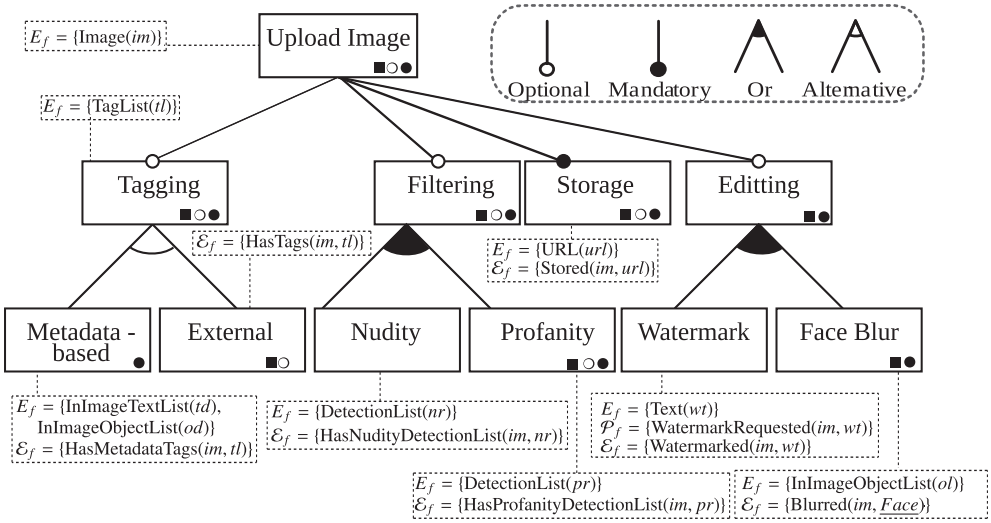

Fig. 1. An annotated feature model for the upload image service family.

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 Fig. 6-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:

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

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

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

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

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

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

S = {}

Fig. 2. The context model for the upload image service family .

Services

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

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

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

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

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

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

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

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

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

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

Fig. 3. Part of the service repository and annotations for the upload image service family.

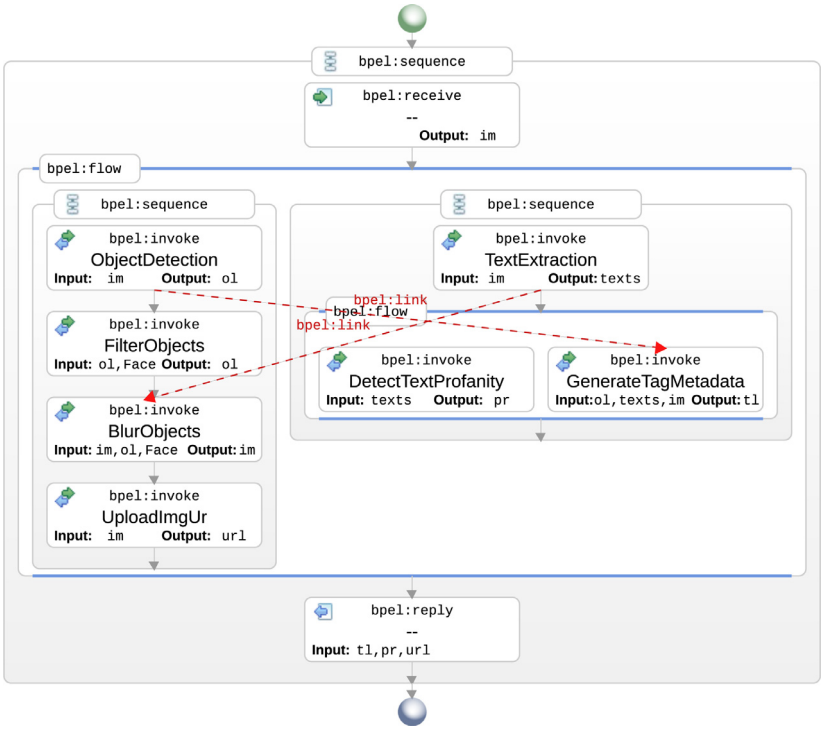

Fig. 4. BPEL process visualization for a possible service composition satisfying requirements of the feature model configuration in Fig. 1.

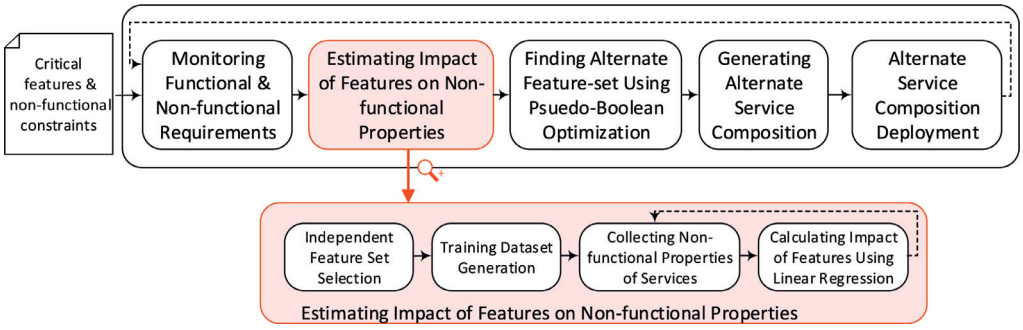

Fig. 5. General overview of the proposed approach for self-healing from functional and non-functional failure.

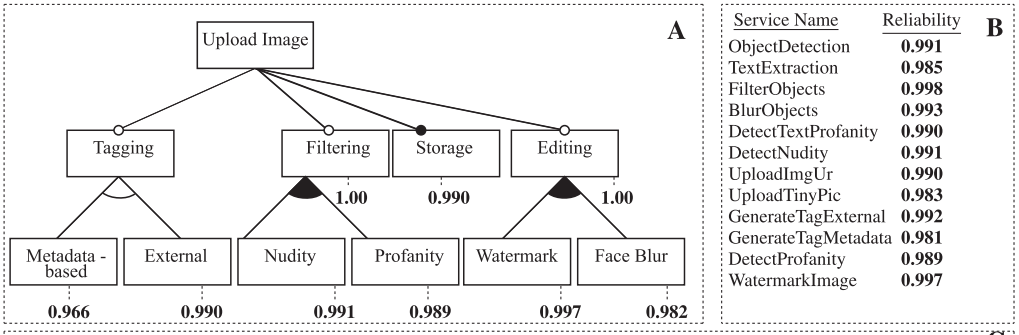

Fig. 6. A. Feature model annotated with contribution value of each feature to the reliability of the service composition. **B.** Reliability of the individual services in the service family **C.** An example of a dataset used for estimating the contribution value of each feature to the reliability of the service composition.

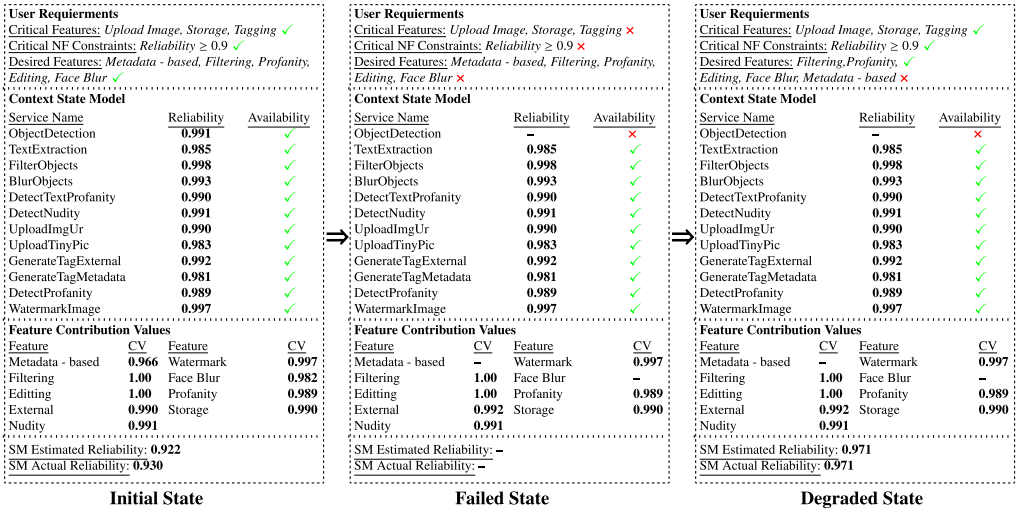

Fig. 7. Initial, failed, and degraded state of the service composition in Fig. 4 after failure of the *Object Detection* service.

