The Journal of Systems & Software 146 (2018) 356–376

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Controversy Corner

Automated composition and optimization of services for variability-intensive domains

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

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

ARTICLE INFO

Keywords:
Feature models
Software product lines
Automated composition
Planning
Workflow optimization
BPEL

ABSTRACT

The growth in the number of publicly available services on the Web has encouraged developers to rely more heavily on such services to deliver products in a faster, cheaper and more reliable fashion. Many developers are now using a collection of these services in tandem to build their applications. While there has been much attention to the area of service composition, there are few works that examine the possibility of automatically generating service compositions for *variability-intensive* application domains. High variability in a domain is often captured through an organized feature space, which has the potential for developing many different application instantiations. The focus of our work is to develop an end-to-end technique that would enable the automatic generation of composite services based on a specific configuration of the feature space that would be directly executable and presented in WS-BPEL format. To this end, we adopt concepts from software product line engineering and AI planning to deliver the automated composition of online services. We will further benefit from such notions as *safeness* and *threat* from AI planning to optimize the generated service compositions by introducing parallelism where possible. Furthermore, we show how the specification of the generated service composition can be translated into executable WS-BPEL code. More specifically, the core contributions of our work are: (1) we show how AI planning techniques can be used to generate a workflow based on a feature model configuration; (2) we propose a method for optimizing a workflow generated based on AI planning techniques; and (3) we demonstrate that the optimized workflow can be directly translated into WS-BPEL code. We evaluate our work from two perspectives: *(i)* we will first formally prove that the methods that we have proposed are *sound* and *complete* from a theoretical perspective, and *(ii)* we will show through experimentation that our proposed work is usable from a practical point of view.

1. Introduction

The service-oriented architecture paradigm promotes the use of self-contained units of functionality in the form of services and/or software components to enhance interoperability, rapid development and distributed deployment, just to name a few. Many industrial entities are now providing public access to their services to developers through publicly available RESTful APIs. Directories such as ProgrammableWeb provide a systematic way of finding the available services. The importance of these open services is that they are often implemented by reliable vendors and provide functionality that are not otherwise easily implementable by smaller companies or individuals. For instance, Google Maps, Zazzle and Paypal are examples that provide access to their services through public APIs. The ease of adoption of the REST-based SOA architecture has increasingly motivated the development of applications on top of public services where multiple services are composed to cater the required functionality (Sheth et al., 2007; Rodriguez-Mier et al., 2017; Cappiello et al., 2011). Such services are often composed of a number of other services and provide added-value through new functionality. The added value of service compositions is through the emergence of newer functional capabilities that were not available prior to the integration of the already existing services.

While practitioners have traditionally employed manual or semi-automated approaches for composing services to build their required applications (Mayer et al., 2016), there is strong body of research that has investigated the automated composition of services given some input constraints and requirements specified by the users (Daniel et al., 2009; Deng et al., 2016; Liu et al., 2007). In order to facilitate the process of modeling and composing service compositions, several researchers have already identified the synergy between Software

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

https://doi.org/10.1016/j.jss.2018.07.039
Received 12 October 2017; Received in revised form 7 April 2018; Accepted 16 July 2018
Available online 19 July 2018
0164-1212/

Product Lines (SPL) and service-oriented architectures (Basile et al., 2017; Medeiros et al., 2009; Soltani et al., 2012). Within the SPL paradigm, a *feature* is often defined as an incremental prominent or distinctive user-visible functionality of a software and is therefore a good candidate to be represented and implemented through services. The integration of services and features has already been extensively investigated in the literature (Lee and Kotonya, 2010; Tizzei et al., 2017). For instance, in the model proposed by Lee et al., features are operationalized through atomic or composite services (Lee and Kotonya, 2010). 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.

The mapping of services and features allows one to develop customized service-oriented applications through the configuration of the associated software product line. However, the limitation is that while the selection of the services happens automatically in this approach as a result of the configuration, the sequence of service interactions are not determined. In other words, while the approach can determine which services should be included in the final application, it does not specify in which order they should be executed. The work that we present in this paper is positioned within the application engineering phase, described earlier, and provides mechanisms for automatically composing services based on a set of functional requirements. The objective of our work is to address the following challenges:

1. Existing work that integrate software product line techniques and service-oriented architectures face the limitation of not being able to determine the sequence of services once they have been selected through the software product line configuration process. For this reason, the concept of *business process families* (Gröner et al., 2013) has been introduced where business process templates are designed that include placeholders that show where selected services should be plugged in. A business process template limits the possibility of achieving the same objective but using a service composition with a different process structure.
2. In the business process family paradigm, given the fact that the business process template is pre-defined, there is no possibility to optimize the generated business process model depending on the circumstances. Therefore, the maximum possible variation is the selection of the appropriate services to be placed in the placeholder locations from amongst the possible service options.
3. Even in cases where business process families are not used and service composition happens directly based on the individual services using techniques such as AI planners, the final result is often a sequence of services that are chained together to perform the task. The main reason is that techniques that are used for planning or composition produce sequential solutions. Therefore, the generated solutions are not fully optimized with regards to criteria such as execution time.

Considering these limitations, the work in this paper aims to benefit from the integration of software product lines and service-oriented architecture to propose an automated approach for composing services without the need for business process templates within the context of *variability-rich domains*. Therefore, the novel aspect of our work is that it allows for the automated generation of executable service compositions for *highly variable domains*, which includes an end-to-end method encompassing (1) domain configuration based on functional requirements, (2) variability-aware service composition, (3) optimization of the service composition and (4) the generation of executable code. The concrete contributions of our work are enumerated as follows:

- We propose an AI planning based method for automated service composition for *variability-rich domains*, which operates based on software product line configurations as the main input model for specifying requirements and automatically generates a workflow that satisfies the functional requirements.
- We further propose a method for optimizing the created workflow by considering the concepts of *safeness* and *threat* from the AI planning domain in order to inject parallelism into the generated workflow and improve its execution efficiency (e.g., reduce execution time).
- We demonstrate how the generated optimized workflow can be directly translated into WS-BPEL code that can be executed without any input from the designer. This provides the added benefit for the designer in that this approach would only require her to select the desirable features from a software product line as a result of which an executable WS-BPEL code will be generated.
- We have developed tool support for all of the introduced methods in this paper. We theoretically prove the *soundness* and *correctness* of our proposed work and also report on extensive evaluation of our work under different circumstances to validate its usability and practicality.

It should be noted that this paper is an extension of our earlier work (Mahdi et al., 2016) and extends it in the following directions: (1) the current paper provides an end-to-end solution for delivering executable BPEL code based on a set of functional requirements, which includes the composition, optimization and generation processes. However, the earlier paper did not address the generation aspect of this process; (2) We provide formal proof for the *completeness, correctness* and *validity* of the generated workflows that was not present in the earlier publication; (3) We extend the discussion on the findings in the experiments and include two additional research questions; and (4) we introduce our fully functional publicly available online tool suite and also provide an extensive comparative analysis of the literature beyond what was covered earlier. It should be noted that this paper focuses on functional requirements and interested readers are encouraged to see (Mahdi et al., 2017) for the treatment of non-functional requirements.

The rest of this paper is organized as follows: The next section will cover the related literature and systematically compare our work with the state of the art. Section 3 will cover the required background information and the problem statement. In Section 4, we will describe the details of the proposed approach. This is followed by the introduction of our tool suite in Section 5. Section 6 will provide the details of the experiments and the insights gained from them. Finally, the paper is concluded in Section 8 where future work is also discussed.

2. Related work

Given the scope of this paper, we cover the related literature from four aspects: (1) The theme of our work is close to existing service composition methods that have been extensively explored in the literature. We review some of the most similar work in service composition to our work and systematically compare our approach with them in Table 1; (2) we review how software product line feature models have been used to specify service compositions and systematically compare our work with them in Table 2; (3) We review earlier work on service composition optimization, which is also addressed in our work; and finally, (4) we provide a synopsis of the work that attempt to automatically generate code for service composition and place our work within such context.

2.1. Automated service composition

Our work is positioned among considerable other research on service composition methods. Here, some of the approaches that are closely related to the approach proposed in this paper are discussed and summarized in Table 1, while the interested reader is encouraged to see recent surveys on these approaches in Sheng et al. (2014),

Table 1
Comparison of recent approaches for automated service composition.

Approach	Requirement specification	Composition specification	Composition method
Hristoskova et al. (2013)	OWL-S service description	OWLS composite service	HTN planning
Bertoli et al. (2010)	Goal State by prepositional logic	WS-BPEL	Planning with model checking
Hatzi et al. (2013)	Initial/Goal state	OWLS composite service	Conversion to PDDL
Fujii and Suda (2006)	Natural language	Sequence of operations	Inference engine
Proposed approach	Feature model	WS-BPEL	Conversion to PDDL

Table 2
Comparison of the main approaches to for service composition using feature models.

Approach	Automated	Composition specification	Composition method
Baresi et al. (2012)	✓	WS-BPEL	Configuring base model based on substitution model
Asadi et al. (2014)		BPMN	Direct mapping
Alferez et al. (2014)		WS-BPEL	Direct mapping
Proposed approach	✓	WS-BPEL	Conversion to PDDL

Lemos et al. (2016), Syu et al. (2012a). In Table 1, four comparative approaches, which are highly cited in the literature, have been compared with our work from three perspectives: (1) the models they use for requirement specification; (2) the formalization used for describing the service composition; and (3) the method used for generating the service composition.

