Automated Composition of Service Mashups Through Software Product Line Engineering

Mahdi Bashari[1(✉)], Ebrahim Bagheri[2], and Weichang Du[1]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, Canada
{mbashari,wdu}@unb.ca
[2] Department of Electrical and Computer Engineering,
Ryerson University, Toronto, Canada
bagheri@ryerson.ca

Abstract. The growing number of online resources, including data and services, has motivated both researchers and practitioners to provide methods and tools for non-expert end-users to create desirable applications by putting these resources together leading to the so called *mashups*. In this paper, we focus on a class of mashups referred to as service mashups. A service mashup is built from existing services such that the developed service mashup offers added-value through new functionalities. We propose an approach which adopts concepts from software product line engineering and automated AI planning to support the automated composition of service mashups. One of the advantages of our work is that it allows non-experts to *build and optimize* desired mashups with little knowledge of service composition. We report on the results of the experimentation that we have performed which support the practicality and scalability of our proposed work.

Keywords: Service mashups · Feature model · Software product lines · Automated composition · Planning · Workflow optimization

1 Introduction

More and more companies are now making their application services publicly available to non-affiliated developers through online platforms such as ProgrammableWeb. These services can be accessed through well-defined RESTful APIs. Many of these services are highly reliable and provide functionalities that cannot be otherwise easily implemented by smaller software development companies or end-users such as Google Maps, Zazzle and Paypal, just to name a few. Therefore, the popularity of such publicly available online services and the ease of adoption of their REST-based SOA architectures have motivated researchers and practitioners to develop tools and methods which allow end-users to seamlessly build new services by composing existing APIs [4]. Such services are often known as *service mashups*. A service mashup is a service which is composed of a number of other services and provides added-value through new functionalities.

G.M. Kapitsaki and E. Santana de Almeida (Eds.): ICSR 2016, LNCS 9679, pp. 20–38, 2016.
DOI: 10.1007/978-3-319-35122-3_2

The added value of service mashups is through the emergence of newer functional capabilities that were not available prior to the integration of the already existing services.

There is considerable amount of research on semi-automatic and automatic methods for composing service mashups [10]. Most of these approaches assume that the end-user is familiar with the specifics of each and every instance of services' execution and invocation criteria, i.e., their pre-requisites, input and output types and other types of execution requirements. However, when considering the fact that the objective of such work is to enable automated runtime selection and composition of services from the available service possibilities with minimal user intervention and high-availability, this becomes a noticeable shortcoming. It can prevent non-expert users who do not have the required knowledge to benefit from and use such approaches.

In order to address this issue, we follow an intuitive approach to separate the non-expert end-users from the complexities of the services by using concepts from Software Product Lines (SPL). It has already been argued in the literature that while end-users might have difficulty understanding the underlying specifics of services, they are more comfortable when dealing with higher-level representations of functionality expressed through SPL features [13]. A *feature* is often defined as an incremental prominent or distinctive user-visible functionality of a software and is therefore quite understandable for the end-users. In other words, while the end-user may not know which specific services are collectively needed to satisfy her requirements, she would know which user-visible functionalities are expected from the final product.

The integration of services and features have already been extensively investigated in the literature [13]. We specifically base our work on the model proposed by Lee and Kotonya where features are operationalized through atomic or composite services [13]. In this model, two distinct lifecycle phases are introduced: *(i)* domain engineering phase: during which appropriate services that can operationalize features are identified, and are connected to their corresponding features, and *(ii)* application engineering phase: during which the end-users select their desired features through which the right services are identified. Our work is positioned within the application engineering phase of this model and provides mechanisms for automatically composing and optimizing a service mashup based on the user-specified feature requirements.

In this paper, we provide the following concrete contributions:

- We propose an AI planning based method for automated service mashup composition which operates based on feature model configurations as the main input model for specifying user requirements and generates a WS-BPEL workflow that satisfies those requirements.
- We further propose a method for optimizing the created WS-BPEL workflow by considering the concepts of safeness and threat from the planning domain in order to inject parallelism into the generated workflow and improve its execution efficiency (e.g. reduce execution time).

The rest of this paper is organized as follows: Sect. 2 will cover the required background information and the problem statement. In Sect. 3, we will describe the details of the proposed approach. Section 4 will then provide the details of the experiments and the insights gained from them. The related work is covered in Sect. 5 and finally, the paper is concluded in Sect. 6.

