Self-healing in Service Mashups Through Feature Adaptation

Mahdi Bashari
Faculty of Computer Science,
University of New Brunswick
Fredericton, NB, Canada E3B 5A3
mbashari@unb.ca

Ebrahim Bagheri
Department of Electrical and Computer Engineering, Ryerson University
Toronto, ON, Canada M5B 2K3
bagheri@ryerson.ca

Weichang Du
Faculty of Computer Science,
University of New Brunswick
Fredericton, NB, Canada E3B 5A3
wdu@unb.ca

ABSTRACT

The composition of the functionality of multiple services into a single unique service mashup has received wide interest in the recent years. Given the distributed nature of these mashups where the constituent services can be located on different servers, it is possible that a change in the functionality or availability of a constituent service result in the failure of the service mashup. In this paper, we propose a novel method based on the Software Product Line Engineering (SPLE) paradigm which is able to find an alternate valid service mashup which has maximum possible number of original service mashup features in order to mitigate a service failure when complete recovery is not possible. This method also has an advantage that it can recover or mitigate the failure automatically without requiring the user to specify any adaptation rule or strategy. We show the practicality of our proposed approach through extensive experiments.

CCS CONCEPTS

•**Software and its engineering →Software product lines;**

ACM Reference format:
Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. 2017. Self-healing in Service Mashups Through Feature Adaptation. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 10 pages.
DOI: 10.1145/3106195.3106215

1 INTRODUCTION

With the growing number of publicly available services, there has been a trend on providing methods and tools to allow users to take advantage of these services in order to create their own personalized services called *service mashups* [6]. There has been efforts in this area to provide automated or semi-automated methods for composing service mashups [9, 14]. However, the way these methods specify the requirements for the service mashup usually requires in-depth expertise in SOA. In our earlier work [2], we proposed using SPLE concepts [21] to facilitate the service composition process. Software product lines provide a rich set of models and methods for managing variability in a problem domain. Specifically, we utilized *feature models* [17] as the main model for expressing the specification of the desired service mashup. Feature models are capable of capturing variability of a system in term of its features, which are themselves user-tangible fragments of functionality. Feature models have been effectively used in the software product line community as a shared artifact between system developers and end-users for specifying product requirements [5].

Since the services which make up a service mashup can be hosted by different providers, service mashups operate within a highly dynamic environment in the sense that services can become unavailable, change interface, or change functionality at any time. These changes may result in the failure of the service mashup that relies on them, which directly affects mashup reliability. In this paper, a self-healing *Dynamic Software Product Line (DSPL)* [8, 12] is proposed where a service mashup is either repaired or reorganized even in cases when complete recovery from failure is not possible. In the proposed failure recovery method, online planning is used in order to find alternate service mashups, which can replace the failed service mashup. If such alternative mashup cannot be found, the recovery method tries to mitigate the effect of failure by finding an alternate service mashup with minimal loss of features compared to the existing service mashup through feature model re-configuration.

There are existing methods for enabling self-adaptation through building DSPLs [1, 3, 16, 19, 20, 26]. These works usually enable adaptation by defining a detailed adaptation strategy over features. Consequently, they need to define a large number of rules in order to enable self-healing in response to service failure since a system can fail in many different ways. Therefore, we adopted a different approach to enable self-healing where the recovery method automatically looks for an alternative feature model configuration where those features which result in a failure are not considered. Furthermore, some researchers have already focused on the automated recovery from failed services by using AI planning [14, 15]. However, these methods also fall short when restoration of the service mashup to its full functionality is not possible.

This work is different from our previous work [2] in the sense that it provides a method for enabling self-healing on the service mashups whereas, in our previous work, we only generated a static service mashup based on a feature model configuration. Concretely, the current paper offers the following contributions:

- We propose a method for enabling automated self-healing in service mashups without requiring any auxiliary information about possible mitigation strategies provided by the users or service mashup designers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '17, Sevilla, Spain
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5221-5/17/09…$15.00
DOI: 10.1145/3106195.3106215

- We propose a method which enables service mashups to be able to continue servicing with limited features when complete service recovery is not possible.

In the following, we first provide a background on feature models and how our proposed service composition method works using an example. Subsequently in the next section, the problem which we address in this paper is formally defined and the proposed approach is outlined. In the experiments section, three different experiments focusing on practicality and effectiveness of the proposed approach are designed and performed. Furthermore, our work is compared with existing work in the literature in the related work section.

2 BACKGROUND

The work in this paper revolves around the fundamental notion of feature models where the expected requirements of a service mashup are specified by configuring a feature model and using that configured feature model to automatically compose a service mashup. On this basis, the goal of our work is to dynamically adapt a service mashup at runtime when a service failure occurs. In the following, background on feature models, how service mashups can be modeled in this context, and how automated composition of service mashups can be performed will be provided.

Feature models are among the popular models used in the SPLE community for representing possible variability of the problem domain [5]. Feature models allow for hierarchical representation of features that are related to each other through structural and/or integrity constraints. The structural constraints relate features to their parents through Mandatory, Optional, Alternative, and Or relations. Integrity constraints represent dependencies between features which are not hierarchically related. Having a feature model, users can define their desired variants by selecting features from the feature model which results in a specialized model known as *feature model configuration*. A feature model configuration consists of a subset of features in the feature model which respects structural and integrity constraints and can be used as the reference model for composing the desired system.

Figure 1-A shows a feature model for a family of service mashups 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 the image and finds some keywords which 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 set of features marked with (•) in Figure 1 shows a valid feature model configuration.

In the work proposed in [2], service mashups are represented in Business Process Execution Language (BPEL), which work with a set of partner services from a service repository. In order to relate the services and features together, a new model known as the *context model* is used which contains a set of entity and fact types and instances. Using the context model, both features and services are annotated by how they affect the context model. Using these annotations, the problem of finding a service mashup for a feature model configuration is reduced to a planning problem and solved using an AI planner.