Existing work in the literature employ four main approaches for defining and expressing *composition requirements* (second column of Table 1). The first approach is based on the natural language specification of the requirements. For example in Fujii and Suda (2006), 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 for specifying requirements, it does not provide the user with a tangible model of system functionality, which makes its adoption unreliable (Cremaschi and De Paoli, 2017). The second more popular approach is to adopt service description languages for expressing requirements. As an example, the authors in Hristoskova et al. (2013) propose to use OWL-S to define the requirements. This might be hard to use by the stakeholders as it requires a good understanding of service standards such as OWL-S. The third approach, which is commonly used in methods that use AI planners for service composition, is to express requirements based on the planning language that can then easily be translated into the planner input. For example, in Klusch et al. (2005), an XML dialect of PDDL is used to define the expected service specification. However, specification of requirements in those languages requires expert knowledge. In order to facilitate the design of service compositions in some approaches, the concrete service composition is generated from the abstract process created by the user using some GUI interface. For example in Ngu et al. (2010), users drag and drop the required components of their service composition into a canvas and create the flow by connecting these components using arcs where this process is facilitated using semantic annotations for components. Such approaches still rely on the users for designing the logic of the service interactions. In our work, different from earlier work in the literature, we propose the idea of using software product line feature models to express requirements. Feature models strike the right balance between the need for understanding functional requirements as well as avoiding low-level implementation details (Lee et al., 2002).

In terms of *composition specification* (third column of Table 1), many existing approaches use simple models such as sequence and direct graphs. However, more complex functionality are usually needed in the practical applications of service composition. Therefore, many languages and standards have been proposed in the industry such as WS-BPEL, BPML (Thiagarajan et al., 2002), and ebXML (Gibb and Damodaran, 2002). These languages and standards provide capabilities such as complex logic support, exception handling and ability to be executed by an engine. One of the popular ways to represent service composition is OWL-S (previously DAML-S). Although OWL-S main purpose is to describe semantic web services, it also allows for defining a composite service which is built on other services using basic operations. OWL-S composite services is widely used for describing service composition because of its semantic description and reasoning support which facilitates composition of the service. Examples of services composition using OWL-S are Hristoskova et al. (2013), Hatzi et al. (2013), Klusch et al. (2005), Sirin et al. (2004). For example in Hristoskova et al. (2013), existing services are defined as OWL-S atomic services and described using OWL-S precondition and effect tags. In this method, HTN planning is adopted which uses OWL-S control constructs to define the process of a composite service which serves as the goal service. Although OWL-S semantic support facilitates composition of services, it lacks capabilities which are required in many industrial contexts such as error handling (Sheng et al., 2014). We adopted WS-BPEL as a standard for defining service composition in our approach since it provides a rich language for defining business processes and it is widely used in the industry and hence increases the chances of our work being relevant and applicable for practitioners.

Now from the perspective of the *composition method* (fourth column of Table 1), planning is often considered to be the main activity in service composition since it figures out the workflow of the composite service. The methods used in service composition can be categorized into two groups: top-down and bottom-up (Syu et al., 2012b). In the top-down methods, the planner commences with the description of the desired composite service which it then decomposes to make less abstract until the process reaches a concrete workflow. Moving from a more abstract workflow to a less abstract workflow is guided by domain knowledge. For example, in Menasc et al. (2008), using an ER-model an ontology representing the domain knowledge is built. The ontology is used in order to create an abstract workflow. In the bottom-up approaches, available services are connected by matchmaking methods which attempt to build desired services using AI planning approaches such as HTN (Sirin et al., 2004), (Hristoskova et al., 2013), Model Checking (Bertoli et al., 2010; Traverso and Pistore, 2004), Theorem proving (Rao et al., 2004), and other (McDermott, 2002). Different AI planning techniques have different advantages and restrictions. For example, HTN planning can provide more efficient composition but it requires the specification of certain decompositions. PDDL has been proposed to standardize the way a planning problem is represented which facilitates changing the planner based on input requirements. Conversion of web service composition to PDDL allows for the selection of the most appropriate planner based on the context and provides the ability to use state-of-the-art planners as they are introduced in the future. In McDermott (2002), the service composition problem has also been reduced to a planning problem represented by PDDL and then solved using estimated-regression planning. However, the proposed

approach requires extending PDDL in various ways. The works presented in Sirin et al. (2004) and Hatzi et al. (2013) are similar to our work in the sense that they convert the problem into PDDL and use a planner to find the solution. But our work is different in terms of how users represent their requirements and the way service composition is represented in addition to the extra steps that we propose for workflow optimization and code generation.

2.2. Feature models in service specification

Since service-oriented engineering and software product line engineering share the goal of reusing existing assets for developing new products, there exists some methods which try to use concepts from Software Product Line Engineering in order to systematically manage variability of service oriented systems (Narwane et al., 2016; Asadi et al., 2014; Alferez et al., 2014). Some of these methods specifically use feature models as the central model of capturing variability. However, existing automated approaches which work on services provide customization of the service composition rather than composing a new service composition. Here, we review existing prominent approaches in this area and compare them to our work. Table 2 reviews these approaches.

The approach in Baresi et al. (2012) (first row of Table 2) builds service compositions from feature model configurations using an implementation of the Common Variability Language (CVL). CVL (Haugen et al., 2008) is a suggested generic approach for managing variability in domain specific languages. In this approach, each feature is annotated with a set of substitutions. Using these annotations, selected features in the feature model configuration are mapped to a series of substitutions in the base BPEL code. Base BPEL code defines the BPEL process of a generic solution to the problems in the problem domain. It also has placement locations where new process fragments can be inserted or removed for customization. A substitution is defined as placing a BPEL aspect from the CVL library into a placement location in the base code. The target business process is derived using these placements.

The work described in Asadi et al. (2014) (second row of Table 2) conceives a method for representing variability of a reference business process model using a feature model. A reference business process model Rosa et al. tries to capture generic structure of solution processes in a problem domain. This reference business process model is then customized based on specific requirements of a problem. In this method, features are mapped to activities in the reference process model using a mapping model. In the mapping model, a presence condition, which is a first order logic condition over features, is defined for each activity in the reference business model which determines if those activities should exist in the final business process. Using this mapping, a customized business process model can be built given a feature model configuration. However, the possible variation in the business process model is limited since the mapping mechanism only allows addition and removal of activities to the business process and does not allow more substantial changes such as changes to flow of the process. Additionally, this approach represents the final process using Business Process Model Notation (BPMN) which cannot be directly executed.

In Alferez et al. (2014) (third row of Table 2), an approach for building self-adaptive service composition is proposed, which is able to change its structure based on context changes. The adaptations are enabled using software product line configuration techniques. The variability of the system is modeled using a feature model and changes in the context would result in various feature model configurations. The authors propose a method for building new service compositions based on the feature model configuration using an intermediate model called the *weaving model*. Each feature is mapped to a BPEL code fragment which is inserted in the corresponding variation point in the base BPEL code of the system when that feature is selected. Although, this allows composition of the different service compositions, all these compositions should have been developed at design time. Additionally, use of the approach is limited to the variability's which can be represented in a single point composition process.

In most, if not all, of these approach which create a service composition based on a feature model or other common variability modeling techniques, the basic logic for the composition is provided by a human designer. Our work is different from these works in term of being able to automatically compose the logic satisfying the selected features without requiring a direct mapping and primarily based on the automated conversion of feature model configurations into executable WS-BPEL code.

2.3. Service composition optimization

BPEL allows sequential as well as parallel invocation of services. However, most AI planning methods come up with total-ordered sequential composition of services (Rodriguez-Mier et al., 2011). For example in Chafle et al. (2007), a planner is used to find the goal service composition which is sequential although BPEL is used to represent the composition. Some automated service composition methods generate compositions which take advantage of parallelism (Rodriguez-Mier et al., 2011; Jiang et al., 2010). For example in Rodriguez-Mier et al. (2011), service composition is modeled as a tree search problem where the goal is to find a service composition with maximum parallelism. In another example (Jiang et al., 2010), 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 parallelism is embedded in the composition process of these methods. In order to compose services with parallel execution, Peer (2005) 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 Nguyen and Kambhampati; Backstrom. 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 (Siddiqui and Haslum, 2013). However, it has not been used in the context of service composition. Our approach uses 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.

2.4. BPEL generation

Existing service composition methods which create BPEL as their final representation in the literature can be categorized into three groups. The first group of methods generate the final BPEL code by selecting appropriate partner services for an existing abstract process. These methods allow limited functionality variation in the result service and are therefore usually used in domains where the goal of composition is optimization of the result service in terms of non-functional properties. For example, Synthy (Agarwal et al., 2005; Chafle et al., 2007) proposes a physical composition approach which accepts an abstract BPEL workflow in which the workflow works with service types instead of actual service instances. This method uses a cost function to select from existing service instances of each type to create a deployable BPEL workflow which optimizes workflow quality of service. The methods in the second group generate the BPEL process directly from the requirements (Bertoli et al., 2010; Baresi et al., 2012; Pistore et al., 2004; Syu et al., 2011). As an example in Bertoli et al. (2010), a state transition system encoding BPEL processes has been proposed and an equivalent state transition system for a specific requirement is found by model checking. However, the BPEL

language is a complex language since it is an implementation level language rather than a modeling language and it is hard to consider all aspects of this language in designing a method for generating its process. Therefore, usually a more abstract model of the resulting service composition is created and then is converted to BPEL in the third group of methods (Chafle et al., 2007). This facilitates the design of the method for generating a service composition by removing redundant information. Then this abstract workflow is converted to a BPEL process (Mcdermott et al., 1998; Agarwal et al., 2005; Chafle et al., 2007). We adopt a similar approach in our proposed method.