2 Problem Statement and Background

The objective of our work is to enable non-expert end-users to automatically *optimially compose* publicly available services in order to satisfy the requirements without being concerned with the technical details of service composition. To achieve this objective, we rely on the integration of *services* and software product line *features*. As mentioned earlier, researchers such as Lee and Kotonya [13] have already explored and concretely investigated how services and features can be integrated. There is ample literature that builds on a two-phase lifecycle that integrates services and features in its first phase and then, in the second phase, uses the integrated model to derive a product that satisfies the end-users' desired feature selections [13]. The derived product will then be operationalized by the services that are connected to the selected features. In this paper, we assume that the first domain engineering phase of the lifecycle, i.e., the connection between services and features, has already been completed using one of the established methods in the literature [13]. Our focus will therefore be to systematically support the application engineering phase of the lifecycle. Current automated service composition methods work on inputs such as OWL-S service descriptions [10], temporal logic [5], or other languages, which are used to specify the characteristics of the desired composed service mashup. However, we are interested in an input specification model *abstract* enough to be used by non-expert end-users to specify their requirements and an output that would be *concrete* enough to be directly executable. For this purpose we use feature models as the input specification model and generate the final outcome of the composed service mashup in WS-BPEL.

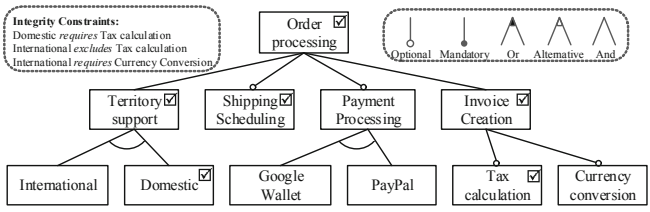

Fig. 1. A sample feature model for an order processing mashup family.

2.1 Feature Models

Feature models are among the widely used variability modeling tools used in Software Product Line Engineering (SPLE). A feature model provides a hierarchical tree structure that represents the organization of and the relation between the features. Features can be structurally related to each other through optional, mandatory, Xor-, Or-, or And-group relations. These relations express the possible variabilities of the product family. Feature models also represent cross-cutting variations using integrity constraints. The use of feature models has the advantage of being understandable while having the power to represent complex variability of a family and therefore is usually used as a shared model between users and system developers in software product line engineering [14].

Figure 1 depicts a feature model for a product family that processes a purchase request and creates an invoice in different ways. The product family represented through the 'Order Processing' root node has four sub-features, namely invoice creation, shipping scheduling, payment processing, and territory support, where shipping scheduling and payment processing features are optional. Territory support sub-features are mutually exclusive. Furthermore, the selection of the 'international' feature prevents the selection of the 'tax calculation' feature and requires the selection of the 'currency conversion' due to the integrity constraints.

A feature model *configuration* is a subset of the features in a feature model which satisfies the structural and integrity constraints, and represents a viable instance of the family. Prior work has shown that a feature model configuration can be used as an effective tool for representing the end-users' requirements [14]. For example, the selection of features marked with a checkbox in Figure 1 represents a valid feature model configuration that can also be considered to be the requirements expressed by an end-user.

2.2 Business Process Execution Language (BPEL)

The Web Service Business Process Execution Language (WS-BPEL), commonly interchangeable with BPEL, is a well-known standard for the specification and execution of service-oriented business processes. In WS-BPEL, processes are built using WSDL-SOAP services and processes themselves are exposed as WSDL-SOAP web services. Control flows in WS-BPEL are expressed by structured activities and data is passed between services by sending variables as parameters. Figure 2 represents a graphical representation for a WS-BPEL code for the possible realization of the feature model configuration in Fig. 1 where features marked with checkboxes are selected. The general process for a service composition is made of hierarchical organization of activities using $<flow>$ and $<sequence>$ tags. The activities in $<flow>$ can be executed in any order or in parallel while activities in a $<sequence>$ tag should be executed in order. The synchronization between activities in a $<flow>$ tag can be done using $<link>$ tags which has been shown by yellow arrows in Fig. 2.

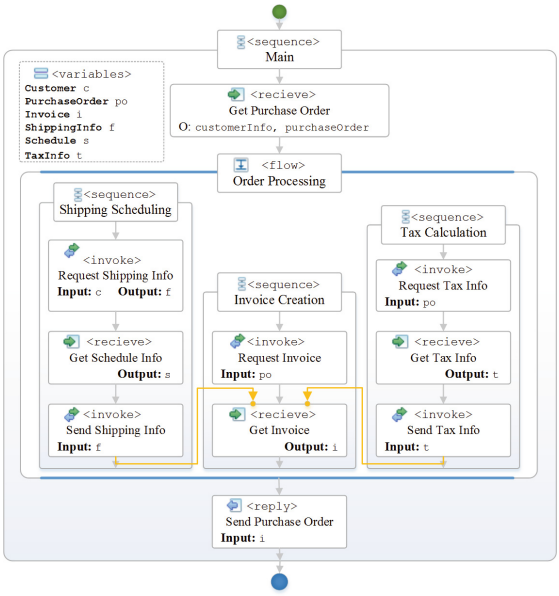

Fig. 2. Graphical representation of a possible WS-BPEL process for order processing.