- *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 Fig. 7, which relates to the service family in Fig. 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.

User's requirements outlined in the Initial State of Fig. 7 shows an example of the input a user needs to specify for service composition. It specifies that Upload Image, Storage, and Tagging are critical and Metadata-based Tagging, Filtering, Profanity, Editting, and Face Blur are desired. It is also critical for the user that the composed service has a reliability higher than 0.9. Based on the availability and reliability measures of the services reported in the context state model of the initial state, the features shown in Fig. 4 can be selected to offer a reliability value of 0.93, which satisfies the constraint specified by the user. In the following, we describe a functional and non-functional failure and use them as running examples in the reset of this section. An example of functional failure is when the Object Detection service in Fig. 4 becomes unavailable. The context state model and the satisfaction of user requirements in this situation has been shown as failed state in Fig. 7. In this state none of the user's requirements is being satisfied. An example of non-functional failure would be when the reliability of Object Detection service decreases from 0.991 to 0.941 as shown in the Failed state of Fig. 9. Such reduction in reliability will result in reduction of the overall reliability of the service composition to 0.883 which is less that 0.9 that is critical for the user.

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 (Boros and Hammer, 2002), which can subsequently be reduced to a boolean satisfiability (SAT) problem (Een and Sorensson, 2006; Manquinho et al., 2009) 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 $X = <x_1, \ldots, x_n>$, $x_i \in \{0, 1\}$, a pseudo-boolean optimization problem can be formally defined as:

$$\begin{aligned} &\textbf{Minimize}: f(x_1, \ldots, x_n) \quad f: \mathbb{B}^n \rightarrow \mathbb{R} \\ &\textbf{Subject to}: R_1, \ldots, R_m \quad R_i: a_{i1}x_1 + \ldots + a_{in}x_n > a_i \end{aligned} \tag{9}$$

In this definition, f is a function whose domain is a set of variables $x_1, \ldots, x_n$ with values that are in the set $\mathbb{B} = \{0, 1\}$ and its range is $\mathbb{R}$ which is the set of real numbers. The possible minimum value for this function should be found with respect to a set of constraints $R_1, \ldots, R_m$ where each constraint is an inequality relation defined over the input variables of function f. To be more specific, R_i is an inequality defined as the weighted sum of variables $x_1, \ldots, x_n$ where a_{ij} is the weight of x_i and the value of this weighted sum is greater than a threshold value a_i.

A feature model configuration C can be represented as an array of binary variables $\mathscr{C} = <c_1, \ldots, c_n>$ where c_i would be one if $f_i \in C$ and zero

otherwise. Using this representation of the feature model configuration, the difference between two feature model configurations can be represented as a pseudo-boolean function. Before defining such function, it is required to specify how the difference between the alternate feature model configuration and the current one can be quantified. One possible way to quantify this difference is using the loss in the utility for the user when an alternate feature model configuration is used. The utility is a numerical value used in the requirement engineering process to capture relative usefulness of a product for a user (Berger, 2013). Therefore, loss of utility as a result of feature model reconfiguration can be used as a measure that needs to be minimized. The problem of finding an alternate feature model configuration C' with the minimum loss of utility from C can be formally defined as a pseudo Boolean function as follows:

$$\textbf{Minimize}: loss(\mathscr{C}, \mathscr{C}') \quad \textbf{Subject to}: S, E, I, N \tag{10}$$

In this definition, *loss* is a function which takes two configurations as input and returns a numerical value representing the loss of utility when the feature model configuration represented with $\mathscr{C}$ is replaced with the feature model configuration represented with $\mathscr{C}'$. In this function, the parameter $\mathscr{C}$ is constant and the goal is to find an assignment to $\mathscr{C}'$ which minimize the loss of utility. Furthermore, *S, E, I, N* represent different sets of constraints which an assignment to $\mathscr{C}'$ 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 (Bashari et al., 2014; Bagheri et al., 2010) 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 $\mathscr{C}$ with feature model configuration $\mathscr{C}'$ can be represented as a function $loss(\mathscr{C}, \mathscr{C}')$ where:

$$loss(\mathscr{C}, \mathscr{C}') = \sum_{i \in \{1,..,n\}} c_i(1 - c_i')U(f_i) - (1 - c_i)c_i'U(f_i) \tag{11}$$

In this equation, i is the index for features assuming that we have n features where f_i maps to the i^{th} feature. With this, $U(f_i)$ would be the utility of the i^{th} feature. This equation iterates over all features in the feature model and calculates the amount of utility lost by using $\mathscr{C}'$. 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 $\mathscr{C}$ but not available in $\mathscr{C}'$, the utility of the unselected feature is added to the sum of the lost utility. If a feature exists in $\mathscr{C}'$ while it does not exist in $\mathscr{C}$, 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 $loss(\mathscr{C}, \mathscr{C}')$ function for situations when utility values of features are not available. We refer to this function as $distance(\mathscr{C}, \mathscr{C}')$ that uses only the number of features which are different between the two feature model configurations. The *distance* function can be used instead of the *loss* function:

$$distance(\mathscr{C}, \mathscr{C}') = \sum_{i \in \{1,..,n\}} (1 - c_i)c_i' + c_i(1 - c_i') \tag{12}$$

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 $\mathscr{C}'$ represents a valid feature model configuration in terms of structural and integrity constraints. All types of structural and integrity constraints can be represented using linear constraints over features. For example, the optional relationship between the Upload Image feature and the Editing feature can be represented as $c_{UploadImage} - c_{Editing} > -1$. There exist methods for translating a feature model structure and its constraints into a set of linear constraints. In our work, we have used the translation proposed in Noorian et al. (2017).

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 Fig. 1 will require the condition *WatermarkRequested*(im,wt) to be true before execution, which might not necessarily be true. The linear constraints

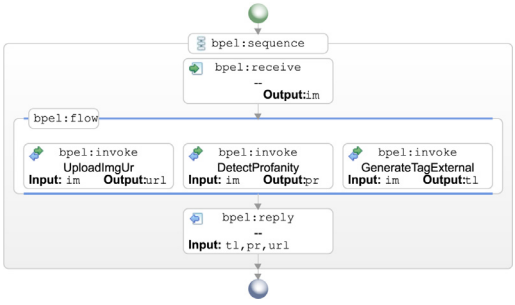

Fig. 8. BPEL process visualization for an alternate solution for BPEL process in Fig. 4 after adaptation by feature model reconfiguration as a result of the *ObjectDetection* service failure.