Considering that BPEL can be readily executed and is adopted by many businesses, different methods have been proposed for converting models such as BPMN (Ouyang et al., 2008; Ouyang and Dumas, 2006), Petri-net and their variations (Aalst and Lassen, 2008; van der Aalst et al., 2005), workflow graph (Ning et al., 2007; Gtz et al., 2009) and other models (Yuan et al., 2008; Cesari et al., 2010) into BPEL formalism. Many of these models are considered graph-based while ideal BPEL code is fully structured. One of the main factors in evaluating a BPEL generation mechanism is readability of the generated code since BPEL code is verbose and complex and may needs refinement or testing (Ouyang et al., 2008). Therefore, different metrics has been proposed for evaluating the readability of a BPEL code (Aalst and Lassen, 2008; Reijers and Mendling, 2011; Reijers et al., 2011; Lassen and van der Aalst, 2009). All these methods point to structuredness as the main determinant of readability of the process and different metrics for structuredness have been proposed. One of the main factors that affect structuredness is how the workflow is organized in hierarchical components. Therefore, some work have been done which try to find single-entry single-exit (SESE) components (Johnson et al., 1994) that can then be organized in a hierarchical manner. Examples of such approaches are Reijers et al. (2011), OMG (2009), Gtz et al. (2009). However, practical workflow structures cannot always be modularized using well-structured SESE components but still organizing them as components will reduce the complexity of the structure of the workflow (Ouyang et al., 2008). In such situations, existing methods that look for SESE will stop or require user assistance. In our work, we suggest a method which allows the building of components as long as it helps the structuredness of the BPEL process.

2.5. Advantages and scope of this work

Our work in this paper distinguishes itself from the existing work in the service composition literature in the following ways:

1. End-to-end solutions that automatically generate executable business process code for service compositions based on input requirement specifications are primarily dedicated to non-variable systems. In other words, the focus of these service composition methods is to efficiently connect existing services to satisfy the requirements. Our work is among the first to provide an end-to-end solution for *highly variable* domains where the configuration of the target application as well as the generation of the final service composition need to be performed in tandem.
2. As discussed in Section 2.2, there have been earlier work that employ product line feature models to address service composition in *highly variable* domains; however, our work is among the few to perform *automated* generation of a composition based on a set of feature selections while no existing work addresses issues of composition optimization and executable code generation within the same framework for variablity-rich application as proposed in this paper.

It should be noted that this present work is limited to only considering functional requirements and not treating non-functional requirements. We have addressed the issue of considering non-functional requirements in a separate work reported in Mahdi et al. (2017).

3. Problem statement and background

The objective of our work is given a set of requirements and constraints from the users to automatically *optimally compose* services in order to satisfy the presented requirements. To achieve this objective, we rely on the integration of *software services* and software product line *features*. As mentioned earlier, researchers such as Lee and Kotonya (2010) 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 (Medeiros et al., 2009; Lian et al., 2018). 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 (Lee and Kotonya, 2010). Our focus will, therefore, be on the development of a method which automates the application engineering phase of the lifecycle and is able to build a service composition based on a set of selected features. Current automated service composition methods work on inputs such as OWL-S service descriptions (Hristoskova et al., 2013), temporal logic (Bertoli et al., 2010), or other formal languages, which are used to specify the characteristics of the desired service composition. However, we are interested in an input specification model *abstract* enough to be used by non-expert end-users to specify their requirements and an output that would be *concrete* enough to be directly executable. For this purpose we use feature models as the input specification model and generate the final outcome in WS-BPEL.

In the following, we will briefly review the two core technologies that are the foundations of the work in this paper, namely feature models and the business process execution language.

3.1. Feature models

Feature models are amongst the widely used variability modeling tools used in Software Product Line Engineering (SPLE) for representing product lines. A product line is a group of products that share meaningful similarities and are distinguished by unique characteristics often known as variabilities. A feature model provides a hierarchical tree structure that systematically organizes the similarities and variabilities of a product line through 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 known as *integrity constraints*. A *feature model configuration* is a subset of the features of a feature model which satisfies a given set of structural and integrity constraints, and represents a valid instance of the family. Prior work has shown that a feature model configuration can be used as an effective tool for representing end-users' requirements (Lee et al., 2002; Noorian et al., 2017).

Fig. 1 depicts a sample feature model for a software family that processes a purchase request and creates an invoice. The family represented through the 'Order Processing' root feature has 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. The selection of features marked with a checkbox in Fig. 1 represents a valid feature model configuration that can also be considered to be the functional requirements expressed by an end-user.

3.2. Business process execution language (BPEL)

The Web Service Business Process Execution Language (WS-BPEL), commonly interchangeably known as 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. 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. 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.

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

3.3. AI Planning

The area of AI planning focuses on developing methods for solving a class of problems known as *planning problems*. In a planning problem, a state-based world is assumed and a number of actions is defined where each action requires a number of conditions over the state space to be true for it to be executed. The execution of an action results in a change to the state of the world. It this context, a planning problem is the problem of finding actions which can change the state of the world from an initial state to a goal state. Considering that many real-world problems can be represented as a planning problem, there has been significant work in this area. To standardize the way a planning problem is described in different domains, a language called Planning Domain Definition Language (PDDL) (Mcdermott et al., 1998) has been proposed which is employed as the input language by many planning tools. As such, a planning problem can be converted into PDDL and solved using AI planners.

Planners can be categorized into two groups based on the solution they find for a planning problem (Minton et al., 1994): *total-order planners*, and *partial-order planners*. Total-order planners return a strict sequence of actions, which needs to be executed to reach the goal while partial-order planners keep the order of planning as loose as possible so the actions can be executed in different sequences or in parallel.

Total-order and partial-order planners adopt different approaches for finding a solution to a planning problem. Classical total-order planners use forward or backward chaining to find a single path of action from the initial state to the goal state. This path is represented by a strict order of action executions and returned as the planning result. Partial-order planners conversely focus on removing *threats* by introducing partial-ordering to action executions. A *threat* is present when there is a possible ordering of the partial-order plan which breaks the precondition of an action. The threat would be in the form of adding a partial ordering constraint, which prevents that specific order to be executed. Therefore, the result of partial-order planning is a set of partial order constraints. Although, partial-order planners offer more flexibility for action execution, methods for generating partial-orders are more complex; therefore, they do not scale well, which makes them unsuitable for many problem domains (Nguyen and Kambhampati, 2001b). Therefore, the focus in both research and industry has been on total-order planners. As such, we also use a total-order planner for generating a service composition and then use concepts from partial-order planning to optimize the returned total-order plan.

4. Proposed approach

In this paper, we propose an automated service composition and workflow optimization method which receives a set of functional requirements in the form of a feature model configuration and automatically builds fully executable WS-BPEL code. Fig. 3 shows the overview of our proposed service composition and optimization process. Our process adopts SPLE's two-phase lifecycle (Pohl et al., 2005).

In the first phase, a feature model and the other related artifacts for the variability intensive family of products are built based on the requirements of that family. Here, the requirements of the family are captured and modeled in the form of a set of models that are collectively called *domain models* (Pohl et al., 2005). In our approach, the domain model consists of five sub-models, namely the feature model, the feature model annotation, the context model, the service model, and the service model annotation. The feature model captures possible variability in the service composition in terms of features. The service model represents the services which are available for realizing different variants of the service composition family. The context model provides the means for representing the state of the service composition environment. The context model serves as a bridge, which links features to services. Feature model annotations use the context model to describe how each feature affects the preconditions of a service composition and how it impacts the operating context. Similarly, service model annotations describe the preconditions and effects of individual services. On the basis of these models, the features in the feature model are linked to services realizing them using a context model, feature model annotations and service model annotations.

In the second phase, the feature model from within the domain model is used to select the desired features that need to be included in the final service composition through the *Requirement Specification* activity. Once the features to be included are selected and a feature model configuration is developed, the *Automated Composition Construction* activity works with the feature model configuration as the input and automatically generates the appropriate BPEL code for the service composition. In this paper, we realistically assume that domain engineering phase has already been performed and the features of the feature model configuration have been connected to relevant services based on existing techniques from the literature (Lee and Kotonya, 2010); therefore, only the realization of the '*automated composition construction*' activity is the focus of this paper.

Our proposed automated service composition and optimization process consists of three steps. In the first step, the problem of finding the service composition which satisfies the selected features of the input feature model configuration is reduced to an AI planning problem. The result of this step is an intermediate model, which we refer to as the *workflow*, which abstractly represents the final goal service composition. The second step is optimizing the generated workflow in order to effectively use parallelism where possible. Although, this step is optional, it significantly improves the efficiency of the workflow from different perspectives. In the third step, the optimized workflow is converted into a well-structured readable BPEL code which can be directly executed. Since, the three steps work with domain models, we first formally define the *domain models* in the rest of this section and then introduce the details of how each of these three different steps are realized.