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

Using the context model, the services in the service repository are annotated. Each service has two sets: I and O, which define its inputs and outputs where members of these two sets are entities with types from the context model. Furthermore, each service is 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 ($\mathcal{P}_I$), (2) those facts that will be true after invocation of the service (Q_I), and (3) those facts that will become false after invocation of the service ($\mathcal{R}_I$). Figure 1-C shows some parts of the service repository for the example service mashup 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 $\mathcal{P}_I$ includes two facts of type $HasObject$ and $HasText$. The set Q_I has a fact of type $HasMetadataTags$ which specifies that the tag list entity in the output would represent the tags generated from the objects and texts in the image. The set $\mathcal{R}_I$ is empty for this service.

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

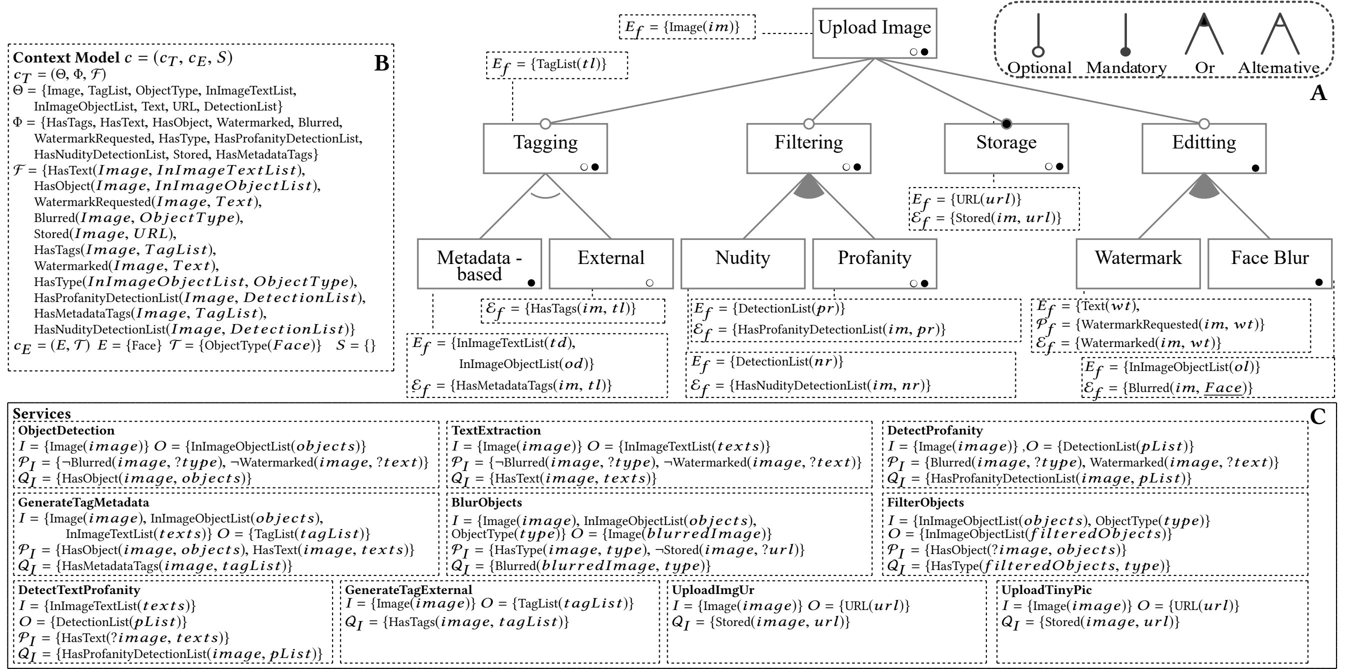

Figure 1: A. An annoated feature model for the upload image family. B. The context model for the upload image service mashup family C. Part of the service repository and annotations for the upload image service mashup family.

We refer to the collection of a feature model, a service repository, context model, service annotations, and feature annotations as a *domain model* for the service mashup family. In order to express his/her requirements, the user configures the feature model from a domain model. Having the domain model and a feature model configuration, the problem of finding a sequence of service invocations which satisfy the requirements of that feature model configuration can be reduced into a planning problem represented using the PDDL language and solved using an existing planner. After invoking a planner with the planning problem, the planner will return a single sequence of service invocations. Afterwards, an optimization method will convert the sequence of execution to a dependency graph with the goal of introducing concurrency in order to optimize non-functional properties such as response time compared to the input sequence of invocations. In the dependency graph, service invocations are represented as nodes and their dependencies on other service invocations are represented as edges. Then, a BPEL generation method converts the dependency graph into the BPEL process structure.

For example, Figure 2 shows the visualization of the BPEL process code for the feature model configuration containing features marked with (•) in Figure 1-A. The service operations in this process have been organized with three structure types: flow, sequence, and link. The service operations in the sequence structure should be executed in order, the service operations in a flow structure can be executed in any order or simultaneously, and the link structure allows enforcing order between two operations in different flow and sequence structures. Using the link structure makes sure that operations in the source for the link is executed before the operations in the target of the link.

3 PROBLEM STATEMENT AND SOLUTION APPROACH

Based on the foundation introduced in the previous section, a service mashup can be generated based on a set of selected features. However, since service mashups are often composed of services that are likely to be distributed across different third-party providers, it is possible that these services go offline or change functionality which results in the failure of the service mashup. In order to address this issue, we adopt a Dynamic Software Product Line (DSPL) approach. Dynamic software product line engineering is a paradigm for enabling self-adaptation using concepts from the SPLE domain. By taking a DSPL approach, we propose an automated feature model re-configuration method which is capable of finding an alternate feature model configuration with partial features when a service mashup fails and cannot be recovered. In our work, the feature model is automatically re-configured to a new configuration such that it does not rely on the failed services for its realization and at the same time has the least *loss of utility* compared to the current feature model configuration. The utility is a quantitative value defined for each set of features and used for representing the user relative preference over different set of features. This means that if the utility for a set of features is higher than another set of features, the user prefers the first set of features over the second one.