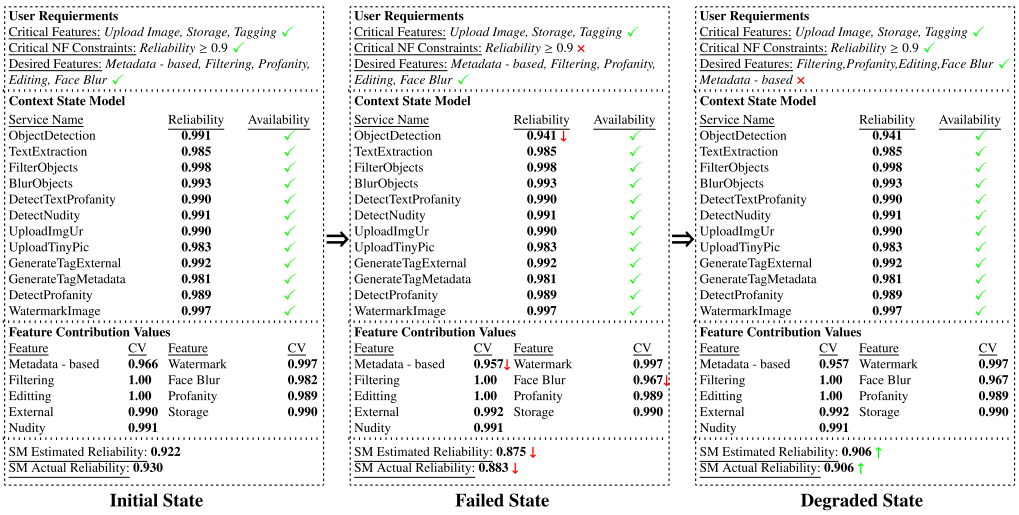

Fig. 9. Initial, failed, and degraded state of a service composition in Fig. 4 after change in the reliability of the *Object Detection* service.

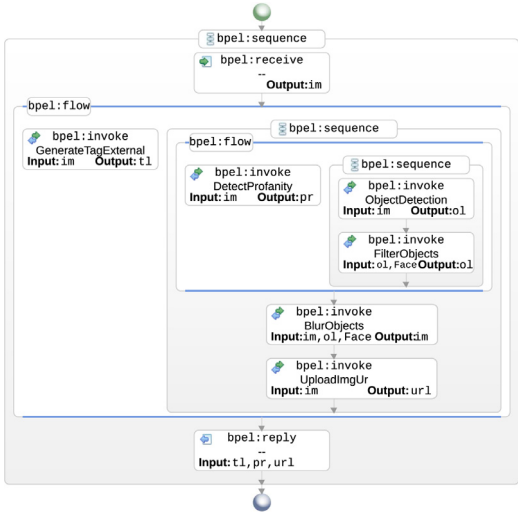

Fig. 10. BPEL process visualization for an alternate solution for BPEL process in Fig. 4 after adaptation by feature model reconfiguration as a result of the decrease in the reliability of the *ObjectDetection* service.

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 $\mathscr{P}(f)$ 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 $\mathscr{P}(f)$ 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:

$$\sum_{i\in\{1..n\}\ s.t.\ \mathscr{P}(f_i)\neg\subseteq Pre(C)} c_i' < 1 \tag{13}$$

Assuming that $Pre(C)$ returns all of the preconditions required by the service composition realizing feature model configuration C as outlined in Bashari et al. (2016), the sum operator in this linear constraint makes sure that the alternate feature model configuration will not have those features that add preconditions not present in the precondition set of the initial service composition. This condition will make sure that the alternate feature model configuration has the same or looser preconditions compared to the failed configuration. In the case of the example feature model configuration, this condition would be $c_{watermark} < 1$ considering that the precondition for the current configuration is an empty set along with the only feature whose precondition is not non-empty. This equation means that the watermark feature cannot be selected in the alternative feature model configuration considering that it requires a text to be watermarked on the image.

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

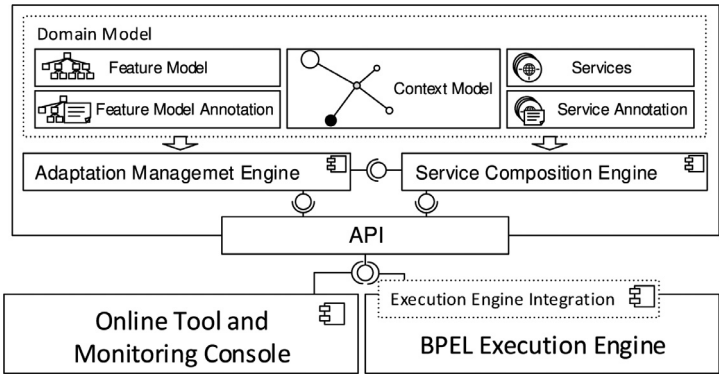

Fig. 11. Architecture of the proposed method.

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:

$$\sum_{i\in\{1..n\}\ s.t.\ f_i\in C}(1-c'_i)+\sum_{i\in\{1..n\}\ s.t.\ f_i\notin C}c'_i>0 \tag{14}$$

This constraint would not be satisfied if the feature model configuration corresponding to the current assignment is the same as the feature model configuration C and true otherwise. For the example configuration, the constraint set I after a functional or non-functional failure - such as the ones mentioned above - initially includes only the following constraint:

$$\begin{aligned}8-c'_{uploadImage}-c'_{tagging}-c'_{metadataBased}-c'_{filtering}-c'_{profanity}\\-c'_{storage}-c'_{editing}-c'_{faceBlur}+c'_{external}+c'_{nudity}+c'_{watermark}>0\end{aligned} \tag{15}$$

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

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 Eq. (1). Therefore, the constraint for a non-functional property can be represented as:

$$\sum_{i\in\{1,..,n\}}p_i c'_i<q \tag{16}$$

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 alternate feature model configuration is less than the threshold value when the user has specified that the non-functional must be lower than the threshold. If the user has specified that the value must be greater than a threshold, the lower than operator should be switched to the greater than operator. In the previous 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. The Initial state in Fig. 9 shows the calculated contribution value for each feature for the example case study as well as the updated contribution value of the features after the example non-functional failure as result of decline in the reliability of the object detection service. It can be seen that the contribution value of metadata-based tagging and face blur show decline, which is intuitive considering that it can be seen in Fig. 4 that this service is used in realizing the functionality of both of those features. Considering that the product operator is an appropriate aggregation operator for reliability, the constraint in Eq. (16) can be defined over the natural logarithm reliability values as shown below:

$$\begin{aligned}&ln(0.957)c'_{metadataBased}+ln(1)c'_{filtering}+ln(1)c'_{editing}+ln(0.992)c'_{external}\\&+ln(0.991)c'_{nudity}+\\&+ln(0.997)c'_{watermark}ln(0.967)c'_{faceBlur}+ln(0.989)c'_{profanity}\\&+ln(0.990)c'_{storage}>ln(0.9)\end{aligned} \tag{17}$$

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

It should be noted that these values are estimated rather than being exactly specified, there is no guarantee that the actual non-functional property of the service composition is the same as the estimated value. In order to address this issue, we use a probabilistic approach. Assuming e is the error of estimation, it can be assumed that it has a normal distribution $\mathcal{N}(0,\sigma)$ with a mean of 0 and a standard deviation of σ. The value of σ can be calculated using the feature model configuration in the dataset and their corresponding compositions. Having the distribution of error e, the value for threshold q can be set in such a way that there is a specific confidence, e.g., 95%, that the generated service composition satisfies.

In summary, based on the formal definitions provided for the loss of utility and the four constraints required in Eq. (10), we are now able to automatically derive a new feature model configuration that alleviates the functional and non-functional requirement violations as well as respecting the four types of constraints. The adaptation happens by optimizing the minimization problem of Eq. (10) in the context of Eqs. (12)–(15). The outcome of this optimization problem is an alternate feature model configuration that would be used for generating an alternate service composition.

For the case of the sample functional failure, the above method results in a feature model configuration which does not provide metadata-based tagging, editing and face blue features but provides external tagging. The new feature model configuration is marked with (❍) in Fig. 1. The alternative feature model configuration represents a degraded state where service composition still provides upload image, storage, and tagging which are critical for the user but does not provide some non-critical features. Fig. 8 shows the alternate service composition built based on this configuration. This BPEL process uses the same service for realizing storage but uses alternative services for tagging and detecting profanity considering that previous logic for realizing them is no longer possible after the failure of the object detection service.