4.1. Domain models

As mentioned earlier, we define the domain model to consist of five sub-models, namely the feature model, the service model, the context model, the service model annotation, and the feature model annotation. In addition to the domain models, our method works with two other models, namely the feature model configuration and the workflow model. A feature model configuration is a subset of features in the feature model, which respect constraints enforced by the feature model and is created by the end-user to specify a desired functionality for the system. The workflow model is an internal model, which is generated through an optimization process and is used to generate a BPEL code. In essence, the workflow model is a graph representing the precedence relations between service executions. This method enables us to simplify the solution by separating the activities that are required for the creation of a service composition from those activities that are required for generating BPEL code from that service composition. In our work, we formally define a feature model and feature model configuration as follows:

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

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

For the feature model shown in Fig. 1, the set F would include all the features in the model and the function $\mathcal{F}_\mathcal{M}$ will include relations such as (*ShippingScheduling, InvoiceCreation*) which represents the constraint that the *ShippingScheduling* feature should be in the configuration when the *InvoiceCreation* feature is in that configuration. Other structural and integrity constraints in the feature model can be defined similarly. Using the above definition, a feature model configuration can defined as follows:

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

Based on this definition, a valid feature model configuration is a subset of features that satisfy the structural and integrity constraints expressed in the feature model. Now, given such a feature model configuration, our objective is to develop a workflow that would realize the feature model configuration using services. In order to represent how requirements represented by a feature model configuration is fulfilled using services, the specific orchestration of services that satisfy those requirements is captured in a workflow model. A workflow specifies the sequence of interactions between the services in a service composition. We first formally define the service model, which is the basis for our workflow model, as follows:

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

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

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

In this definition, a service is defined by the entities that it takes as input, the entities that it returns after invocation, and the entities that it returns in its callback if it results in a callback. Each of these entities is strictly typed. For example in Fig. 2, the service *requestProductionScheduling* can be defined with $I = \{inputPurchaseOrder\}$, $O = \{\}$, and $O_c = \{outputProductionSchedule\}$ where the type of entities *inputPurchaseOrder* and *outputProductionSchedule* is *PurchaseOrder* and *ProductionSchedule*, respectively. Using this definition, we define the workflow model as:

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

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

Fig. 4 shows a graphical representation of a workflow model represented as a directed graph in which nodes are operation nodes and edges represent the dependencies between operation nodes. For example, node A shows an operation node which involves the invocation of *requestInvoiceCreation* service and the edge from node A to C shows that the invocation of *requestShipping* should be performed after the invocation of the *requestInvoiceCreation* service.

In order to be able to automatically make a transition from a feature model configuration to a workflow model, we define the *context model*, which represents the environment in which the service composition will

operate in. Relations between the feature model, the service model and the context model are represented with annotations on these models. These annotations are used for creating a workflow from the feature model configuration. We formally define a context model as follows:

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
  - $\mathcal{F}$: $\Phi \mapsto \Theta \times \cdots \times \Theta$ is a function which specifies 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
  - $\mathcal{T}$: $\mathcal{E} \mapsto \Theta$ is a function which defines the type of each entity
- S is a *context state* which is a set $S \subset \Phi \times E \times \cdots \times E$ such that for each fact $f = (\phi, e_1, \cdots, e_k) \in S \Rightarrow (\phi, \mathcal{T}(e_1), \cdots, \mathcal{T}(e_k)) \in \mathcal{F}$ 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. 5. In this example, *c* and *po* are two context entities, which are of customer and purchase order types, respectively. Furthermore, the fact *ordered*(*c, po*) expresses that customer *c* has ordered the purchase order *po*. This fact is represented using the fact type *ordered* which relates an entity of type customer to an entity of type purchase order. We will explain in the following how the context model information will be used to annotate features and services.

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

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

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

Fig. 5 shows the annotations for our order processing feature model. As seen in the figure, for each feature, E_f, $\mathcal{P}_f$, $\mathcal{E}_f$ are defined as needed. For instance, the figure shows that for the 'Invoice Creation' feature to be included in the goal service composition, a context entity *i* of type *Invoice* needs to be present in the context model. Furthermore, when the service composition consisting of the 'Invoice Creation' feature is executed, the fact *hasInvoice*(*po, i*) will become true as an effect, which means purchase order entity *po* will have an invoice entity *i*.

In addition to feature model annotations, we also annotate services in a similar vein. The annotation of services with pre-conditions and post-conditions (effects) has been already widely used in the literature (Hristoskova et al., 2013) and we adopt a similar strategy.

Definition 7. *(Service model annotation)* A service annotation for service *s* is a tuple $\mathcal{A}_s = (\mathcal{P}_I, \mathcal{Q}_I, \mathcal{R}_I, \mathcal{P}_C, \mathcal{Q}_C, \mathcal{R}_C)$ where assuming $\mathcal{IO} = \mathcal{I} \cup \mathcal{O} \cup \mathcal{O}_c$ we have

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

For example in the service *requestProductionScheduling* in Fig. 2, assuming the input *inputPurchaseOrder* is of type *PurchaseOrder* and the callback output *outputProductionSchedule* is of type *ProductionSchedule* in the context model, one could define the annotations for this service as $\mathcal{P}_I = \{ordered(?customerInfo, inputPurchaseOrder)\}$, $\mathcal{Q}_C$={*hasProductionSchedule* (*inputPurchaseOrder, outputProductionSchedule*)}, and the other annotation sets would be empty. This annotation means that after the invocation of this service, the value of the output would be the shipping information for the input customer and after receiving the callback the value of the callback output would be the shipping schedule for the input customer.

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

4.2. Workflow generation

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

We propose to represent the above problem as a standard planning problem and represent it using Planning Domain Definition Language (PDDL) (Mcdermott et al., 1998) and then find a solution through AI planning. A planning problem is defined by the initial context state as the starting point of the planner and the expected context state as the goal of the planner. A planning problem can be defined in different domains. A domain defines a set of possible actions that can be applied in order to change the state. A planner can find a sequence of actions, which move from the initial state to a goal state for a given domain if such sequence exists. Representing the problem of generating a desired workflow as a planning problem allows us to take advantage of existing highly optimized planners for finding a possible workflow solution.

In order to represent the desired workflow generation problem as a planning problem, we first formally define the planning problem and the planning domain and then discuss how generating a desired workflow can be accomplished as a result. 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 (Mcdermott et al., 1998). A planning problem in STRIPS (Fikes and Nilsson, 1972) can be defined as follows:

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

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

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

Definition 9. *(Planning problem solution)* Sequence $s = <a_1, \cdots, a_k>$ is a solution to the planning problem $p = (S_{initial}, S_{goal})$ in a domain A if

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

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

Definition 10. *(Planning problem based on feature model configuration)* For a feature model configuration C, the initial and goal states of a planning problem $p = (S_{initial}, S_{goal})$ that are defined as follows:

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

Initial and goal states of the planner are built by aggregating the annotations of the selected features from the feature model configuration. Fig. 6-(a) shows the PDDL representation of a planning problem of the feature model configuration shown in Fig. 1. Lines 13–17 show the initial condition for the planning problem, created by unioning over all $\mathcal{P}_f$ sets of the selected features. For example, the fact on Line 15 has been created as a consequence of the annotation on the *Domestic* feature, which enforces that the destination of the purchase order is Canada. Lines 19–26 show the goal condition for the planning problem, which has been created in a similar vein using $\mathcal{E}_f$ sets of the selected features.

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

Definition 11. *(Planning domain based on services)* In a domain where a set of services S is available, and for each $s \in S$ its corresponding annotation is $\mathcal{A}_S(s)$, the problem domain would be the set of actions A. For each service $s \in S$, A includes an action $a_{invoke}(s)$ and it includes action $a_{callback}(s)$ if the service call results in a callback. These actions can be defined as:

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

The sets F_{pre}, F_{add}, and F_{del} for an action are built based on the set of $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{R}$ of the corresponding operation in the related service. Fig. 6-(b) shows parts of the PDDL representation of the planning domain based on the services used for realizing the service composition satisfying the requirements of the feature model configuration shown in Fig. 1. Lines 13–25 show the action representation of invoking the *requestProductionSchedule* service. Lines 15–19 shows the parameters for the action which is made of inputs and callback output since invocation of the service has no output. The precondition for service invocation corresponding action (i.e., the set F_{pre}) is represented in Lines 20–22. The effects of the services invocation corresponding actions (i.e., the sets F_{add}, F_{del}) are represented in Lines 24–26.

Now that the planning goal and planning problem domain are concretely defined, a planner can be used in order to find a solution for the planning problem. The solution will be a sequence of actions with input variable assignments, which would take us from the initial context state to the goal context state. Fig. 7 shows the sequence of actions which has been found by the FF planner (Hoffmann and Nebel, 2001) as a solution for the example problem where each action and its input assignment is represented in a separate line. In the following, we formally define how the workflow can be built based on the planner solution and prove that the generated workflow is a valid workflow and satisfies the requirements specified in the feature model configuration.