Problem Statement. Lets assume that *SM* is a service mashup, which has been generated based on feature model configuration *C* and that service *s* which is invoked in service mashup *SM* has become unavailable. The goal is to propose an *automated* method to find an alternative service mashup for replacing the failed service mashup such that the alternative (1) is *valid* with regards to the

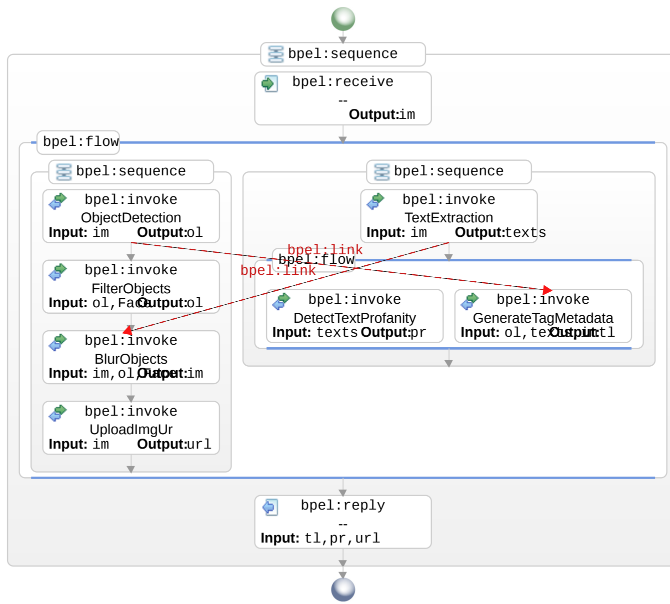

Figure 2: BPEL process visualization for a possible service mashup satisfing requirements of feature model configuration in Figure 1.

constraints, (2) does not rely on *s*, and (3) has the *least loss of utility* compared to the failed service mashup in terms of the provided features.

In order to be able to react to failure, we first define a *service availability model* which contains availability status of all services in the service repository. Additionally, we also assume that the service mashup has a rollback mechanism which rolls back the transactional processes to the initial state if a service failure occurs during execution of the service mashup. The BPEL language itself provides mechanisms for such situations. The failure recovery and mitigation process works as follows: after a failure takes place during mashup execution, the service availability model is updated to reflect the unavailability of the service. Afterwards, the recovery process begins by looking for a service which has the same preconditions and effects. If such service is found, the invocation of the failed service in the current process is replaced by the alternate service and the new process is executed. Otherwise, a planning mechanism is used to find an alternate process which has the same functionality. If such process does not exist, the mitigation process is started in which an alternate service mashup for replacing the current service mashup is found such that the alternate service mashup has minimal loss of utility for the user in terms of features.

3.1 Adaptation without change in features

The first recovery strategy which is used after a service failure takes place is to find an alternate service with the same functionality. As discussed in the background section, the functionality of services are defined using the preconditions and effects defined over their inputs and outputs. If such a service is found, the invocations of the failed service is replaced with the invocation of the alternate service in the failed service mashup process. For example in the service mashup in Figure 2, if the service *UploadImgUr* fails, it can

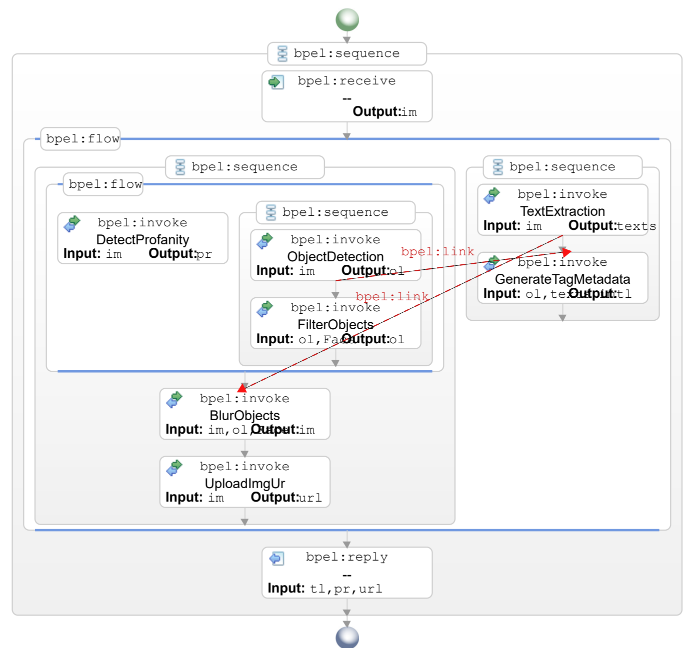

Figure 3: BPEL process visualization for an alternate solution for BPEL process in Figure 2 after adaptation by replanning as a result of *DetectTextProfanity* failure.

be replaced with the *UploadTinyPic* service since it can be seen in Figure 1-C that they have the same functionality in terms of preconditions and effects.

If such a service is not found, the objective will be to find a alternate service mashup that would be able to replace the failed service. Finding such a alternate service mashup can be defined as a service mashup composition problem with the difference that the failed service and other services marked as unavailable in the service availability model are removed from the domain model. Here, replanning can be used to find the alternate service mashup. If replanning results in a solution, the solution is optimized and based on that, the replacement process is generated. Otherwise, the process of feature model re-configuration in order to mitigate the effect of failure begins. For example in the service mashup in Figure 2, if the service *DetectTextProfanity*, which works on the text extracted from that image fails, there exists no service with the same functionality to replace it. Therefore, replanning is used to find an alternate solution which does not rely on this service. Figure 3 shows the alternate BPEL process found through replanning. In this case, the structure of the process has changed such that the *DetectProfanity* service which receives an image as input and extracts profanity from it can be used for the purpose of detecting profanity in the image.