For the case of the sample non-functional failure, the adaptation mechanism results in a feature model configuration which does not provide metadata-based tagging and uses external tagging instead. This feature model has been represented in Fig. 1 with features marked with (■). Again, this configuration provides the critical features and the corresponding service composition has a reliability of 0.906 which is higher than the user specified threshold. Fig. 10 shows the alternative

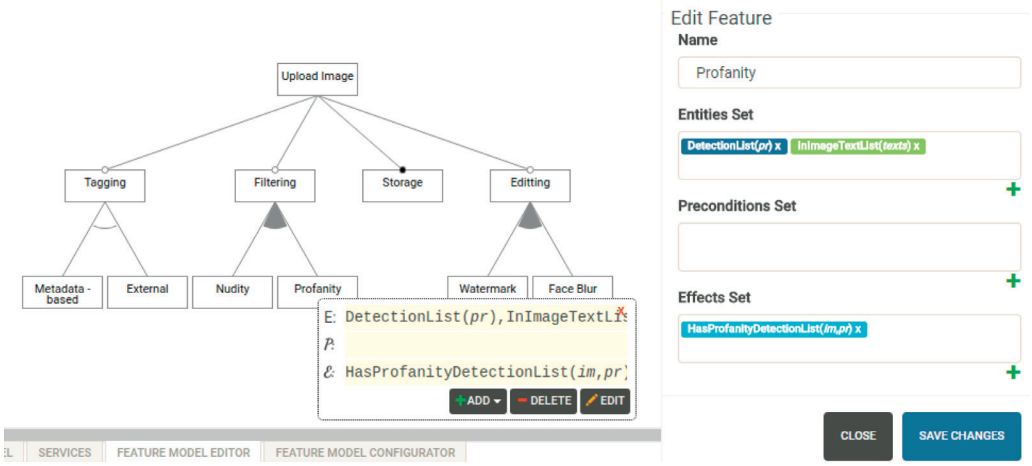

Fig. 12. The Domain Design Tool interface for annotating features in the feature model.

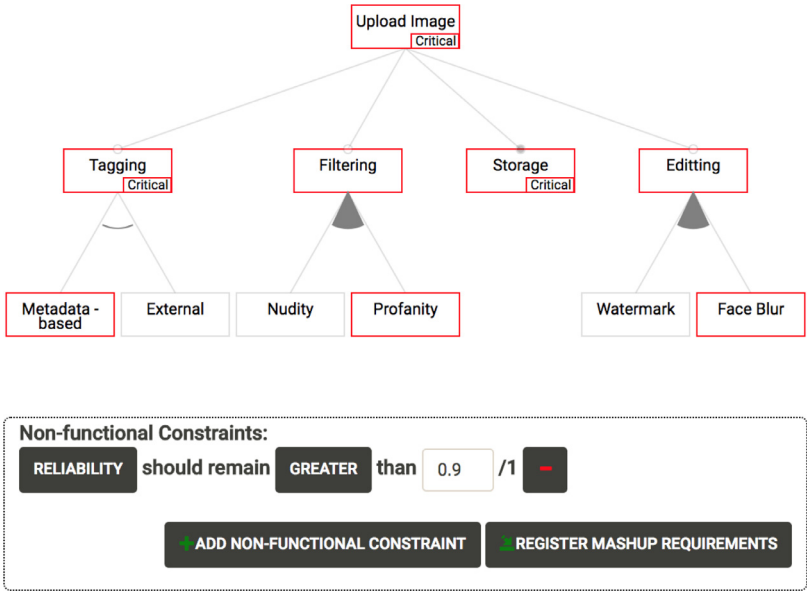

Fig. 13. The Configuration Tool interface for specifying user's requirements.

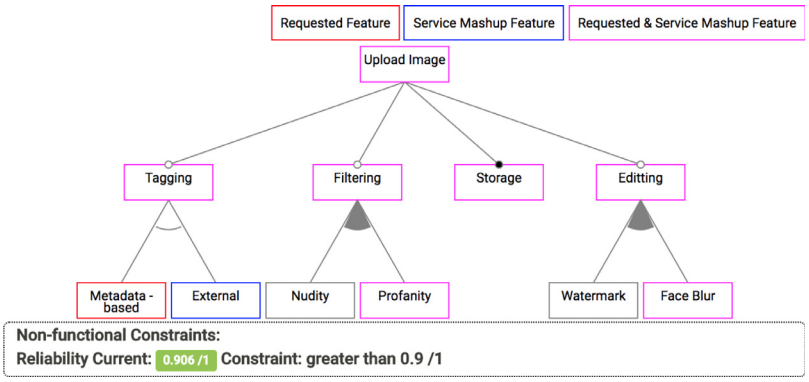

Fig. 14. The Adaptation Management Engine interface for monitoring current state of the service composition.

service composition built based on the alternative feature model configuration. This BPEL process still uses degraded object detection service considering that this service is required for realizing three of the desired features (i.e., tagging, editing, and face blur) but uses different logic for realizing tagging and detecting profanity features which results in higher reliability than the logic in Fig. 4.

6. Architecture

Fig. 11 shows a high level architecture of a general system that would adopt our proposed adaptation management approach. The top component represents a DSPL which is proposed by this approach. It works with online tools and a BPEL engine integration component

which extends BPEL engine to detect failure and perform adaptation through an API component. This architecture can be considered as a connected DSPL (Cetina et al., 2008) considering that the product interacts with the DSPL at runtime to decide about appropriate adaptation at each state. We already discussed how adaptation management logic of this DSPL works in this paper and also discussed the domain model and service composition component in our previous paper (Bashari et al., 2016). In this section, we discuss the functionality of the API and BPEL execution engine components and their role in enabling adaptation.

The API component provides a simplified RESTful service interface for composition and adaptation of services which standardize the way those requests are made. This component works with internal interfaces of composition and adaptation components to realize those services. Developing an API has a number of advantages compared to a solution which extends an existing BPEL logic. First, it promotes separation of concerns by separating the logic required for running the BPEL process from the logic required for performing adaptation. This will facilitate maintaining both DSPL and BPEL execution engines. Second, it allows the DSPL mechanism to manage different BPEL execution engines by developing executions engine integration components for them. This reduces the effort required for introducing the adaptation management mechanism to a new BPEL execution engine.

Then Execution engine integration component is developed for each type of execution engine and works with the internal logic of the execution engine to enable integration with DSPL API. This component has two main functionalities. First, it monitors the BPEL process execution and updates the context state model of the DSPL by calling the relevant APIs. Second, it deploys the replacement BPEL process. We have not yet implemented an execution engine integration component for an execution engine and this has been left as our future work. However, we discuss possible approaches for realizing the functionality of this component.

The Monitoring functionality component can be realized by embedding a probe in the service execution engine which would monitor and collect data about service invocations. Conditions such service availability can be defined by rules over these data (e.g., a service is unavailable if its invocation results in a error three consecutive times) and non-functional properties can be defined by queries over these data (e.g., the response time for a service can be defined as the weighted average of response time over the last hundred invocations). In this paper, we focus on adaptation management and the proposed method is agnostic on how the values in the context state model is updated and therefore different methods (Rabiser et al., 2017; Psiuk and Zielinski, 2015) can be used for enabling monitoring in various domains and for different purposes.

Replacing failed service composition with an alternate one can be simply done by deploying a new process in service execution and redirecting new requests to this service which prevents new invocations of this process from failing. However, our approach requires working on stateless services or having a rollback mechanism in place in order to allow those invocations which have failed mid-execution to recover. Having a rollback mechanism or running on stateless services ensures that the execution of the replacement service composition from the initial state would be a valid execution.