Definition 12. *(Workflow based on planning problem solution)* Based on a solution to a planning problem denoted as $s = <a_1, \cdots, a_k>$ for an input feature model configuration C, a workflow $w = (E, N, \mathcal{E})$ can be built where:

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

Lemma 1. *(Soundness of the Workflow) Let C be a feature model configuration. A workflow w is generated based on Definition 12. w is a valid workflow in that the following conditions hold for it:*

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

Proof Cond 1.1. The proof goes by contradiction. Assume that there exists some nodes in w whose preconditions do not hold before their execution. Therefore there must be an ith node in the workflow execution sequence which is the first node in the sequence whose precondition is not satisfied. We show that such node cannot exist. If precondition for operation node n_i does not hold, it means there is at least one precondition p which does not hold. This operation node can

be mapped to its corresponding planning action a_i. According to Definition 11, each precondition for n_i can be mapped to a precondition for a_i. Therefore, if p does not hold after executing operation nodes 1 to $i-1$ then there exists a p' which is a precondition for a_i that does not hold after executing actions 1 to $i-1$. The planner has chosen a_i for the next action, meaning that the planner has chosen an action whose preconditions are not satisfied. This is not possible. □

Proof Cond 1.2. The proof goes by contradiction. Assuming there exists a receive operation node n_i whose corresponding invoke has not been called. This operation can be mapped to its corresponding action a_i. According to Definition 11, one of the requirements of this action is a condition which only becomes true when the corresponding invoke action has been executed. Since the corresponding invoke action has not been executed, the planner has chosen an action whose preconditions are not satisfied. This is not possible. □

Lemma 2. *(Completeness of the workflow) Let C be a feature model configuration and w be a workflow generated based on Definition 12, the execution of w requires only the conditions specified by the feature model configuration and results in a state which satisfies all effects specified in the feature model configuration.*

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

Theorem 1. *(Correctness of the workflow) Let C be a feature model configuration and w be a workflow generated based on Definition 12, w is a valid workflow and it consists of all the features specified in the feature model configuration.*

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

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

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

4.3. Workflow optimization

Although the generated workflow can be used to generate BPEL code, given that the AI planners produce strictly sequential plans, the generated workflow could benefit from potentially more efficient and valid plans which use parallel execution of operations when possible. Using parallelism in a service workflow can significantly affect the efficiency of the composed service (Rodriguez-Mier et al., 2011). Therefore, once a plan is generated by the AI planner, we take an additional step to optimize the workflow.

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

Algorithm 1. Pseudo-code for workflow optimization.

Let us first provide an overview of how the optimization is performed before providing the formal details. The idea of our proposed approach for workflow optimization is to remove as many edges from the workflow as possible in order to relax the execution sequence without violating the semantics of the execution. In order to do this, we assume each edge in our workflow is a candidate for being removed from the workflow and evaluate whether it can be removed or not. The condition for removal is that the required preconditions for any forthcoming service execution in the workflow is not violated; in other words, we can only remove an edge if its removal does not lead to the premature execution of services whose preconditions are not yet prepared. We refer to this as the *safeness* condition. Therefore, simply stated, the process of workflow optimization is the iterative removal of edges from the workflow that respect the safeness condition.

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

The definition of SELECT and TERMINATIONCONDITION in Algorithm 1 depends on the optimization method which has been selected. The definition for SAFE which is responsible for examining the 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 McAllester and Rosenblatt. In this algorithm, the main loop iterates over all operation nodes of the workflow (Lines 2–16) and its immediate inner loop iterates over all of the facts that are required to be true as the precondition of the node (Lines 3–15). For each precondition fact p of each node n, this algorithm iterates over all the nodes in the workflow (Lines 5–11) in order to find an operation node n' which

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

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

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

makes that fact true; meaning that $p \in Q(n')$ and is executed before node n ensured by calling after(w, n', n) (Line 6). The function after(w, n', n) (Line 6) always looks for a path from n' to n in the workflow. If such path exists, it means that n will be executed after n'. 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 2. This algorithm works by iterating over all nodes in the workflow and analyzes if it can pose a threat to the causal link (Lines 2–6). A node can be considered a threat to a causal link if it does not execute before the start node or after the end node of the causal link and makes the related fact to that causal link false (Line 3).

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

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

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

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

Although we prove that preconditions of all workflow operation nodes will be satisfied, we still need to make sure that the whole workflow precondition and effects stays the same during optimization. This can be done by a small modification in the input workflow: a new start operation node with workflow preconditions as its effects is added to the beginning of the workflow and an operation node with workflow expected effects as its preconditions is added to the end of the workflow. Considering that the optimization will not affect precondition satisfaction, it can be easily proven that if the start and end node is removed from the workflow after optimization, the resulting workflow will satisfy its expected preconditions and effects.

4.4. Generating BPEL code

Two main approaches have been used in the literature for representing service compositions: approaches based on block structure Microsoft; Thatte and graph-based approaches OMG; Leymann. In the block structure approaches, the process is represented with hierarchical blocks which decompose the process structure to smaller blocks that can themselves be built of smaller blocks. Example of such models are Windows Workflow Foundation (WF) Microsoft, Microsoft XLANG (Thatte, 2001), and conventional programming languages. In the graph-based models, the structure of the dependency between operations are represented using arcs between the operations similar to how we represent workflow in our work. Example of such models are BPMN (OMG, 2009) and IBM's WSFL Leymann. BPEL has been built based on IBM's WSFL Leymann which is graph-based language and Microsoft XLANG (Thatte, 2001), which is a block structured language. BPEL has inherited from both these languages and therefore has the capability to represent a service composition in both ways. The generated workflow in our work can easily be represented in BPEL using a graph-based structure using <*link*> tags. For example, Fig. 15-A shows the graphical representation of a solely graph-based BPEL code for the workflow in Fig. 4. However, the graph-based representation of a workflow is usually hard to comprehend and is often not chosen as a way to represent a process (Aalst and Lassen, 2008; Reijers and Mendling, 2011).

In BPEL, the structured way to represent dependencies between operations other than control operations are <*flow*> and <*sequence*> structures. Therefore, some approaches have proposed methods for building fully block structured BPEL using only those structures from graph-based languages such as BPMN (OMG, 2009). However, the

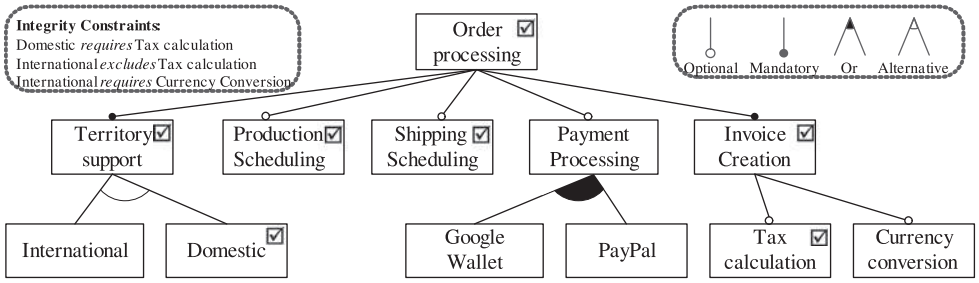

Fig. 1. A sample feature model for an order processing family.

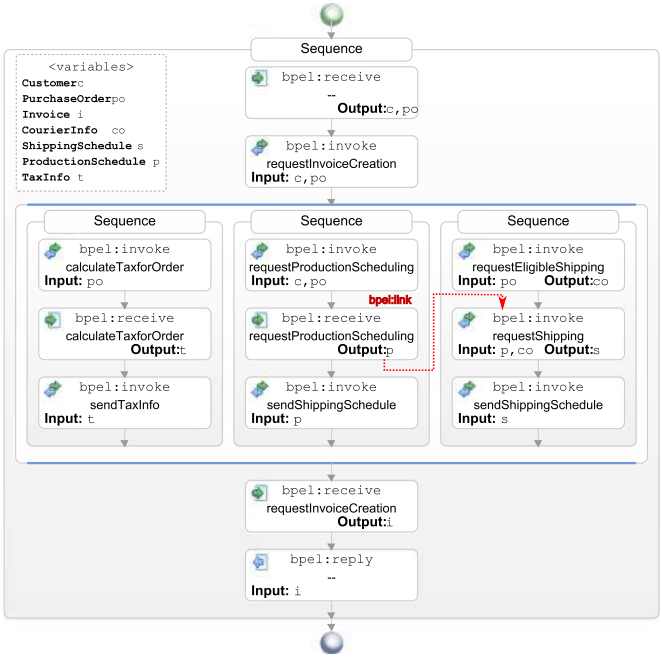

Fig. 2. Graphical representation of a possible WS-BPEL process for order processing.

dependency in a graph-based language cannot be fully captured through only hierarchical use of these two structs. That is the reason why BPEL provides the link tag which allows the representation of graph-like dependencies between nested children of the flow activity. Considering that, it has been shown that using more graph-based structures (i.e., link tags) in the BPEL process reduces its readability (Reijers and Mendling, 2011). For this reason, we propose a method for converting our generated workflow into a BPEL process which minimizes the number of graph-based structures while preserving the actual dependency represented in the workflow graph.