3.2 Adaptation through feature model re-configuration

The goal here is for the service mashup to degrade gracefully instead of failing completely. This way the service mashup continues to provide service with limited or alternative features. This will be accomplished by finding an alternate feature model configuration

which satisfies the structural and integrity constraints, does not rely on the failed service, and has the least loss of utility compared to the failed feature model configuration. Finding an alternate feature configuration ensures that the set of features in the alternative service mashup represents a valid combination of features. In this paper, we reduce the problem of finding the alternative feature model configuration into an optimization model.

On the one hand, the well-accepted and efficient method for finding a valid feature model configuration which satisfies feature model constraints is reduction to boolean satisfiability (SAT) problem [5]. On the other hand, minimizing the loss of utility between alternate and current feature model configuration can be seen as an optimization problem. In order to find a solution with regards to these two requirements, we reduce this problem into a *Pseudo-boolean optimization* (PBO) problem [7]. In this type of optimization, optimization is performed on a function which is defined over binary variables and whose constraints are defined as boolean formulas. This type of optimization problem has the advantage of being reducible to a SAT problem [10, 18] and efficiently solved using existing solvers. In the following, the pseudo-boolean optimization and the reduction of existing problem into pseudo-boolean optimization is formally defined.

pseudo-boolean Optimization. Assuming an array of boolean variables $X = (x_1, ..., x_n)$, a pseudo-boolean optimization problem can be formally defined as:

$$\textbf{Minimize: } f(x_1, ..., x_n) = \sum_{S \subseteq \{1,..,n\}} c_S \prod_{i \in S} x_i \tag{1}$$
$$\textbf{Subject to: } B(x_1, ..., x_n)$$

In this definition, function f is defined as the sum of weights for those subsets of variables which are all true given that $\prod_{i \in S} x_i$ would be only one if all the variables in S are true and zero otherwise. Therefore, an optimization function can be defined by specifying the weights for all subsets of boolean variables in general form. The constraints of this optimization function is a boolean function B. The solution of the pseudo-boolean optimization is an assignment to X which minimizes the value of f while making B true at the same time.

Representing a feature model configuration as an array of binary variables $C = (c_1, ..., c_n)$ where the feature corresponding to c_i would be selected in the feature model configuration if c_i is 1 and unselected if it is 0, the problem of finding an alternate feature model configuration C' with minimum loss of utility from C can be formally defined as a pseudo-boolean optimization problem as:

$$\textbf{Minimize: } loss(C, C') \tag{2}$$
$$\textbf{Subject to:}$$
$$B(c' \in C') = S(c' \in C') \wedge E(C, C') \wedge I(c' \in C')$$

In this definition, *loss* is a function which gets two configurations as an input and returns a numerical value representing the loss of utility when configuration C is replaced with C'. Furthermore, the boolean function representing the constraints of this optimization problem is defined as the conjunction of three boolean functions: $S(c' \in C')$, which is a set of predicates representing the feature model's structural and integrity constraints, $E(C, C')$, which is a set of predicates that make sure the new configuration makes valid assumptions about the input data, and $I(c' \in C')$ is made of a set of predicates which enforce the selection of those feature model configurations which do not require the inclusion of failed services for their realization. In the following, we discuss how the utility function and these boolean functions can be presented.

3.2.1 Loss of utility. Assuming that the user has requested a service mashup based on the feature model configuration C, the goal here is to find a way to measure the loss of utility if the user is provided with a service mashup with features in feature model configuration C' where utility is a quantitative value representing relative desirability of that feature model configuration. In this work, it is assumed that the utility of each feature for the user is independent of the other features which exist in the system. Assuming that such values are available through a function $U : F \rightarrow \mathbb{R}$, the loss of utility by providing a service mashup with feature model configuration C' instead of configuration C can be represented as a function $loss(C, C')$ where:

$$loss(C, C') = \sum_{i \in \{1,..,n\}} c_i(1 - c'_i)U(f_i) - (1 - c_i)c'_i U(f_i) \tag{3}$$

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

The drawback of this approach is that the utility of features is not always available. Therefore, we adopt a restricted version of $loss(C, C')$ called $distance(C, C')$ which uses only the number of features which are different in term of their selection in the new configuration as a measure for loss of utility after using an alternative configuration. The *distance* function can be used instead of the *loss* function as the goal of minimization in Equation 2 when utility of features is not available.

$$distance(C, C') = \sum_{i \in \{1,..,n\}} (1 - c_i)c'_i + c_i(1 - c'_i) \tag{4}$$

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 feature are selected or unselected. In case one of c or c' is one and the other is zero, this expression evaluates to one. Therefore, this function will be zero when the two feature model configurations are identical.

3.2.2 Feature Model Structural and Integrity Constraints. The set of features in the alternative service mashup needs to respect the constraints defined in the feature model. For example, the storage feature cannot be unselected in the feature model configuration since it is a mandatory child of the root feature. The boolean function $S(c \in C)$ would be evaluated as true when the assignment to C represents a valid feature model configuration in terms of structural and integrity constraints. Previous work has shown that

all types of structural and integrity constraints can be represented using propositional logic. For example, the optional relation between the Upload Image and Editing feature can be represented as $UploadImage \vee \neg Editing$. There exist methods for translating a feature model structure and its constraints into propositional logic. In our work, we have used the translation proposed in [4].

3.2.3 Service Mashup Precondition Constraints. The new service mashup which replaces the old service mashup cannot make new assumptions about the precondition state of its execution and needs to have the same preconditions. Therefore, feature model configurations that produce new assumptions about the preconditions of the service mashup cannot replace the failed feature model configuration. For example, adding the watermark feature to the feature model configuration containing features marked with (•) in Figure 1-A will require the condition *WatermarkRequested*(im,wt) to be true before execution, which might not necessarily be true. The boolean function $E(C, C')$ ensures that the alternative feature model configuration does not change the preconditions of the service mashup.