The atomic activities in WS-BPEL are made of service invocations, receiving a callback for a service invocation, and a number of WS-BPEL actions or control activities which will not be considered in this paper for the sake of simplicity and without loss of generality. Each service invocation may receive some variables as input and may return one or more outputs. WS-BPEL code can be readily executed using existing WS-BPEL engines.

In the next section, we will describe our proposed automated mashup composition and workflow optimization method which receives the end-users' requirements through a feature model configuration process and automatically builds a fully executable WS-BPEL process to serve as the target service mashup.

3 Proposed Approach

In our work, an input feature model configuration serves as the end-users' requirements and it is realistically assumed that the features of the feature model configuration have already been connected to relevant services during the domain engineering phase [13]. We refer to the feature model configuration and the services connected to the features as the *domain model*. The objective is to generate a fully executable WS-BPEL process based on the domain model. Our proposed method first creates a workflow model that consists of all the features present in the domain model through an AI planning problem. The obtained workflow is then optimized and converted into WS-BPEL. In the following, we first formally define the domain model.

3.1 Domain Model Specification

We define the *domain model* to consist of five sub-models, namely feature model, service model, context model, service annotations, and feature model annotations. We start by formally defining a feature model configuration and a workflow and then define the models that connect these two together.

In order to define feature model configuration, we first need to define the feature model. A feature model can be formally defined as:

Definition 1 (Feature model). A feature model is a tuple $fm = (F, \mathcal{P}, \mathcal{F}_O, \mathcal{F}_M, \mathcal{F}_{IOR}, \mathcal{F}_{XOR}, \mathcal{F}_{req}, \mathcal{F}_{exc})$ where

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

Consequently, a feature model configuration is defined as follows:

Definition 2 (Feature model configuration). A feature model configuration is a set $C \subseteq F$ where

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

In order to operationalize a feature model configuration in a SOA model, the orchestration of features implemented using services needs to be implemented in a workflow. A workflow specifies the sequence of interactions between the services. Our objective is to first develop a workflow from a feature model configuration and then convert that into WS-BPEL. We define a workflow based on a service specification as:

Definition 3 (Service). A service specification $s = (I, O, O_c)$ is a triple where

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

Definition 4 (Workflow). A workflow is a triple $w = (E, N, \mathcal{E})$ where

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

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

Definition 5 (Context model). A context model is a triple $c = (c_T, c_E, S)$ where

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

In our context model definition, *context entities* are similar to object instances passed between functions, and *context types* are used for strictly specifying entity types. Furthermore, the context model also consists of the *context state*, which is defined by *facts*. Facts can express the relationship between zero or more context entities. Let us elaborate on this using Fig. 3. In this example, c and po are two context entities, which are of customer and purchase order types, respectively. Furthermore, the fact $ordered(c, po)$ expresses that customer c has ordered the purchase order po. This fact is represented using fact type *ordered* which relates an entity of type customer to an entity of type purchase order. We will explain in the following how the context model information will be used to annotate features.

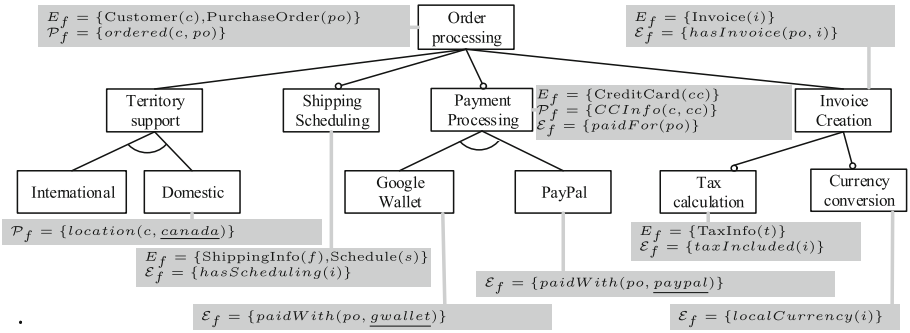

Fig. 3. An annoated feature model for the order processing family.