7. Tooling support

The ideas from this paper have been implemented in an online tool suite called *Magus.Online* which is publicly available[1] and supports the whole lifecycle of service composition from design to managing its adaptation. 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.

7.1. 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 enables for creating and editing a context model and its entities and fact types and saving and loading it as an OWL ontology.
- It enables importing and modifying OWL-S definitions of existing services into the domain model.
- It provides a graphical user interface for creating, editing, and serializing feature models as well as annotating features using the context model.

In order create a service 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. Fig. 12 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.

7.2. 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 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

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

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. Fig. 13 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.

7.3. Adaptation management engine & API

Rather than implementing this tool as an extension of an existing BPEL execution engine which limits the application of the proposed method to that specific engine, an API is developed which can work with an execution engine. This API simply sends the current state of the system to the Adaptation Management Engine where it detects failure and recommends replacement BPEL processes when necessary. Specifically, the Adaptation Management Engine & API provides the following functionalities:

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

After a service composition is generated, it can be deployed on a BPEL execution environment. An execution environment can be extended to be managed using the Adaptation Management Engine through its API. This API is well-documented and is available online[2]. Using an API makes adaptation management agnostic to the execution environment and therefore can be used for different BPEL execution environments. The execution environment sends the current context state model to the adaptation management engine through the API where satisfaction of user requirements is examined. In the case of failure, an adaptation takes place and the updated service composition is sent back to the execution environment. The Adaptation Management Engine also updates the online console with updated information of the service composition. Fig. 14 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.

8. Experiments

Our proposed approach has been implemented using the FF planner (Hoffmann and Nebel, 2001) for AI planning and NaPS solver (Sakai and Nabeshima, 2015) 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 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.

8.1. 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 (Segura et al., 2012) to generate the feature models with 30 features which is the average number of features in the SPLOT repository[3] 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 Thum et al. (2009), these parameters reflect structural properties of a real feature model. The number of possible configurations for the generated feature models ranged between 400 and 2,000, which was divided into 5 groups. The feature models were annotated using the context model with 30 entity types and 600 fact types using a feature model annotation generator with parameters $\mathcal{N}(2, 1)$, $\mathcal{N}(0.2, 0.8)$, and $\mathcal{N}(1, 1)$ as the number of entities, preconditions, and effects, respectively. $\mathcal{N}(\mu, \sigma)$ is a normal distribution with a mean of μ and a standard deviation of σ. 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, $\mathcal{N}(1, 1)$, and $\mathcal{N}(2, 1)$ for the number of preconditions and effects.[4]

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

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

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

[4] All generated dataset are available for download at: http://magus.online/experiments/1

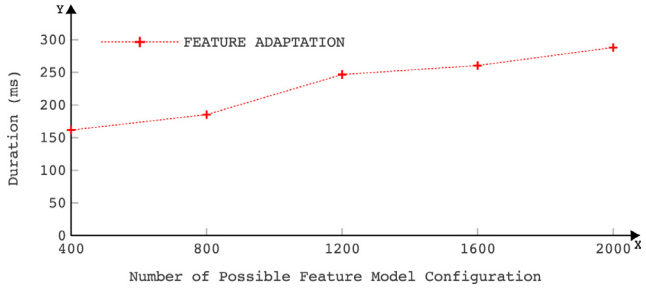

Fig. 15. Feature adaptation execution time in terms of feature model configuration size.

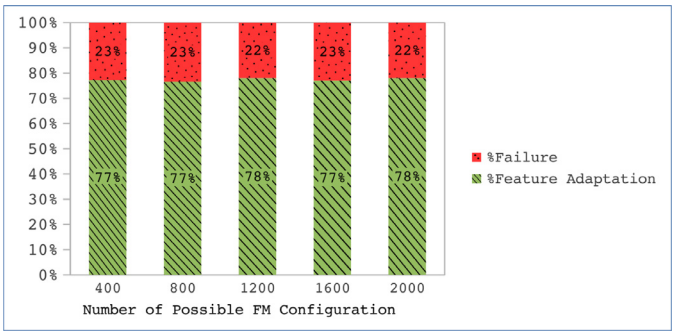

Fig. 16. Distribution of success versus failure of the adaptation mechanism.

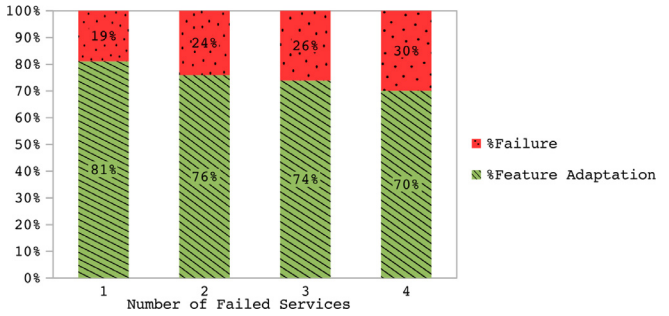

Fig. 17. Distribution of success versus failure mechanisms after a service failure in terms of number of failed services.

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.[5]

For each service family, the experiment was performed by selecting a random valid feature model configuration with 15 ± 1 features from the feature model and generating its corresponding service compositions. Then, a random service from the generated compositions was selected and failed. The failure was handled through adaptation. This process was repeated 100 times for each service family. Having five groups and ten service families in each group, this activity was repeated 5000 times.

Fig. 15 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*.

Fig. 16 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.

8.2. Robustness of adaptation toward 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.

Fig. 17 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.

Fig. 18 shows the distribution of the distance between the alternate

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

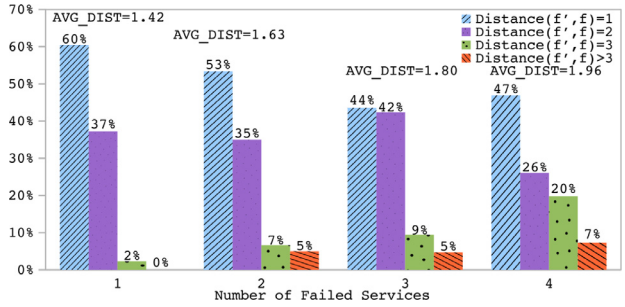

Fig. 18. Distribution of the distance between the alternate feature model configuration and the failed one after service failure in terms of number of failed services.

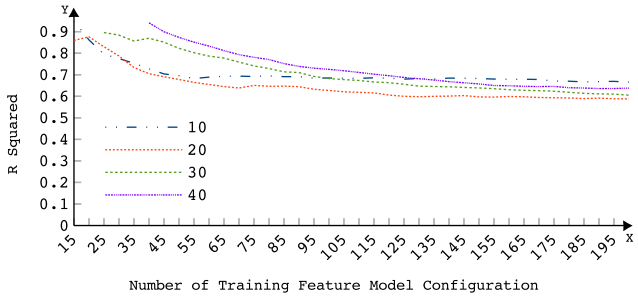

Fig. 19. R Squared of regression model for estimation in terms of number of training feature model configuration.

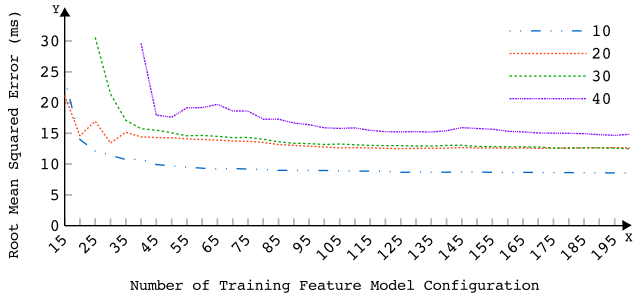

Fig. 20. Root Mean Squared Error of the estimation method in terms of number of training feature model configuration.

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.