Considering that each feature f in the feature model is annotated with the set $\mathcal{P}(f)$ containing the preconditions that the selection of the feature will add to the precondition of the service mashup, the new configuration cannot include a feature whose $\mathcal{P}(f)$ is not a subset of $Pre(C)$. Therefore, the function $E(C, C')$ can be defined as:

$$E(C, C') = \bigwedge_{i \in \{1..n\}\ s.t.\ \mathcal{P}(f_i) \not\subseteq Pre(C)} \neg c'_i \qquad (5)$$

Assuming that $Pre(C)$ returns all of the preconditions required by service mashup realizing feature model configuration C as outlined in [2], the And operator in this equation makes sure that configuration process does not select those features which add preconditions which are not in the precondition set of service mashup realizing configuration C.

3.2.4 Service Independence Constraints. These constraints prevent the selection of those feature model configurations which rely on the failed service for their realization. Since, there is no direct mapping between features and services in our service mashup composition 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 the service failure without trying to compose them.

In order to address this issue, an incremental approach is taken. We define a set consisting of those feature model configurations which rely on the failed service for realization. In the beginning we initialize this set with the feature model configuration related to the failed service mashup. Consequently, as alternative feature models configurations are found, corresponding service mashups to these configurations are built using existing services. If building the service mashup corresponding to that feature model configuration fails, the feature model configuration is added to this set and the process of looking for alternative feature model configurations continues. The boolean function $I(c' \in C')$ is defined in such a way that it would return false if the feature model configuration corresponding to the current assignment exists in the failed feature model configuration set and true otherwise.

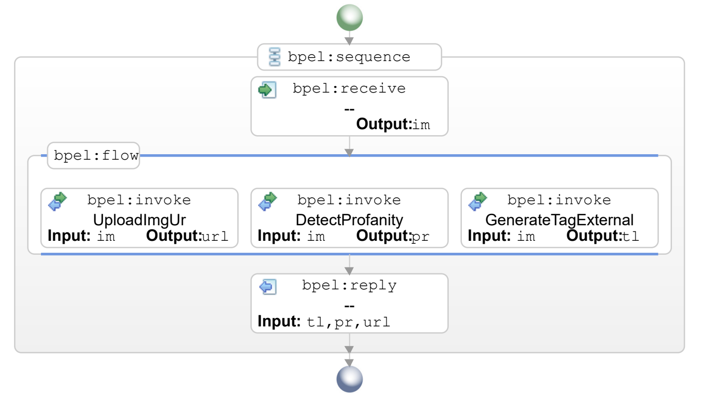

Figure 4: BPEL process visualization for an alternate solution for BPEL process in Figure 2 after adaptation by feature model re-configuration as a result of *ObjectDetection* failure.

By representation of the feature model configuration problem as a pseudo-boolean optimization problem, the alternative feature model configuration can be found using a solver. Based on this feature model configuration, the replacement service mashup is composed and executed. For example for the service mashup in Figure 2, if the service *ObjectDetection*, which extracts objects in the image fails, the service mashup cannot fully recover using existing services in the service repository. The above method results in a feature model configuration with features marked with (○) in Figure 1-A as the alternate feature model configuration which does not rely on the failed service. Figure 4 shows the alternative service mashup built based on this configuration. In the alternative feature model configuration, the Metadata-based Tagging, Editing, and Face Blur features are removed and External Tagging is added.

4 EXPERIMENTS

The proposed approach has been implemented in a tool suite called magus.online[1] using FF planner [13] for AI planning and NaPS solver [22] for pseudo-boolean optimization. Using this implementation, we performed three different sets of experiments in order to investigate the practicality of the proposed method. The experiments were performed on simulated services and feature models given that actual feature/service repositories that have different sizes do not exist. 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.

4.1 Effect of Feature Model Configuration Size on Adaptation

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 number of valid feature model configurations of the feature model increases?

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

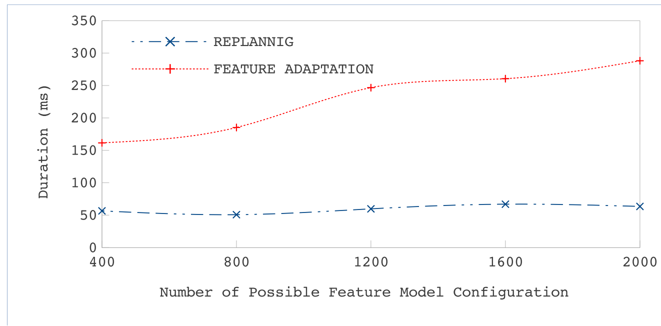

Figure 5: Replanning and Feature Adaptation Duration in terms of Feature Model Possible Configuration.

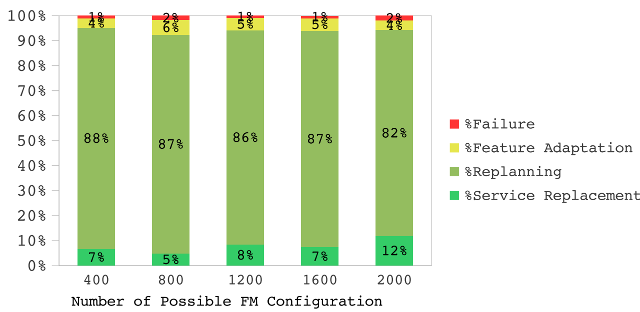

Figure 6: Distribution of Recovery and Mitigation Mechanisms after a Service Failure in terms of Possible Feature Model Configuration.

RQ 1.2 - What types of strategies, and to what extent, are used in order to address a service failure as the number of valid feature model configurations of the feature model increases?

The goal for asking the first question is to evaluate if the adaptation process is performed in a reasonable time. Furthermore, the second goal for asking this question is to investigate the applicability of the proposed method in terms of adaptation time for larger feature models. The goal of asking the second question is to first investigate if the proposed self-healing approach is capable of recovery from service failures and second to examine if the adaptation method remains feasible as the size of the feature model increases.