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

Definition 6 (Feature model annotation). The annotation for feature model fm is a function $\mathcal{A}_{FM}$ which maps each feature f in the feature model to a triple $(E_f, \mathcal{P}_f, \mathcal{E}_f)$ where

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

Figure 3 shows the annotations for our order processing feature model. As seen in the figure, for each feature, $E_f, \mathcal{P}_f, \mathcal{E}_f$ are defined as needed. For instance, the figure shows that for the 'Invoice Creation' feature to be included in the goal service mashup, a context entity i of type *Invoice* needs to be present in the context model. Furthermore, when the service mashup consisting of the 'Invoice Creation' feature is executed, the fact $hasInvoice(po, i)$ will become true as an effect, which means purchase order entity po will have an invoice entity i.

In addition to feature model annotations, we also annotate the services in a similar vein. The annotation of services with pre-conditions and post-conditions (effects) has been already widely used in the literature [10] and we adopt a similar strategy.

Definition 7 (Service annotation). A service annotation for service s is a tuple $\mathcal{A}_s = (\mathcal{P}_I, \mathcal{Q}_I, \mathcal{R}_I, \mathcal{P}_C, \mathcal{Q}_C, \mathcal{R}_C)$ where

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

For example in the service *Request Shipping Info* in Fig. 2, assuming the input $customerInfo$ is of type customer, the output $shippingInfo$ is of type shippingInfo, and the callback output $shippingSchedule$ is of type schedule in the context model, one could define the annotations for this service as $\mathcal{Q}_I = \{hasShippingInfo(customerInfo, shippingInfo)\}$, $\mathcal{Q}_C = \{hasShippingSchedule\ (customerInfo, shippingSchedule)\}$, and the other annotation sets would be empty. This annotation means after the invocation of this service the value of the output would be the shipping information for the input customer and after receiving the callback the value of the callback output would be the shipping schedule for the input customer.

In our model, the feature and service annotations serve as a bridge between the feature and service spaces and allow us to automatically compose a service mashup based on the end-users' feature selections.

Problem Statement. Given a context model type c_T, a feature model fm, a feature model configuration C, a feature model annotation $\mathcal{A}_{FM}$, a set of services S, and their corresponding annotations $\mathcal{A}_S$, the goal is to find a workflow w using services in S which satisfies the requirements of feature model configuration C.

3.2 Proposed Solution

We propose to formalize the above problem statement as a planning problem and provide a solution through AI planning. The AI planning model would be concretely defined by the initial context state as the starting point of the planner and the expected context state as the goal of the planner. Therefore, we need to formalize how the initial context state, expected goal state and the service invocations can be defined with an AI planning context to generate a workflow.

We adopt the widely used STRIPS planning specification model to provide our problem formalization, which can easily be converted to a Planning Domain Definition Language (PDDL) model. A planning problem in STRIPS [7] can be defined as below:

Definition 8 (Planning problem). A planning problem is a triple $p = (S_{initial}, S_{goal}, A)$ where:

- $S_{initial}, S_{goal}$ are the initial and goal states. These states are represented by a set of atomic facts,

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

Definition 9 (Planning problem solution). Sequence $s = <a_1, ..., a_i>$ is a solution to planing problem $p = (S_{initial}, S_{goal}, A)$ if

- a_1 is applicable on state $S_{initial}$;
- for each $1 < j \leq i$ action a_j is applicable in state S which has been resulted by consecutive application of action $a_1, ..., a_{j-1}$ on the initial state $S_{initial}$;
- consecutive application of actions $a_1, ..., a_i$ on initial state $S_{initial}$ will result in a state S such that $S_{goal} \subseteq S$.

In the proposed method, we formalize the problem as a method for finding a workflow expressed through a sequence of service invocations and callbacks, which results in the expected context state and satisfies the requirements expressed in the configured feature model.

Generating Initial and Goal States. Initial and goal states of the planner are built by aggregating the annotations of the feature model configuration, which represents the end-users' requirements. For a feature model configuration C, initial and goal states for planning problem $p = (S_{initial}, S_{goal}, A)$ would be:

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

Generating Actions. In our planning model, actions are considered to be the service operations that can be executed in the final service mashup. These actions are invocations of different services or receiving callbacks. Each of these actions affects the context state.

- **Invocation.** An invocation of service $s = (I, O, O_c)$ with annotation $\mathcal{A}_s = (\mathcal{P}_I, \mathcal{Q}_I, \mathcal{R}_I, \mathcal{P}_C, \mathcal{Q}_C, \mathcal{R}_C)$ can be defined as an action $a_{invoke}(s) = (I, F_{pre}, F_{add}, F_{del})$ where
  - input of the action is $I = I \cup O \cup O_c$
  - $F_{pre} = \mathcal{P}_I$
  - $F_{add} = \mathcal{Q}_I$ and a predicate showing that service s callback is pending ($callbackPending(s)$) if it has a callback
  - $F_{del} = \mathcal{R}_I$

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

Algorithm 1. Pseudo-code for workflow optimization.