8.3. 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:

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

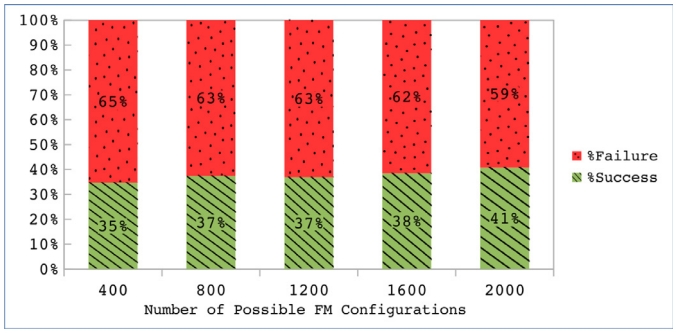

Fig. 21. Distribution of success versus failure mechanisms after a non-functional constraint violation.

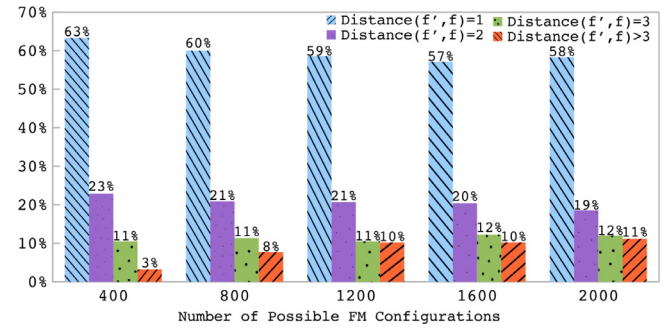

Fig. 22. The distribution of the distance between the alternate feature model configuration and the failed one after a non-functional constraint violation.

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 $\mathcal{N}(200, 50)$. Then, the contribution values of the features were calculated using the smallest possible dataset of feature model configurations, converging to a single result. Using the contribution values, the value of R^2 was calculated. The dataset was augmented with five more feature model configurations. The response time pairs and the updated contribution values and the new R^2 value were then calculated. This process was performed three times for each service family for each group.

Fig. 19 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

Table 1
Comparison of recent approaches for self-healing in service compositions.

Approach	Adaptation Cause		Adaptation Goal		Adaptation Realization	
	Functional	Non-functional	Functional	Non-functional	Strategy	Method
Hristoskova et al. (2013)	Service Failure	–	Recovery	–	Substitution / Reorganization	AI Planning
Tan et al. (2014)	Service Failure	–	Recovery	Mitigation	Reorganization	Genetic Algorithms
Angarita et al. (2016)	Service Failure	–	Recovery	Mitigation	Retry/Substitution	Rule-based
Nallur and Bahsoon (2013)	–	Constraint Violation	Recovery	Recovery	Substitution	Market-based Heuristics
Carzaniga et al. (2015)	Service Failure	–	Recovery	–	Reorganization	Rule-based
Our Approach	Service Failure	Constraint Violation	Mitigation	Recovery	Reorganization	Reduction Pseudo-boolean Optimization

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 Eq. (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.

Fig. 20 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.

8.4. 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 $\mathcal{N}(200, 50)$. Then, a random feature model configuration from the feature model was selected and the corresponding service composition was built. Then, the response time for the built service composition was calculated using service annotations. Then, a non-functional constraint was added having a threshold value which is equal to 110% of the calculated response time. Afterward, the services in the service composition were randomly selected and their response time was increased by 25% until the non-functional constraint 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.

Fig. 21 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.

Fig. 22 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.

8.5. Threats to validity

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

8.5.1. Internal validity

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

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

which is the value of the non-functional property. However, there are other factors which can affect non-functional properties of a service considering that different BPEL processes with different non-functional properties can realize a feature model configuration. The design of the planner affects which of those processes are selected. The planner may have a bias on selecting a specific realization of a feature model configuration and therefore our observation may not be valid when a realization is selected randomly from the set of possible realizations. However, this does not affect the applicability of the proposed method used for estimating the non-functional properties of the process created using a planner.
- Considering that there are currently no real-world dataset available to perform experiments, the models used in these experiments were synthetically generated. All the parameters and distributions used in generating the models have been reported in the paper for replication studies. However, there could be unaccounted parameters that could affect the values observed in the experiments if and when such unaccounted parameters are found.

8.5.2. External validity

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

- The values reported on the success rate of the adaptation in response to both functional and non-functional failures are on the domains which have been synthetic. The success rate of adaptation can change depending on the domain model as a result of change in the different properties of the domain model. The effect of this threat can be mitigated by applying the proposed method on real-world domains and reporting on the efficiency of the proposed approach on those domains.
- The way that a non-functional property for a service composition is calculated based on the value of individual services value differs for various types of non-functional properties. This can affect the accuracy and fitness of the method for estimating non-functionality of service compositions based on the selected features. This threat can be mitigated by extending Experiment 2 of the evaluation on larger sets of non-functional property types.

8.6. Lessons learned

Based on our experience from the experiments, we present the lessons that was learned from these experiments.

- In our work, we assumed a linear relationship between the presence of the features and non-functional properties of a service composition. Although this is a simplification of the complex relationship between these two variables, our experiments showed such assumption can be effectively used for deciding about adaptation in the case of response time. However, this might not necessarily be the case for all types of non functional properties considering that the way a non-functional property is calculated can differ for different types. Therefore, our findings might not be generalizable to all non-functional properties. As future work, the effectiveness of the proposed method for different types of non-functional properties will be evaluated and we will look for alternative estimation approaches if and when the linear relation does not work effectively.
- One of the limitations of our approach is that it takes feature adaptation as the first step toward addressing functional or non-functional failure. Although, this allows the service composition to recover from failures which cannot be addressed with current failure recovery methods, it may result in larger effect on the service composition functionality in cases where a failure can be addressed by smaller changes. As a next step, we will work on a hybrid method which applies existing failure recovery methods such as service replacement and re-planning as a first step to address the failure and uses feature adaptation when such methods cannot recover the service composition from failure.
- As a part of the objective evaluation of the proposed method, we tried to find datasets and tools which can be used to compare the proposed method with. We were not able to find such datasets or tools to perform the comparison. This can be considered as a limitation of the work reported in this domain. In order to enable other researchers to compare their method with the proposed method, we publicly shared the datasets of this paper as well as the source code of the tool suite.
- Although the proposed adaptation mechanism in this study makes sure the critical constraints are satisfied after an adaptation, it does not guarantee the service composition is in optimal state in terms of non-functional properties. This can be considered as a limitation of the proposed approach. In future work, we will look for ways to adapt the composition in such a way that ensures that the composition is in optimized state in terms of non-functional properties as well.

9. 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.

9.1. 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 (Moser et al., 2008; Subramanian et al., 2008; Koning et al., 2009). For example in Subramanian et al. (2008), 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 ($\geq$ 2013) compared to the proposed method. In the following, we go through the different aspects of self-healing mechanisms covered in Table 1 and dummyTXdummy- discuss similarities and differences between existing methods and our proposed method.

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. (2013) 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 and Bahsoon (2013) 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. (2016) 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. (2014) 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 (Huang et al., 2015): *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 (Elnozahy et al., 2002; Dobson, 2006). 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 (Carzaniga et al., 2015; Huang et al., 2015; Tibermacine et al., 2015). One example of such methods is the approach proposed by Christos et al. (2008) 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 (Ali and Reiff-Marganiec, 2012; Liu et al., 2007), AI planning (Huang et al., 2015; Hristoskova et al., 2013), and other methods (Boudaa et al., 2017; Ismail and Cardellini, 2013) to recover from failure. For example, the failure recovery method proposed in Huang et al. (2015) 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.