The experiments were designed as follows: For different service mashup families with the same service repository size but different feature model sizes, a possible feature model configuration was randomly selected and the service mashup satisfying it was composed. Then a service was randomly selected and placed in the failed state and the time to perform adaptation and the type of adaptation chosen to address the failure was recorded.

The experiment was performed for five different groups of service mashup families where feature models in the same group had the same number of features but different number of possible configurations. Each group had 10 different service mashup families which were generated as follows: We used the Betty feature model generator [23] to generate the feature models with 30 features which is the average number of features in the SPLOT repository feature models [2] and with maximum a branching factor of 10. 25 percent of the features were in Or group, 25 percent were in Alternative group and the rest of the features were split equally between Optional and Mandatory categories. According to the survey that is performed in [24], these parameters reflect structural properties of a real feature model. The number of possible configurations for the generated feature models ranged between 400 a and 2,000 which is divided between 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 composition 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 service precondition-effect annotation. 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 precondition effect annotation or a paper in the literature which reports on service precondition-effect generation parameters. Therefore, the precondition and effect distributions for the services in our case studies were used for service generation. 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 precondition and effects. [3]

For generating services, a customized service model generator was implemented in order to make sure that all possible configurations of a feature model have corresponding service mashups. In this method, the generator iterates over all possible configurations of the input feature model and makes sure required services for realizing that feature model configuration exist in the repository. For each feature model configuration, the planner is used to examine if that feature model configuration can be realized using existing services in the repository. If the feature model configuration cannot be realized using existing services, a planner is used to find the sequence of services has the highest degree of 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 mashup family can be generated using services in the service repository. [4]

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

Figure 5 shows the average time to perform re-planning and feature adaptation for feature models with different number of possible configurations. The average re-planning time was the same (60 ms) in different feature model configuration sizes. This is expected since the number of services and feature model configuration size which

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

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

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

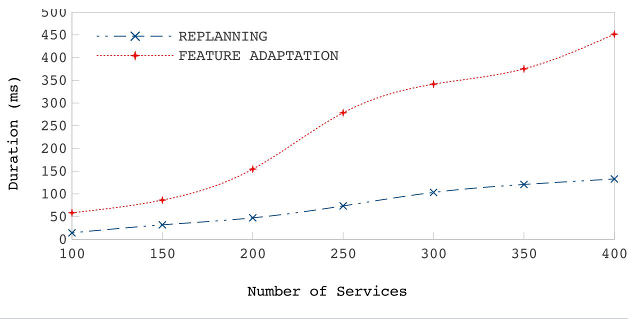

Figure 7: Replanning and Feature Adaptation Duration in terms of Service Repository Size.

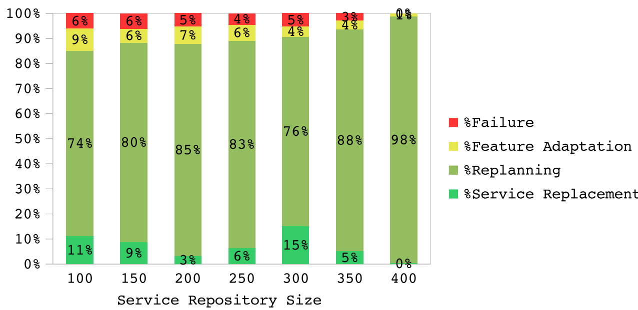

Figure 8: Distribution of Recovery and Mitigation Mechanisms after a Service Failure in terms of Service Repository Size.

affect the size of the planning problem is the same for different groups of service mashups. Conversely, the time for feature adaptation increases as the number of possible configurations increases as the problem gets more complex to solve. It should be noted that the feature adaptation time remains fast (<300 ms), and increases in a linear way.

Figure 6 shows the distribution of responses taken as service failure occurs. In our dataset of service mashup families, the adaptation approach is able to recover from failure of a service through service replacement or re-planning in 92% or more of the cases. It is also able to recover or mitigate the failure in 98% of cases and only 2% or less of service failure resulted in the failure of service mashup. It should be noted that the recovery rate for a service mashup family can be influenced by different factors of how it has been realized through services. However, the results show that this approach can recover a failed service mashup while the rate of success can differ between different service mashup families. It can also be seen from this figure that the increase in the feature model in terms of possible configurations does not affect the way the service mashup responds to failure.

4.2 Effect of Service Repository Size on Adaptation

In the second set of experiments, we focus on the effect of change in the number of services available in the service repository on the different aspects of adaptation. In our experiments, we are looking to answer two research questions:

RQ 2.1 - How does the time for performing an adaptation change as the service repository size for the service mashup increases?

RQ 2.2 - What types of strategies, and to what extent, are used in order to address a service failure as the service repository size grows?

The goal of the first question is to examine the practicality of the proposed approach in terms adaptation time for larger service mashup families. The goal of the second experiment is to study how the behaviour of the adaptation approach changes as the number of services in the service mashup family increases to see if the proposed method is still practical for failure recovery in service mashup with larger size in terms of the services.

In order to create the dataset for this experiment, we used an annotated feature model configuration from previous experiments with 2,000 possible configurations. We used the service generator from previous experiments with the same parameters but with different repository sizes ranging from 100 to 400 with an interval of 50. This range was selected based on the correlation between the number of possible configurations and the number of services in the implemented case studies using the service mashup composition approach since an actual feature/service repository is not available. This resulted in seven service mashup families. We used these service mashup families to perform the experiment.

The experiment was performed by selecting a random feature model configuration with 15±1 features for a service mashup family and composing its corresponding service mashup. A random service from the generated service mashup is selected and considered as a failed service. The failure is addressed by adaptation and the adaptation duration and its type is collected. This activity was repeated 250 times for each service mashup family in the dataset.