- **Callback** is of service $s = (I, O, O_c)$ with annotation $\mathcal{A}_s = (\mathcal{P}_I, \mathcal{Q}_I, \mathcal{R}_I, \mathcal{P}_C, \mathcal{Q}_C, \mathcal{R}_C)$ can be defined as an action $a_{callback}(s) = (I, F_{pre}, F_{add}, F_{del})$ where
  - input of the action is $I = I \cup O \cup O_c$
  - $F_{pre} = \mathcal{P}_C \cup \{callbackPending(s)\}$
  - $F_{add} = \mathcal{Q}_C$
  - $F_{del} = \mathcal{R}_C \cup \{callbackPending(s)\}$

Workflow Creation. Now that the planning goal and planning problem domain are concretely defined, a planner can be used in order to find a solution for the planning problem. The solution will be a sequence of actions which takes us from the initial context state to the expected context state. Based on the solution of the above planning problem $s = <a_1, ..., a_i>$, a workflow $w = (E, N, \mathcal{E})$ can be built where:

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

Workflow Optimization. Although the generated workflow can be used to generate WS-BPEL code, given that the AI planners produce strictly sequential plans, the generated workflow would not benefit from potentially more efficient and valid plans which use parallel execution of operations when possible. Using parallelism in a service workflow can significantly affect the efficiency of the

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

Algorithm 2. Pseudo-code for examining safeness of a workflow.

composed service [20]. Therefore, once a plan is generated by the AI planner, we take an additional step to optimize the workflow.

Workflow optimization can be performed by consecutive removal of the edges in the workflow which do not affect the *safeness* [15] of the workflow. The details of our method for optimization has been shown in Algorithm 1. In the main loop in the algorithm (Lines 2–13), the edges are removed consecutively until the termination condition (Line 13) is met. In each iteration of the loop, each edge in the workflow is examined (Line 4) to see whether the workflow stays safe even after the removal of that edge or not (Line 8). If so, the edge is added a set W (Line 9). The new workflow after removal of an edge would be a revised workflow which would not include the removed edge but instead new edges are added to preserve the connectivity of the workflow. This is done by adding edges from the start node of the removed edge to the immediate nodes after the end node of the removed edge and similarly the immediate nodes before the start node and the end node of the removed edge (Lines 5–6). This ensures that the order of execution for the nodes before and after stay the same. After all edges are examined, the best workflow is selected from the set W and the current workflow is replaced by that workflow (Line 12).

The definition of SELECT and TERMINATIONCONDITION depends on the optimization method which has been selected. The definition for SAFE which is responsible for examining the safeness of a workflow has been defined in Algorithm 2. The definition of this function has been inspired by the safeness condition in partial order planning [15]. In this function, the main loop iterates over all operation nodes of the workflow (Lines 2–16) and its immediate inner loop iterates over all facts that is required to be true as the precondition of the node (Lines 3–15). For each precondition fact p of each node n, this algorithm iterates

1: **function** ThreatExists(workflow w, node n, node n',fact p)
2: **for all** $n'' \in N$ **do**
3: **if** $\neg$After(w, n'', n') or $\neg$After(w, n, n'') and $p \in \mathcal{R}(n'')$ **then**
4: **return** *true*
5: **end if**
6: **end for**
7: **return** *false*

Algorithm 3. Pseudo-code for examining existence of threat to a causal link.

over all the nodes in the workflow (Lines 5–11) in order to find an operation node n' which makes that fact true and is executed before node n (Line 6). The relation between node n' and n is called causal link for p. If such a node is found, it is examined if a *threat* to that causal link exists (Line 7). If there is no threat to the causal link between two nodes, a safe causal link has been found (Line 8). If there exists no safe causal link for a precondition fact of a node (Line 12), the workflow is not safe.

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

Although the optimization process keeps the workflow safe, it does not ensure that the workflow has the same preconditions and effects. A small modification can be done in the input workflow in order to ensure that workflow preconditions and effects remain the same during and after the optimization. This modification adds a new start operation node with no precondition and workflow preconditions as the effects to the beginning of the workflow and an end operation node with no effect and with expected effects as the preconditions to the end of the workflow. Considering that the optimization will not affect preconditions, it can be easily proven that if the start and end nodes are removed from the workflow after optimization, it will satisfy the expected preconditions and effects.

In order to make the derived workflow executable, it needs to be converted into WS-BPEL. In WS-BPEL, each flow is defined by a <*process*> tag which is made of the <*variables*> tag and a set of actions organized with <*sequence*> and <*flow*> tags. Actions in the flow tag can be executed in parallel while actions in the sequence tag should be executed sequentially. Often more than one WS-BPEL code can satisfy user's goals. Here, we adopt the method used in [18] for creating an efficient WS-BPEL code from the created workflow. This method takes as input a workflow represented as a directed graph and generates an efficient WS-BPEL representation.