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

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

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

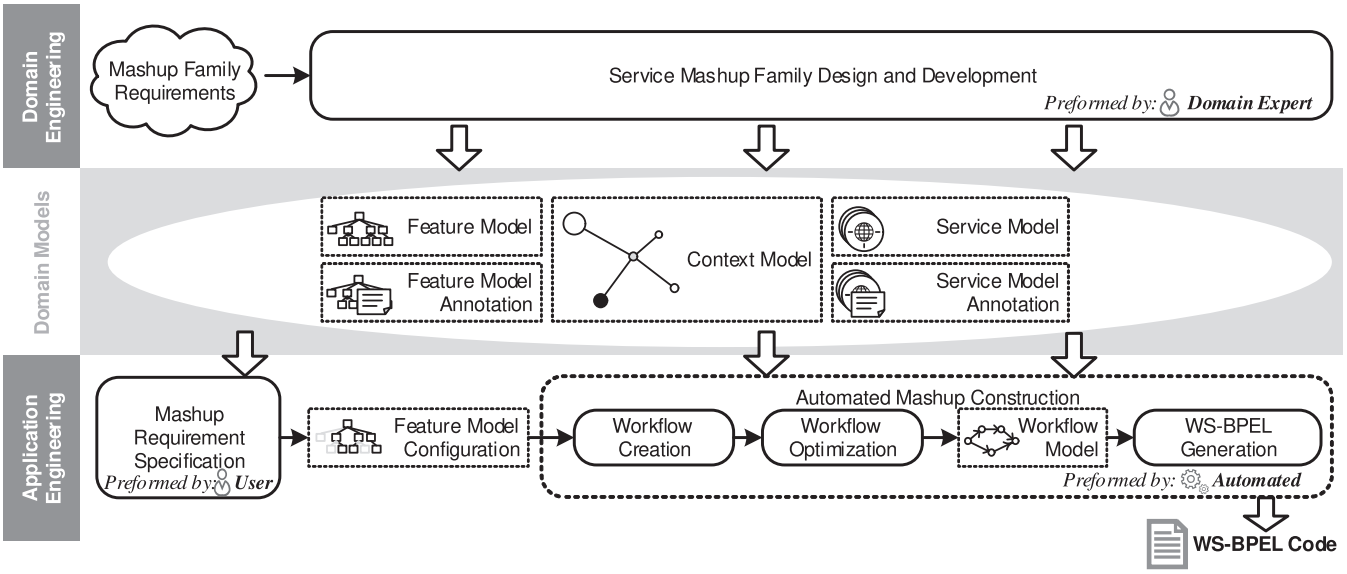

Fig. 3. The proposed two phase process for service composition.

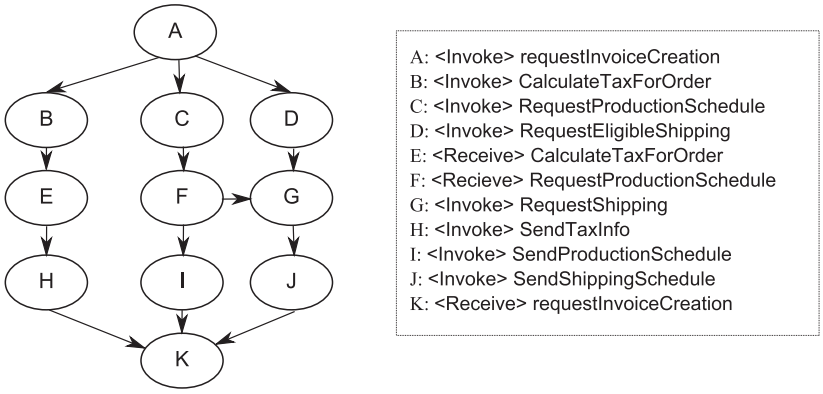

Fig. 4. A workflow solution satisfying requirements of feature model configuration in Fig. 1.

– A sequence $< a_1, \cdots, a_n >$ which is an activity made of a sequence of sub-activities that should be executed in the order specified in the sequence;
– An atomic activity which can be any of the BPEL atomic operations such as invocation of a service.

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

We propose a process that takes a BPEL process as input and attempts to reduce the number of graph-based structures in the process and replaces them with block-based structures with the goal of reducing the number of graph-based structures. This is done through a *fold* operation in which the set of activities whose dependencies are represented using link tags are replaced with a block structured activity. This is similar to the work presented in Aalst and Lassen (2008). However in Aalst and Lassen (2008), BPEL generation requires human intervention during its process since that method is only able to fold completely well-structured set of activities. A set of activities is called well-structured when all of its members dependencies can be represented using a block structure component and therefore all graph-based structures (i.e., link structures) in which they are involved is removed when it is folded into that component. Therefore, it requires human intervention when such set of activities cannot be found. Our proposed method is fully automated and continues to look for the set of activities which are not well-structured but can be folded and their folding reduces the graph-based structure.

Algorithm 3 shows the function for the proposed folding algorithm. The algorithm finds the parts of the process which can be folded into a block-structured activity by calling the *FindFoldingCandidates* function (Line 2). This function gets a BPEL process as input and returns a set of possible folding candidates which are block-structured activities. The main loop of the function (Lines 3–7) repeats until there is no more folding candidates for creating a block structure. In this loop, the most suitable candidate for folding is selected by calling *SelectBestCandidate* (Line 4). After the best folding candidate has been selected, the activities in BPEL process is folded by calling the *Fold* method (Line 5). We are using the same folding mechanism as the one used in Aalst and Lassen (2008) therefore we do not provide formal definition for the fold method. After that, folding candidates in the result process are found and the loop repeats.

The *SelectBestCandidate* function has been defined in Algorithm 3. This algorithm uses two functions LINK (Line 8) and UNFOLDABLELINK (Line 9). The LINK function returns the total number of links which have been used in the flow activity as well as the nested flows in it. The UNFOLDABLELINK function returns those links in the flow structure which cannot be folded in the rest of the optimization. These are those links which begin or end in an activity which is not the immediate child of flow and nested in one of the activities of the flow. This algorithm works by iterating over all folding candidates (Line 12–18). For each folding candidate, the flow after folding that candidate is found (Line 11). In this new flow, the number of links and unfoldable links in that flow structure is found (Line 14). If the number of unfoldable links in this candidate is less the than best candidate which has been found till

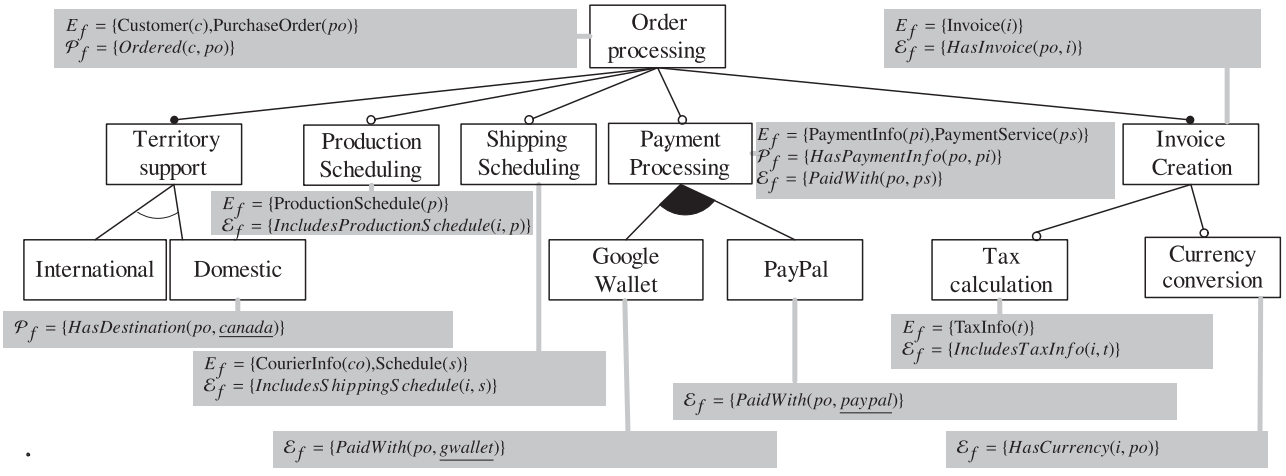

Fig. 5. An annoated feature model for the order processing family.

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

(a) Planning Problem

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

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

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

(b) Planning Domain

Fig. 6. Example of planning problem and planning domain in PDDL.

```
(IREQUESTINVOICECREATIONSERVICE VCUSTOMER VPURCHASEORDER)
(ICALCULATETAXFORORDERSERVICE VPURCHASEORDER VCUSTOMER)
(IREQUESTPRODUCTIONSCHEDULESERVICE VPURCHASEORDER VCUSTOMER)
(IREQUESTELIGIBLESHIPPINGSERVICE VPURCHASEORDER VSHIPPINGCOURIER VCUSTOMER)
(RCALCULATETAXFORORDERSERVICE VPURCHASEORDER VTAXINFO)
(RREQUESTPRODUCTIONSCHEDULESERVICE VPURCHASEORDER VPRODUCTIONSCHEDULE)
(IREQUESTSHIPPINGSERVICE VPURCHASEORDER VPRODUCTIONSCHEDULE VSHIPPINGCOURIER
      VSHIPPINGSCHEDULE VCUSTOMER)
(SENDSHIPPINGSCHEDULESERVICE VSHIPPINGSCHEDULE VPURCHASEORDER)
(SENDTAXINFOSERVICE VTAXINFO VPURCHASEORDER)
(SENDPRODUCTIONSCHEDULESERVICE VPRODUCTIONSCHEDULE VPURCHASEORDER)
(RREQUESTINVOICECREATIONSERVICE VCUSTOMER VPURCHASEORDER VINVOICE VTAXINFO
      VSHIPPINGSCHEDULE VPRODUCTIONSCHEDULE)
```

Fig. 7. Solution returned by a planner for our example problem.

now or it has equal number of unfoldable links but less number of links, the current candidate is selected as the best candidate (Lines 15–17). This algorithm uses a greedy selection method whereby the candidate which results in the least immediate unfoldable links and most reduction of the links is selected. The selection method has shown to be effective in our evaluations.