Figure 7 shows the average time for re-planning and feature model adaptation with different service repository sizes. Similar to the previous experiment, feature adaptation takes longer than re-planning. However, both types of adaptation remain less than 500ms for service mashups in the dataset. Furthermore, it can be seen that the time for both re-planning and feature adaptation increases in a linear manner as the number of services increases.

Figure 8 shows the distribution of the type of strategies adopted in response to service failures. It can be seen that about half of the failures that cannot be addressed with re-planning can be addressed with feature adaptation in all service mashups with different service repository sizes. Another noticeable feature of this figure is that the service replacement time does not show any trend. We speculate that running this experiment with larger number of service mashup families will result in a trend on this strategy as well.

4.3 Effect of the Number of Failures on Adaptation

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 mashup to examine the robustness of our approach. In our experiments, we are looking to answer the following research questions:

RQ 3.1 - What types of strategies, and to what extent, are used in order to address a service failure as the number of failed services increases?

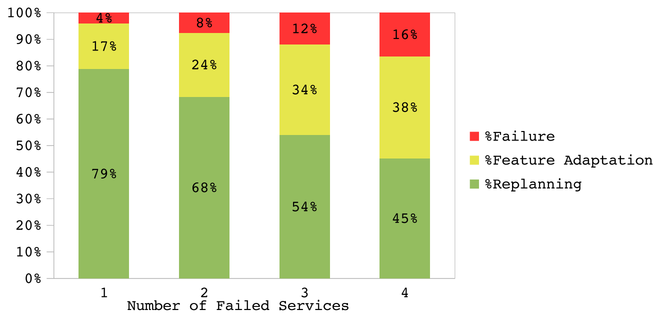

Figure 9: Distribution of Recovery and Mitigation Mechanisms after a Service Failure in terms of Number of Failed Services.

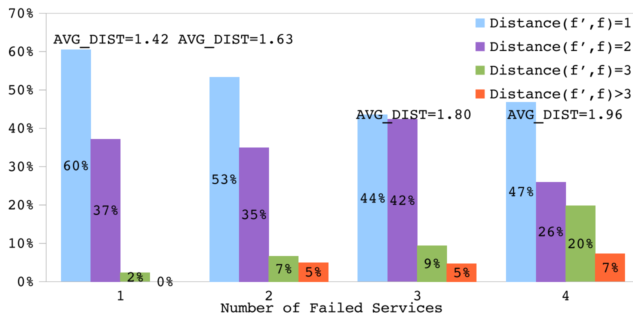

Figure 10: Distribution of Distance for Alternate Feature Model Configuration in Feature Model Adaptation in terms of Number of Failed Services.

RQ 3.2 - 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 goal of asking the first 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 goal of asking the second question is to determine if the changes in the adapted feature model is limited enough to be used as a mitigation strategy.

For generating dataset of this experiment, we created 10 service mashup 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 mashup family and the corresponding service mashup was generated. Then, a random number of services, between 1-4, were selected and set to failed status. The failures were then addressed by the proposed approach. This activity was repeated 250 times for each service mashup family.

Figure 9 shows the distribution of responses taken as service failure occurs for different number of failed services. The service replacement strategy was removed when performing this experiment since it is defined for situations when only one service has failed. As expected, the number of recovered service mashups decreases as the number of failed services increases such that more than half of the service mashups cannot recover their full functionality when four services have failed. However, more than 70% of failures can be mitigated in cases when recovery through re-planning is not possible. This reduces the number of failures to 16% when four services fail at the same time in the service mashup.

Figure 10 shows the distribution of the distance between the alternative feature model configuration and the failed feature model configuration for different number of failures. It can be seen that the average distance of the alternative feature model 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 alternative feature model configuration remains less than 2 features even when four services have failed.

5 RELATED WORK

WTE+ [14] is an approach for automated composition and adaptation of service mashups in which the requirements of the service mashup are defined by specifying the desired service mashup preconditions and effects. In the case of service failure, an alternative mashup satisfying those preconditions and effects is sought through replanning. Our proposed method differs from WTE+ in the sense that it introduces features and relates features to services. This allows the specification of the desired mashup requirements through features as well as deciding on features to address service failure when re-planning fails.

ENTIMID [16] is a middleware for developing service-based DSPLs for home automation system. In this middleware, it is assumed that each feature can be mapped to a set of atomic services which realize that feature. This allows easy dynamic reconfiguration of system at runtime by enabling/disabling corresponding services. However, this work mainly focuses on relating services and features and a systematic way for enabling context-awareness; therefore, automated feature model re-configuration is not provided by this work.

CAPucine [20] is another service-based DSPL which focuses on context-awareness. In this method, contextual information is collected by a set of sensors wrapped as components. Changes in this contextual information can trigger some rules which result in feature model configuration such as selecting or deselecting a feature. The changes in the feature model configuration is reflected on the running system using Aspect Model Weaving. This method allows developing DSPLs which can be adapted for different purposes such as self-healing applications. However, in the case of self-healing applications, the designer needs to design an adaptation strategy for all possible types of failures which can be cumbersome and can result in invalid configurations. This can primarily be due to the fact that the rules need to be designed in such a way that the outcome feature model configuration is always valid; an assumption that is difficult to satisfy in many cases.

Acher et al. [1] propose a method for feature adaptation based on context changes and use it in realizing a video surveillance system. In this method, context variability is captured using a feature model. This context feature model is mapped to the system variability feature model through a set of rules represented using propositional logic. As a change in the context modifies the configuration of context feature model, appropriate system feature configuration satisfying the relation rules is found using a SAT solver. This method

provides an effective means for linking context adaptability and feature adaptability at runtime. However, it is not the best choice for self-healing in response to service failure since feature models are not the best option for representing service availability.

Murguzur et al. [19] propose a DSPL for enabling feature adaptation in data-intensive Operation and Maintenance (O&M) analytics processes for wind farms. Their work is context-aware in in that it addresses issues of ambiguity and heterogeneity in the input data. Using this method, the analytics processes can adapt at runtime through feature model configuration as the structure of the input data changes. This work is different from our work in how we define context. In our work, the context is defined as the availability of services while the context is defined as the input data in [19].