4 Experiments

In order to perform experiments, we have developed a fully integrated toolset that supports our proposed approach. In our implementation, we have used OWL as the representation language for the context model as suggested in [10], OWL-S for representing services and their annotations [9], and SA-FMDL format for representing the feature model and its annotations [2]. For planning, the FF planner [9], which is a fast PDDL planner is used and for optimization a greedy implementation is used that chooses an action with the best immediate gain. Our experiments were performed on a machine with Intel Core i5 2.5 GHZ CPU, 6 GB of RAM, Ubuntu 14.04, Java Runtime Environment v1.8.

4.1 Workflow Generation

The main focus of our experiments with regards to workflow generation is the assessment of the scalability of the proposed method in terms of its running time. We evaluate the efficiency of the method from two perspectives:

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

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

Context Model: We have developed an OWL ontology for the context model with 30 entity types and 600 fact types. This context model is used to annotate the services and the feature model.

Services and Their Annotations: In order to generate the services and their annotations, we developed a random OWL-S service description generator which creates service description with inputs, outputs, precondition, and effects randomly picked from our context model. This OWL-S service description generator is highly customizable with different service model characteristics (e.g. number of inputs, outputs, precondition, and effects). Three service repository sets have been created where services in the repositories of different sets have different numbers of precondition and effects. In our experiments, we used 3, 6, and 9 as the number of preconditions and effects. Each of these repository sets has 10 different repositories of sizes between 1,000 to 10,000. Totally, 30 different service repositories have been created.

Feature Model and Its Annotations: We used the SPLOT feature model generator to generate a feature model with 1,000 features. In order to annotate this feature model, a customized feature model annotation generator is developed which randomly picks annotations from the context model and assigns them

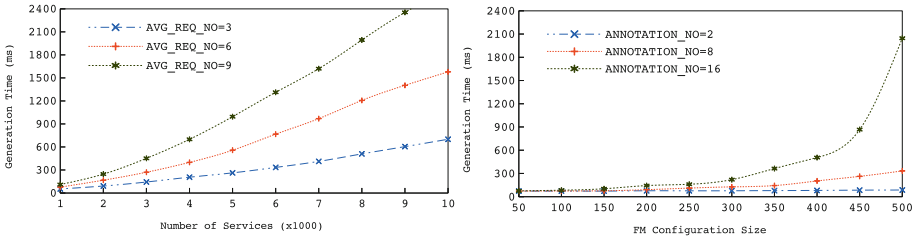

Fig. 4. Workflow generation time in terms of service repository and FM conf. size.

to the features of the feature model. Using this annotation generator, three different annotation sets were created for the feature model where the number of annotations for each feature was 2, 8 and 16. In the first experiment, a feature model configuration with 50 features is selected and the time to generate the workflow using service repositories of different sizes is measured. This operation is done repeatedly 20 times with different feature model configurations of the same size and the average time for generating the workflow is calculated. This experiment is repeated for all three repository sets. Figure 4 (left) shows how the workflow generation time increases with the increase in the size of the service repository. As it can be seen from the figure, the increase in time is linear and does not significantly increase with the increase in the number of services in the repository and remains practical (around 2.4 s for 9,000 services).

In the second experiment, the service repository with 1,000 services and an average sum of precondition and effects of 6 is selected. In this setting, the time for generating workflows for feature model configurations of different sizes is measured. The feature model configuration is generated by a tool which gets a feature model and desired number of features in the configuration and returns a random valid feature model configuration with that size. For each configuration size, 20 different configurations is generated. For each number of annotations, the average time required for generating the workflow is calculated for different configurations. Figure 4 (right) shows the average workflow generation time with different feature model configuration sizes for different number of annotations. As seen in the figure, the generation time remains linear for various configuration sizes when the number of annotations are 2 and 8 per feature. However, when the number of annotations are increased to 16, the generation time becomes exponential and shows rapid increase. It is important to note that (i) even with the increase, the time is manageable for practical purposes, i.e., 2 s for 1,000 services and 500 requirements. (ii) Literature suggests that the number of annotations is typically in the range of 5–6 annotations per feature [1], in which case, the performance of the generation algorithm is linear.