Algorithm 4defines the function for finding folding candidates. The algorithm works by iterating over all activities in the main flow and finding flow and sequence structures related to each activity (Line 2–5). Specifically, for each activity a, it finds sequence structures which start from a and the flows which a can be a part of. The algorithm then adds all found structures to the set C which stores the potential candidates (Line 4). Sequences starting from an activity are found by calling the *FindSequence* function. In this function, for each link element starting from an input activity, we find the sequences from the end activity of the link and create a sequence by adding starting activity to all those sequences (Line 9–11). Function *FindFlow* finds a flow activity that an activity can be part of if such flow activity exists. This function starts with result set F with the input activity as its only member (Line 14). It then iterates over all the activities in the flow to find activities which can form a flow structure with this activity (Line 15–19). An activity can form a flow with the input activity when the set of all activities with a link to and from it would be the same. The function *Preceding* and *Following* return the set of preceding and following activities for an activity, respectively (Line 16). If that is the case, then that activity will be added to the result set F. After examining all activities, if the cardinality of the result set is less than two, it means that the input activity cannot form a flow structure with any other activity and therefore the function returns an empty set. Otherwise, it will create a flow structure with the result set and with all the links in the main flow whose start and end activities are in the sub-activities of the newly created flow children.

5. Tooling support

The contributions of our work in the paper have been implemented in a web-based tool suite[1]. Our tool suite relies on the FF planner (Hoffmann and Nebel, 2001), which is a highly optimized PDDL-compliant planner for finding a solution to a generated planning problem.

[1] The code for the tool suite is available at https://github.com/matba/magus

function OptimizeBPEL(flow $f = (A, \mathcal{L})$)
$C \leftarrow$ FindFoldingCandidates(f)
while $C \neq \emptyset$ **do**
  $c \leftarrow$ SelectBestCandidate(C)
  $f =$ Fold(f, c)
  $C \leftarrow$ FindFoldingCandidates(f)
end while

Link $(f) = \mathcal{L}(f) \cup \{$Link$(f')$ s.t. $f' \in A(f)\}$
UnfoldableLink $(f) = \{l = (a, a') \in \mathcal{L}(f)$ s.t. $a \notin A(f)$ or $a' \notin A(f)\} \cup \{$Link(f') s.t. $f' \in A(f)\}$
function SelectBestCandidate(flow $f = (A, \mathcal{L})$, activityset C)
$bc = \emptyset$
for all $a \in C$ **do**
  $f' =$ Fold(f, c)
  $sizeL = |$ Link $(f')|$, $sizeUL = |$ UnfoldableLink $(f')|$
  if $bc = \emptyset$ or $sizeUL < sizeUL_{bc}$ or ($sizeUL_{bc} = sizeUL$ and $sizeL < sizeL_{bc}$) **then**
    $bc = c$, $sizeL_{bc} = sizeL$, $sizeUL_{bc} = sizeUL$
  end if
end for
return bc

Algorithm 3. Algorithm for BPEL process optimization.

function FindFoldingCandidates(flow f)
$C \leftarrow \emptyset$
for all $a \in A(f)$ **do**
  $C \leftarrow C \cup$ FindSequences(f, a) $\cup$ FindFlow(f, a)
end for
return C

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

function FindFlow(flow f, activity a)
$F \leftarrow \{a\}$
for all $a' \in A(f)$ **do**
  if Preceding(f, a) =Preceding(f, a') and Following(f, a) =Following(f, a') **then**
    $F \leftarrow F \cup \{a'\}$
  end if
end for
if $|F| < 2$ **then**
  return $\emptyset$
end if
$\mathcal{L}' = \{(a, a')$ *s.t.* $(a, a') \in \mathcal{L}(f)$ *and* $a, a' \in \{$SubActivity(f') s.t. $f' \in F\}$
return $\{(F, \mathcal{L}')\}$

Algorithm 4. Algorithm for Finding Folding Candidates.

For optimization and BPEL generation, the Java implementation of the proposed method is used in our tool suite. The tool suite support for the proposed method consists of two main design tools. The first tool is the *Domain Design Tool*. It is an Integrated Development Environment in which designers can create new service families. First of all, it helps the designer build the feature model and import services for the service family. Second, it enables designers to define the context model and annotate feature models and services using that context model. The tool serializes the models and allows for their storage and retrieval. The second tool is the *Configuration Tool*. It is a web application which provides feature model configuration utility by which the end-user can select the desirable features of the service composition. The service composition is generated by this tool based on the input feature model configuration and according the techniques presented in this paper. The Configuration Tool can be accessed online[2] and provides the following functionalities:

- It enables users to load, view, and edit a service family;
- It allows configuration of a service family feature model and validation of the configuration;
- It allows for the generation of the workflow graph and BPEL process

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

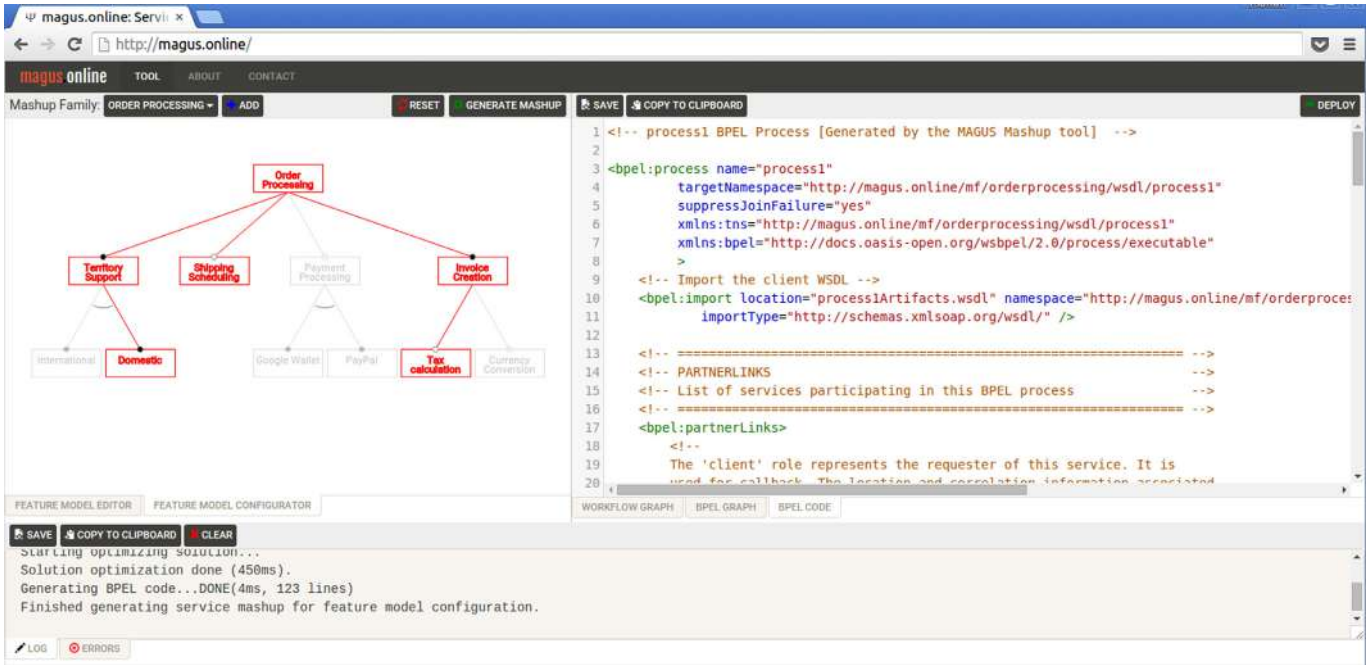

Fig. 8. The screenshot of the Configuration Tool.

for a service composition with specific features;
- It visualizes the workflow graph and BPEL process and enables the editing of the workflow graph and deploying of the BPEL process.

Fig. 8 shows the screenshot of the main page of this tool. The first part is the service family loading toolbar. The user starts by loading a service family. Users can load a service family to the tool using the URI to family XML configuration file. A service family file contains information about the annotated feature model and the service repository. After loading a service family, the feature model for that family is visually presented in the feature model panel. This panel has editing and configuration tabs. A user can select and deselect desirable features by clicking on them in the configuration tab. For cases when the existing feature model does not cover all the requirements of the user, the feature model structure and annotations can be changed in the feature model edit tab. In the editing tab, a set of possible actions for each feature such as adding a child or editing annotations is displayed by clicking on each feature.

Once the feature model configuration is completed, the users can request service composition generation where the set of features are processed. In the first step, the structural and integrity constraints are examined. In the case of a conflict, a message showing the source of conflict is displayed in the logging panel. Otherwise, three different views of the generated composition is displayed in different tabs in the process panel. In the workflow tab, the workflow graph which consists of operation nodes and dependency edges are visually represented. The user can modify this graph if necessary by adding or removing dependency edges between operation nodes. In the BPEL schematic view tab, the visual representation of the organization of the activities is displayed. This visual representation can be used for understanding and verifying the logic of the process since BPEL code is hard to read. In the BPEL code tab, the readily executable BPEL code is displayed which can be saved or deployed to a BPEL execution engine.

6. Experiments

The proposed work can be divided into three main parts, which are: workflow generation, workflow optimization, and BPEL code generation. In the following, we go through these three parts and describe the experiments which have been performed in order to evaluate them. These experiments were performed on a machine with Intel Core i5 2.5 GHZ CPU, 6 GB of RAM, Ubuntu 14.04 and Java Runtime Environment v1.8.