In the Refresh approach [26] for self-healing service compositions, 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). When an alternate feature model configuration is found, the system shutdowns those components corresponding to the unselected features and launches those corresponding to the selected features using a method called *micro-booting*. In this work it is assumed there is a direct link between services and features. However, our proposed work does not make that assumption since is not always true [25]. In practice, a feature can be realized with different combination of services and creating a direct relationship between services and features may not be possible.

Our work is similar to these works in the sense that it uses feature models as the main artifact for representing variability and adaptation takes place by re-configuring it. However, there are three aspects that distinguish our work: First, making changes in the feature of the service mashup is used as the last resort in our proposed method instead of defining an adaptation strategy on the features of the system. Second, our work does not require any predefined rules for adaptation. This has two advantages in the case of responding to failure: (i) it is hard to define adaptation recovery rules for all possible service failures. (ii) adaptation strategies are likely to become incosistent [11]. Third, we do not assume any direct relationship between features and services but instead use planning for finding alternate services mashup.

6 CONCLUSION AND FUTURE WORK

In this paper, we have proposed a method for enabling self-healing service mashups in response to service unavailability. The method proposed in this paper enables service mashups to recover from failure by trying to find another service mashup with the same features. Additionally, we have proposed a method which uses feature model re-configuration to find an alternate feature model configuration which does not rely on the failed services in order to recover from failure. In this method, finding an alternate feature model configuration with the least loss of utility is represented as an optimization problem. To investigate the practicality of the method, we performed experiments, which showed that proposed adaptations can be performed in reasonable time, and do not require substantial changes to the feature model configuration, and can produce partial solutions when complete recovery is not possible.

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. 2011. Modeling Variability from Requirements to Runtime. In *International Conference on Engineering of Complex Computer Systems*. 77.

[2] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. 2016. Automated Composition of Service Mashups Through Software Product Line Engineering. In *International Conference on Software Reuse*. Springer, 20–38.

[3] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. 2017. Dynamic Software Product Line Engineering: A Reference Framework. *IJSEKE* 27, 1 (2017), 1–44.

[4] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines (SPLC 2005)*. Springer, 7–20.

[5] David Benavides, Sergio Segura, and Antonio Ruiz-Cortes. 2010. Automated analysis of feature models 20 years later: A literature review. *Information Systems* 35, 6 (2010), 615–636.

[6] Djamal Benslimane, Schahram Dustdar, and Amit Sheth. 2008. Services Mashups: The New Generation of Web Applications. *IEEE Internet Computing* 5 (2008), 13–15.

[7] Endre Boros and Peter L. Hammer. 2002. Pseudo-boolean optimization. *Discrete applied mathematics* 123, 1 (2002), 155–225.

[8] Jan Bosch and Rafael Capilla. 2012. Dynamic Variability in Software-Intensive Embedded System Families. *Dynamic Variability in Software-Intensive Embedded System Families* 45, 10 (2012), 28–35. DOI: http://dx.doi.org/10.1109/MC.2012.287

[9] Florian Daniel, Fabio Casati, Boualem Benatallah, and Ming-Chien Shan. 2009. Hosted universal composition: Models, languages and infrastructure in mashart. In *International Conference on Conceptual Modeling*. Springer, 428–443.

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

[11] Franck Fleurey and Arnor Solberg. 2009. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *MODELS*. Springer, 606–621.

[12] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2008. Dynamic software product lines. *Computer* 41, 4 (2008), 93–95.

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

[14] Anna Hristoskova, Bruno Volckaert, and Filip De Turck. 2013. The WTE framework: automated construction and runtime adaptation of service mashups. *Automated Software Engineering* 20, 4 (2013), 499–542.

[15] Hui Huang, Xueguang Chen, and Zhiwu Wang. 2015. Failure recovery in distributed model composition with intelligent assistance. *Information Systems Frontiers* 17, 3 (2015), 673–689.

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

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

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

[19] Aitor Murguzur, Rafael Capilla, Salvador Trujillo, Oscar Ortiz, and Roberto E. Lopez-Herrejon. 2014. Context variability modeling for runtime configuration of service-based dynamic software product lines. In *SPLC*. ACM, 2–9.

[20] Carlos Parra, Xavier Blanc, and Laurence Duchien. 2009. Context awareness for dynamic service-oriented product lines. In *SPLC*. CMU, 131–140.

[21] Klaus Pohl, Gunter Bockle, and Frank Van Der Linden. 2005. *Software Product line Engineering: Foundations, Principles and Techniques*. Vol. 10. Springer.

[22] Masahiko Sakai and Hidetomo Nabeshima. 2015. Construction of an ROBDD for a PB-Constraint in Band Form and Related Techniques for PB-Solvers. *IEICE Transactions on Information and Systems* 98, 6 (2015), 1121–1127.

[23] Sergio Segura, Jose A. Galindo, David Benavides, Jose A. Parejo, and Antonio Ruiz-Cortes. 2012. BeTTy: benchmarking and testing on the automated analysis of feature models. In *Workshop on Variability Modeling of Software-Intensive Systems*. ACM, 63–71.

[24] Thomas Thum, Don Batory, and Christian Kastner. 2009. Reasoning about edits to feature models. In *ICSE*. IEEE, 254–264.

[25] Pablo Trinidad, Antonio Ruiz-Cortes, Joaquin Pena, and David Benavides. 2007. Mapping feature models onto component models to build dynamic software product lines. In *International Workshop on DSPL at (SPLC 2007)*.

[26] Jules White, Harrison D. Strowd, and Douglas C. Schmidt. 2009. Creating self-healing service compositions with feature models and microrebooting. *Journal of Business Process Integration and Management* 4, 1 (2009), 35–46.