4.2 Workflow Optimization

The focus of the second set of experiments is on the investigation of the scalability of the optimization method. We explore the optimization method scalability when

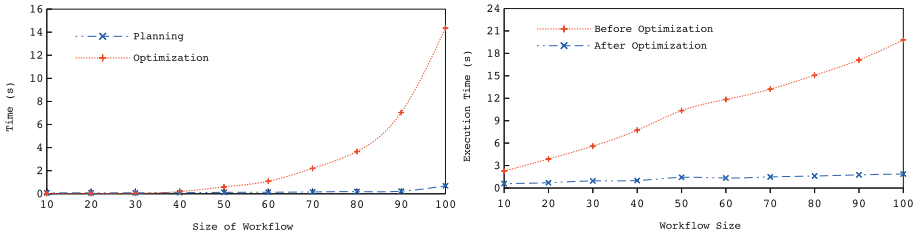

Fig. 5. Workflow optimization and execution time in terms of workflow size.

the size of workflow increases. In addition, we explore whether the optimization method is able to decrease the time-to-completion of the service mashup.

- **Experiment 2.1 (Optimization scalability in terms of workflow size):** How does the workflow optimization time increase with the increase in the size of workflow (in terms of growth in the number of workflow nodes)?
- **Experiment 2.2 (Effectiveness of the optimization in terms service mashup time-to-completion):** How much does the time-to-completion of a service mashup is decreased as a result of the optimization?

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

For the first experiment, 20 different configurations in each workflow size category is randomly selected and the average time for workflow generation and optimization is calculated. Figure 5 shows how workflow generation and optimization time increases as the size of workflow grows. This shows that the workflow optimization method is considerably slower than the planning method. However, given the fact that the optimization method is only a one time task, its benefits in terms of reducing the time-to-completion is noticeable.

In the second set of experiments, the objective is to measure whether the optimization method has been to generate workflows that have a lower time-to-completion (execution time) or not. For this purpose, the time-to-completion of the generated workflows were calculated both before and after the optimization. Figure 5 shows the result of the optimization. As seen in the figure, the time-to-completion of a workflow increases as the size of the workflow increases. However, the optimization method has been able to maximize parallelism in the workflow such that there is no noticeable growth with the increase in the workflow size. For instance, for a workflow with 100 activities, which on average take 20 s prior to optimization, the optimization method has been able to reduce the time-to-completion to 1 s.

5 Related Work

Our work is positioned among considerable other research on service composition using AI planning methods. Given the fact that planners usually take initial and

goal states as the way to define the planning problem, adopting this approach for modeling the expected outcome of a service composition is quite intuitive. Therefore, many existing approaches specify the expected outcome using a planner input language or a model that is easily convertible to a planner input [10,12]. For example, in [12], an XML dialect of PDDL is used to define the expected service specification. However, specification of requirements in those languages requires expert knowledge. In order to facilitate the design of service mashups in some approaches, the concrete service mashup is generated from the abstract process created by the user using some GUI interface. For example in [16], users drag and drop required components of their service mashup into a canvas and create the flow by connecting these components using arcs where this process is facilitated using semantic annotations for components. Such approaches still rely on the users for designing the logic of the service interactions. Another way used for specifying service requirements is through natural language specifications. For example in [8], an approach is proposed where a composite service is created based on a request in natural language using semantic annotation of the components. Although natural language seems an easy to use method of specifying requirements, it does not provide the user with a tangible model of functionalities which makes it confusing to use and unreliable. Feature models provide a tangible way to represent functionalities and have been used to represent service families. However, existing automated approaches only suggest methods which customize the services [1,3]. For example, Baresi et al. [3] use aspect-orientation in WS-BPEL to activate/deactivate aspects in WS-BPEL code of the service composition in order to customize it.

WS-BPEL allows sequential as well as parallel invocation of services. However, most AI planning methods come up with total-ordered sequential composition of services [20]. For example in [6], a planner is used to find the goal service composition which is sequential although WS-BPEL is used to represent the composition. Some of the other automated service composition methods generate compositions which take advantage of parallelism [11,20]. For example in [20], service composition is modeled as a tree search problem where the goal is to find a service composition with maximum parallelization. In another example [11], the service composition problem is modeled as a sub-graph search in a service dependency graph where the goal is to find a composition which satisfies its functional requirements as well as optimizing different quality attributes such as parallelism. However, enabling parallelisim is embedded in the composition process of these methods. In order to compose services with parallel execution, [19] suggests that partial-order planning methods need to be used. However, none of the existing service composition methods use partial-order planning because existing partial-order planners are significantly less efficient than total-order planners [17]. We suggest that enabling parallelism in the workflow can be viewed as an optimization problem. The idea of optimizing a total-order plan in order to take advantage of parallelism has been explored in the planning area [21]. However, it has not been used in the context of service composition. Our approach uses the ideas from the planning domain to propose an optimization model where different optimization methods can be used in order to enable parallel execution of operations in a workflow.