9.2. 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 (Bashari et al., 2017a). 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 (Baresi et al., 2012; Cetina et al., 2009). For example in Cetina et al. (2009), 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 (White et al., 2009) 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 shut downs those components corresponding to the unselected features and launches those corresponding to the selected features using a method called *micro-booting*. The work in Pascual et al. (2015) 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 (Fleurey and Solberg, 2009). Third, existing approaches assume that there is a direct relationship between features and services, which is not always true in practice (Trinidad et al., 2007). 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.

10. 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 compositions 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

Alegre, U., Augusto, J.C., Clark, T., 2016. Engineering context-aware systems and applications: a survey. J. Syst. Softw. 117 (Supplement C), 55–83. http://dx.doi.org/10.1016/j.jss.2016.02.010.

Alfrez, G.H., Pelechano, V., 2017. Achieving autonomic web service compositions with models at runtime. Comput. Electr. Eng. 63, 332–352. http://dx.doi.org/10.1016/j.compeleceng.2017.08.004.

Ali, M.S., Reiff-Marganiec, S., 2012. Autonomous failure-handling mechanism for wf long running transactions. Proceedings of the IEEE Ninth International Conference on Services Computing (SCC). IEEE, pp. 562–569.

Angarita, R., Rukoz, M., Cardinale, Y., 2016. Modeling dynamic recovery strategy for composite web services execution. World Wide Web 19 (1), 89–109.

Bagheri, E., Asadi, M., Gasevic, D., Soltani, S., 2010. Stratified analytic hierarchy process: Prioritization and selection of software features. Proceedings of the International Conference on Software Product Lines (SPLC 2010). Springer, pp. 300–315.

Baresi, L., Guinea, S., Pasquale, L., 2012. Service-oriented dynamic software product lines. Comput. (Long Beach Calif) 45 (10), 42. http://dx.doi.org/10.1109/MC.2012.289.

Bashari, M., Bagheri, E., Du, W., 2016. Automated composition of service mashups through software product line engineering. Proceedings of the International Conference on Software Reuse. Springer, pp. 20–38.

Bashari, M., Bagheri, E., Du, W., 2017a. Dynamic software product line engineering: a reference framework. IJSEKE 27 (1), 1–44.

Bashari, M., Bagheri, E., Du, W., 2017b. Self-healing in service mashups through feature adaptation. Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, Sevilla, Spain, September 25–29, 2017. pp. 94–103. http://dx.doi.org/10.1145/3106195.3106215.

Bashari, M., Noorian, M., Bagheri, E., 2014. Product line stakeholder preference elicitation via decision processes. Int. J. Knowl. Syst. Sci. (IJKSS) 5 (4), 35–51.

Benavides, D., Segura, S., Ruiz-Cortes, A., 2010. Automated analysis of feature models 20 years later: a literature review. Inf. Syst. 35 (6), 615–636.

Benavides, D., Trinidad, P., Ruiz-Corts, A., 2005. Automated reasoning on feature models. Advanced Information Systems Engineering. Springer, pp. 381–390.

Berger, J.O., 2013. Statistical Decision Theory and Bayesian Analysis. Springer Science & Business Media.

Boros, E., Hammer, P.L., 2002. Pseudo-boolean optimization. Discrete Appl. Math. 123 (1), 155–225.

Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Bosch, J., Capilla, R., 2012. Dynamic variability in software-intensive embedded system families. Dyn. Var. Softw.-Intensive Embed. Syst. Fam. 45 (10), 28–35. http://dx.doi.org/10.1109/MC.2012.287.

Boudaa, B., Hammoudi, S., Mebarki, L.A., Bouguessa, A., Chikh, M.A., 2017. An aspect-oriented model-driven approach for building adaptable context-aware service-based applications. Sci. Comput. Program 136, 17–42.

Canfora, G., Penta, M.D., Esposito, R., Villani, M.L., 2008. A framework for qos-aware binding and re-binding of composite web services. Journal of Systems and Software 81 (10), 1754–1769. http://dx.doi.org/10.1016/j.jss.2007.12.792. Selected papers from the 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, September 721, 2006.

Carzaniga, A., Gorla, A., Perino, N., Pezze, M., 2015. Automatic workarounds: exploiting the intrinsic redundancy of web applications. ACM Trans. Softw. Eng. Methodol. (TOSEM) 24 (3), 16.

Cetina, C., Giner, P., Fons, J., Pelechano, V., 2009. Autonomic computing through reuse of variability models at runtime: the case of smart homes. Comput. (Long Beach Calif) 42 (10), 37–43.

Cetina, C., Pelechano, V., Trinidad, P., 2008. An architectural discussion on dspl. Proceedings of the 12th International Software Product Line Conference, Volume 2. pp. 59–68.

Chen, L., Huang, L., Li, C., Wu, X., 2017. Self-adaptive architecture evolution with model checking: a software cybernetics approach. J. Syst. Softw. 124 (Supplement C), 228–246. http://dx.doi.org/10.1016/j.jss.2016.03.010.

Cheng, B., Zhai, Z., Zhao, S., Chen, J., 2017. Lsmp: a lightweight service mashup platform for ordinary users. IEEE Commun. Mag. 55 (4), 116–123.

Christos, K., Vassilakis, C., Rouvas, E., Georgiadis, P., 2008. Exception resolution for bpel processes: a middleware-based framework and performance evaluation. Proceedings of the 10th International Conference on Information Integration and Web-based Applications and Services. ACM, pp. 248–256.

CÃ!‘mara, J., Correia, P., de Lemos, R., Garlan, D., Gomes, P., Schmerl, B., Ventura, R., 2016. Incorporating architecture-based self-adaptation into an adaptive industrial software system. J. Syst. Softw. 122 (Supplement C), 507–523. http://dx.doi.org/10.1016/j.jss.2015.09.021.

Dobson, G., 2006. Using ws-bpel to implement software fault tolerance for web services. Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO'06). IEEE, pp. 126–133.

Een, N., Sorensson, N., 2006. Translating pseudo-boolean constraints into sat. J. Satisfiability Boolean Modeling Comput. 2, 1–26.

Elnozahy, E.N., Alvisi, L., Wang, Y.-M., Johnson, D.B., 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. (CSUR) 34 (3), 375–408.

Fleurey, F., Solberg, A., 2009. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. MODELS. Springer, pp. 606–621.

Gerostathopoulos, I., Bures, T., Hnetynka, P., Keznikl, J., Kit, M., Plasil, F., Plouzeau, N., 2016. Self-adaptation in software-intensive cyberâ;;physical systems: from system goals to architecture configurations. J. Syst. Softw. 122 (Supplement C), 378–397. http://dx.doi.org/10.1016/j.jss.2016.02.028.

Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2008. Dynamic software product lines. Comput. (Long Beach Calif) 41 (4), 93–95.

Hoffmann, J., Nebel, B., 2001. The ff planning system: fast plan generation through heuristic search. J. Artif. Intell. Res. 14, 253–302.

Hristoskova, A., Volckaert, B., Turck, F.D., 2013. The wte framework: automated construction and runtime adaptation of service mashups. Automat. Softw. Eng. 20 (4), 499–542.

Huang, H., Chen, X., Wang, Z., 2015. Failure recovery in distributed model composition with intelligent assistance. Inf. Syst. Front. 17 (3), 673–689.

Im, J., Kim, S., Kim, D., 2013. Iot mashup as a service: cloud-based mashup service for the internet of things. Proceedings of the IEEE International Conference on Services Computing (SCC). IEEE, pp. 462–469.

Ismail, A., Cardellini, V., 2013. Towards self-adaptation planning for complex service-based systems. Proceedings of the International Conference on Service-Oriented Computing. Springer, pp. 432–444.

Jiao, W., Sun, Y., 2016. Self-adaptation of multi-agent systems in dynamic environments based on experience exchanges. J. Syst. Softw. 122 (Supplement C), 165–179. http://dx.doi.org/10.1016/j.jss.2016.09.025.

Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Comput. (Long Beach Calif) 36 (1), 41–50.

Koning, M., ai Sun, C., Sinnema, M., Avgeriou, P., 2009. Vxbpel: supporting variability for web services in bpel. Inf. Softw. Technol. 51 (2), 258–269.

Lee, K., Kang, K.C., Lee, J., 2002. Concepts and guidelines of feature modeling for product line software engineering. International Conference on Software Reuse. Springer, Berlin, Heidelberg, pp. 62–77.

Lemos, A.L., Daniel, F., Benatallah, B., 2016. Web service composition: a survey of techniques and tools. ACM Comput. Surv. (CSUR) 48 (3), 33.

Lemos, R.D., Giese, H., Mller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T., 2013. Software engineering for self-adaptive systems: a second research roadmap. Software Engineering for Self-Adaptive Systems II. Springer, pp. 1–32.

Liu, A., Li, Q., Xiao, M., 2007. A declarative approach to enhancing the reliability of bpel processes. Proceedings of the IEEE International Conference on Web Services (ICWS 2007). IEEE, pp. 272–279.

Manquinho, V., Marques-Silva, J., Planes, J., 2009. Algorithms for weighted boolean optimization. Proceedings of the International Conference on Theory and Applications of Satisfiability Testing. Springer, pp. 495–508.

Margaris, D., Georgiadis, P., Vassilakis, C., 2015. On replacement service selection in ws-bpel scenario adaptation. Proceedings of the IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA). IEEE, pp. 10–17.

Margaris, D., Vassilakis, C., Georgiadis, P., 2016. Improving qos delivered by ws-bpel scenario adaptation through service execution parallelization. Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1590–1596. http://dx.doi.org/10.1145/2851613.2851805.

Mohabbati, B., Gasevic, D., Hatala, M., Asadi, M., Bagheri, E., Boskovic, M., 2011. A quality aggregation model for service-oriented software product lines based on variability and composition patterns. Proceedings of the Service-Oriented Computing - 9th International Conference, ICSOC 2011, Paphos, Cyprus, December 5–8, 2011 Proceedings. pp. 436–451. http://dx.doi.org/10.1007/978-3-642-25535-9_29.

Montalvillo, L., DÃaz, O., 2016. Requirement-driven evolution in software product lines: a systematic mapping study. J. Syst. Softw. 122 (Supplement C), 110–143. http://dx.doi.org/10.1016/j.jss.2016.08.053.

Moser, O., Rosenberg, F., Dustdar, S., 2008. Non-intrusive monitoring and service adaptation for ws-bpel. Proceedings of the 17th International Conference on World Wide Web. ACM, pp. 815–824.

Nallur, V., Bahsoon, R., 2013. A decentralized self-adaptation mechanism for service-based applications in the cloud. IEEE Trans. Softw. Eng. 39 (5), 591–612.

Noorian, M., Bagheri, E., Du, W., 2017. Toward automated quality-centric product line configuration using intentional variability. J. Softw.: Evol. Process. 29 (9), 1–26.

OASIS. Web services business process execution language version 2.0. URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

Pascual, G.G., Lopez-Herrejon, R.E., Pinto, M., Fuentes, L., Egyed, A., 2015. Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications. J. Syst. Softw. 103 (Supplement C), 392–411. http://dx.doi.org/10.1016/j.jss.2014.12.041.

Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D., 2014. Context aware computing for the internet of things: a survey. IEEE Commun. Surv. Tutor. 16 (1), 414–454.

Pohl, K., Bockle, G., Linden, F.V.D., 2005. Software Product Line Engineering:

Foundations, Principles and Techniques. 10 Springer.
Psiuk, M., Zielinski, K., 2015. Goal-driven adaptive monitoring of soa systems. J. Syst. Softw. 110, 101–121. http://dx.doi.org/10.1016/j.jss.2015.08.015.
Rabiser, R., Guinea, S., Vierhauser, M., Baresi, L., Grnbacher, P., 2017. A comparison framework for runtime monitoring approaches. J. Syst. Softw. 125, 309–321. http://dx.doi.org/10.1016/j.jss.2016.12.034.
Sakai, M., Nabeshima, H., 2015. Construction of an robdd for a pb-constraint in band form and related techniques for pb-solvers. IEICE Trans. Inf. Syst. 98 (6), 1121–1127.
Segura, S., Galindo, J.A., Benavides, D., Parejo, J.A., Ruiz-Cortes, A., 2012. Betty: benchmarking and testing on the automated analysis of feature models. Workshop on Variability Modeling of Software-Intensive Systems. ACM, pp. 63–71.
Späth, H., 2014. Mathematical Algorithms for Linear Regression. Academic Press.
Subramanian, S., Thiran, P., Narendra, N.C., Mostefaoui, G.K., Maamar, Z., 2008. On the enhancement of bpel engines for self-healing composite web services. Proceedings of the International Symposium on Applications and the Internet, 2008. SAINT 2008. IEEE, pp. 33–39.
ai Sun, C., Ran, Y., Zheng, C., Liu, H., Towey, D., Zhang, X., 2018. Fault localisation for ws-bpel programs based on predicate switching and program slicing. J. Syst. Softw. 135, 191–204. http://dx.doi.org/10.1016/j.jss.2017.10.030.
Tan, T.H., Chen, M., André, É., Sun, J., Liu, Y., Dong, J.S., 2014. Automated runtime recovery for qos-based service composition. Proceedings of the 23rd International Conference on World Wide Web. ACM, pp. 563–574.
Thum, T., Batory, D., Kastner, C., 2009. Reasoning about edits to feature models. Proceedings of the ICSE. IEEE, pp. 254–264.
Tibermacine, O., Tibermacine, C., Cherif, F., 2015. A process to identify relevant substitutes for healing failed ws-* orchestrations. J. Syst. Softw. 104 (Supplement C), 1–16. http://dx.doi.org/10.1016/j.jss.2015.02.028.
Trinidad, P., Ruiz-Cortes, A., Pena, J., Benavides, D., 2007. Mapping feature models onto component models to build dynamic software product lines. International Workshop on DSPL at (SPLC 2007). pp. 51–56.
Wang, X.H., Zhang, D.Q., Gu, T., Pung, H.K., 2004. Ontology based context modeling and reasoning using owl. Pervasive Computing and Communications Workshops. IEEE, pp. 18–22.
White, J., Strowd, H.D., Schmidt, D.C., 2009. Creating self-healing service compositions with feature models and microrebooting. J. Bus. Process Integr. Manag. 4 (1), 35–46.

Mahdi Bashari received his M.Sc. in Software Engineering from the Sharif University of Technology, Tehran, Iran. Currently, he is a Ph.D. student at the University of New Brunswick and a member of the Laboratory for Systems, Software and Semantics at Ryerson University, Canada. His research interests are Engineering Dynamic Software Product Lines and Self-adaptive Systems. He can be reached at mbashari@unb.ca.

Ebrahim Bagheri is an Associate Professor, a Canada Research Chair in Software and Semantic Computing and an NSERC/WL Industrial Research Chair in Social Media Analytics at Ryerson University, Toronto, Canada. He has an active research program in the areas of Semantic Web, Social Computing and Software Engineering. He is a Senior Member of IEEE and an IBM CAS Fellow. He can be reached at Bagheri@ryerson.ca.

Weichang Du has been a Professor in Computer Science at University of New Brunswick, Canada since 1991. He obtained his M.Sc. and Ph.D. in Computer Science from University of Victoria, Canada in 1985 and 1991. In past 25 years, he has published many research articles and supervised more than 50 Masters and Ph.D.students. In recent years, he has been conducting research on designing and developing knowledge-ábased and intelligent software and knowledge systems and applications on Web and mobile platforms, including health related systems and applications. He can be reached at wdu@unb.ca.