6 Conclusion

In this paper, we propose a method for the automated composition of service mashups. The service mashup composition process is operationalized through a novel approach that combines the modeling power of software product line feature models with AI planning techniques. The novelty of our work is in that end-user requirements are expressed as feature model compositions, which have been shown to be understandable by end-users. We automatically transform the feature model composition into a viable executable workflow through the mapping of the feature space into the AI planning domain. Given the fact that AI planning techniques only generate strictly sequential plans, we further develop an algorithm to optimize the developed workflow through the introduction of parallelism. The final outcome of our approach is an optimized executable business process represented in WS-BPEL format. Through our experiments we have shown that our work is scalable and is also able to efficiently produce workflows that are optimized using parallelism.

References

1. Asadi, M., Mohabbati, B., Groner, G., Gasevic, D.: Development and validation of customized process models. J. Syst. Softw. **96**, 73–92 (2014)
2. Bagheri, E., Asadi, M., Ensan, F., Gasevic, D., Mohabbati, B.: Bringing semantics to feature models with SAFMDL. In: Proceedings of CASCON 2011, pp. 287–300. IBM Corporation (2011)
3. Baresi, L., Guinea, S., Pasquale, L.: Service-oriented dynamic software product lines. Computer **45**(10), 42 (2012)
4. Benslimane, D., Dustdar, S., Sheth, A.: Services mashups: the new generation of web applications. IEEE Internet Comput. **5**, 13–15 (2008)
5. Bertoli, P., Pistore, M., Traverso, P.: Automated composition of web services via planning in asynchronous domains. Artif. Intell. **174**(3), 316–361 (2010)
6. Chafle, G., Das, G., Dasgupta, K., Kumar, A., Mittal, S., Mukherjea, S., Srivastava, B.: An integrated development environment for web service composition. In: ICWS 2007, pp. 839–847. IEEE (2007)
7. Fikes, R.E., Nilsson, N.J.: Strips: a new approach to the application of theorem proving to problem solving. Artif. Intell. **2**(3), 189–208 (1972)
8. Fujii, K., Suda, T.: Semantics-based dynamic web service composition. Int. J. Coop. Inf. Syst. **15**(03), 293–324 (2006)
9. Hoffmann, J., Nebel, B.: The FF planning system: fast plan generation through heuristic search. J. Artif. Intell. Res. **14**, 253–302 (2001)
10. Hristoskova, A., Volckaert, B., Turck, F.D.: The WTE framework: automated construction and runtime adaptation of service mashups. Autom. Softw. Eng. **20**(4), 499–542 (2013)
11. Jiang, W., Zhang, C., Huang, Z., Chen, M., Hu, S., Liu, Z.: Qsynth: a tool for QoS-aware automatic service composition. In: ICWS 2010, pp. 42–49. IEEE (2010)
12. Klusch, M., Gerber, A., Schmidt, M.: Semantic web service composition planning with OWLS-XPlan. In: AAAI Fall Symposium on Semantic Web and Agents (2005)
13. Lee, J., Kotonya, G.: Combining service-orientation with product line engineering. IEEE Softw. **27**(3), 35–41 (2010)

14. Lee, K., Kang, K.C., Lee, J.J.: Concepts and guidelines of feature modeling for product line software engineering. In: Gacek, C. (ed.) ICSR 2002. LNCS, vol. 2319, pp. 62–77. Springer, Heidelberg (2002)
15. McAllester, D., Rosenblatt, D.: Systematic nonlinear planning. In: Proceedings 9th National Conference on Artificial Intelligence (AAAI-91), Anaheim, CA. pp. 634–639 (1991)
16. Ngu, A.H.H., Carlson, M.P., Sheng, Q.Z., Paik, H.Y.: Semantic-based mashup of composite applications. IEEE Trans. Serv. Comput. **3**(1), 2–15 (2010). iD: 1
17. Nguyen, X., Kambhampati, S.: Reviving partial order planning. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence, vol. 1, pp. 459–464. Morgan Kaufmann Publishers Inc. (2001)
18. Ning, G., Zhu, Y., Lu, T., Wang, F.: BPELGEN: an algorithm of automatically converting from web services composition plan to BPEL4WS. In: ICPCA 2007, pp. 600–605. IEEE (2007)
19. Peer, J.: Web Service Composition as AI Planning - A Survey. University of St. Gallen, Switzerland (2005)
20. Rodriguez-Mier, P., Mucientes, M., Lama, M.: Automatic web service composition with a heuristic-based search algorithm. In: ICWS 2011, pp. 81–88. IEEE (2011)
21. Siddiqui, F.H., Haslum, P.: Plan quality optimisation via block decomposition. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, pp. 2387–2393. AAAI Press (2013)