6.1. Workflow generation

We have already formally shown the soundness and correctness of our proposed techniques; therefore, 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 the 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 the 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 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 descriptions generator which creates service descriptions with inputs, outputs, preconditions, and effects randomly picked from our context model. This OWL-S service description generator is highly customizable with different service model characteristics (e.g. number of inputs, outputs, precondition, and effects). Three service repository sets have been created where services in the repositories of different sets have different numbers of 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 1000 to 10,000. Totally, 30 different service repositories have been created.

Feature model and its annotations: We used the SPLOT feature model

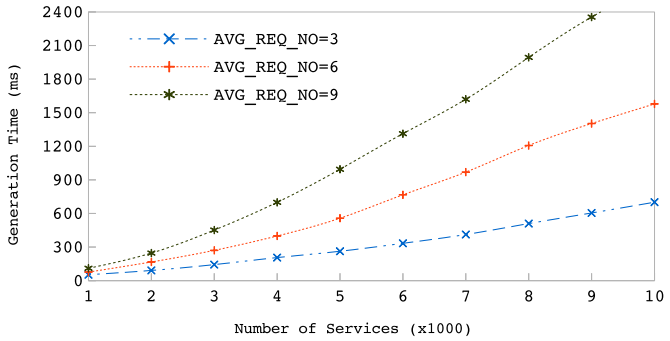

Fig. 9. Workflow generation time in terms of service repository size.

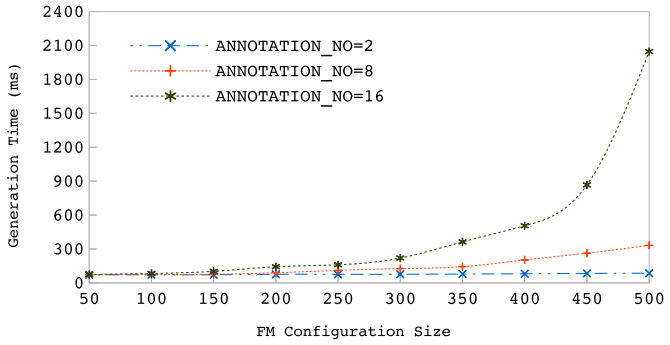

Fig. 10. Workflow generation time in terms of feature model configuration size.

generator (Mendonca et al., 2009) to generate a feature model with 1000 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 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. Fig. 9 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 seconds for 9000 services). In the second experiment, the service repository with 1000 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 randomly selecting features using a tool based on FaMa suite (Benavides et al., 2007) which receives a feature model and the desired number of features in the configuration and returns a random valid feature model configuration with that size. For each configuration size, 20 different configurations are generated. For each number of annotations, the average time required for generating the workflow is calculated for different configurations. Fig. 10 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 is increased to 16, the generation time becomes exponential and shows rapid increase. It is important to note that (i) even with the increase, the time is manageable for practical purposes, i.e., 2 seconds for 1000 services and 500 requirements. (ii)

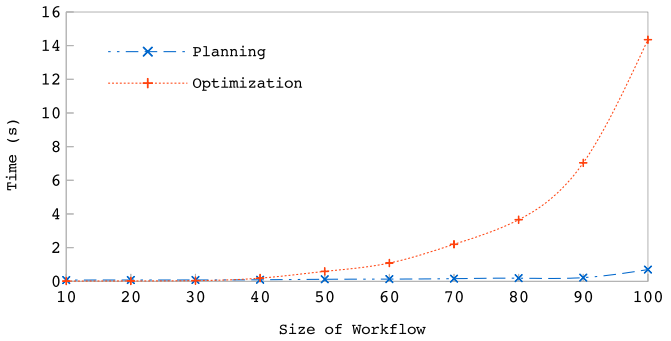

Fig. 11. Workflow optimization time in terms of workflow size.

Literature suggests that the number of annotations is typically in the range of 5–6 annotations per feature (Asadi et al., 2014), in which case, the performance of the generation algorithm is linear.

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

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

For the sake of the experiments, the models from the previous experiment were reused. The service repository with 1000 services and an average number of preconditions and effects of 6 were employed. The services were annotated with random time-to-completion with a normal distribution of $\mathcal{N}(200ms, 50)$.

For the first experiment, 20 different configurations in each workflow size category was randomly selected and the average time for workflow generation and optimization is calculated. Fig. 11 shows how workflow generation and optimization time increases as the size of workflow grows. This shows that the workflow optimization method is considerably slower than the planning method. However, given the fact that the optimization method is only a one time task, its benefits in terms of reducing the time-to-completion is noticeable.

In the second experiment, the objective was to measure whether the optimization method is able to generate workflows that have a lower time-to-completion compared to the original non-optimized workflows.

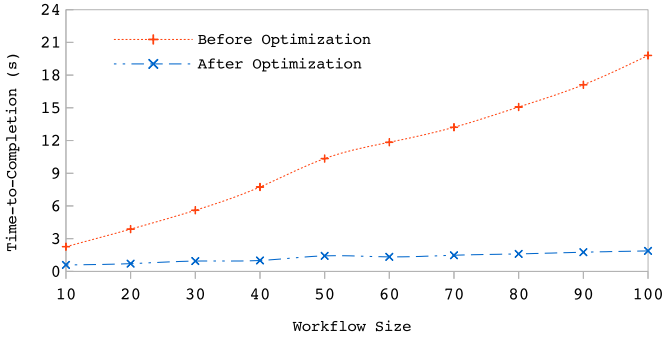

Fig. 12. Time to completion in terms of workflow size.

For this purpose, the time-to-completion of the generated workflow were calculated both before and after the optimization. Fig. 12 shows the result of the optimization. As seen in the figure, the time-to-completion of a non-optimized 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.

6.3. BPEL process generation

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

- *Experiment 3.1 (Process structuredness in terms of the number of block-structure activities):* How well does the proposed method take advantage of block-structured activities in the generated BPEL code compared to other methods of BPEL generation?
- *Experiment 3.2 (BPEL process generation scalability in terms of process size):* How does the BPEL process generation time increase with the increase in the size of input workflow (in terms of growth in the number of workflow nodes)?

In order to run these experiments, the workflow models from previous experiments were re-used. In order to compare our proposed method with other existing methods, the method proposed in Ning et al. (2007) and a method based on the idea in Aalst and Lassen (2008), were implemented. In Ning et al. (2007), an algorithm, known as BPELGEN, for converting a service workflow represented by a directed acyclic graph to BPEL is proposed. The resulting process is a fully blocked-structured process built only using flow and sequence activities. However, this method adds some execution constraints since a graph-based structure cannot be fully captured using block-based structure. For example in Fig. 4, the workflow graph generated by our approach for the example feature model configuration shown in Fig. 1 is presented. Fig. 15-B shows the result BPEL process generated by executing the BPELGEN algorithm. Although, this BPEL process is fully block-structured, it imposes some new constraints such as enforcing the execution of operation I after D or F after D which is not enforced in the workflow graph. In Aalst and Lassen (2008), a method for converting workflow-net which is a petri-net based language to BPEL process has been proposed. In this method workflow-net is converted into a BPEL process by consecutive folding of structures of workflow-net into BPEL activities. However, this method requires human intervention when it could not find any well-structured set of activities for folding as we discussed it before. Fig. 15-C shows the result of this method for the example in Fig. 4 before requiring human intervention. Fig. 15-D shows the resulting BPEL process generated by our proposed approach.

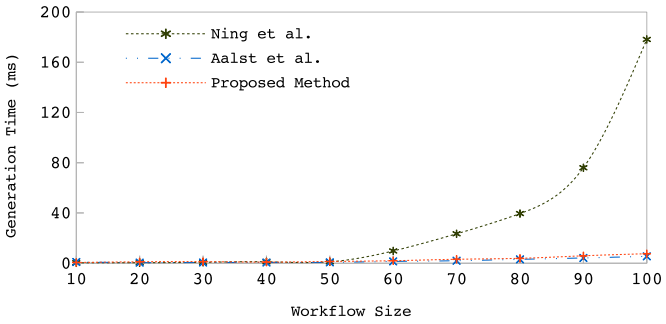

Fig. 14. Comparison between different BPEL generation methods in terms of generation time.

In order to perform the two experiments of this section, 50 different workflows are randomly selected and converted to BPEL using the three different methods. For the first experiment, the number of flow activities, sequence activities and link tags are collected and for the second experiment, the time to generate that BPEL is collected. For the method proposed in Aalst and Lassen (2008), the BPEL code right before requiring human intervention was considered to be the result.

Fig. 13 shows the average number of flows, sequence activities and link tags for BPEL code generated by the different methods. It can be seen that the number of links and total number of elements are highest for the method proposed in Aalst and Lassen (2008), which is expected considering the fact that this method cannot proceed when there is no well-structured set of activities in the BPEL code. Our proposed method creates processes with significantly smaller number of activities and smaller number of links compared to this method. Additionally, the BPELGEN method creates no links and a fewer number of activities compared to our proposed method. However as we mentioned before, this algorithm imposes dependencies which are not in the workflow graph which can negatively affect understandability of the workflow as well as some quality of service attributes such as time-to-completion.

Fig. 14 shows the average time required for generating BPEL processes using the different algorithms. Our proposed method and the method in Aalst and Lassen (2008) are able to generate the BPEL process in a very fast time (<10ms). The BPELGEN algorithm shows exponential increase in BPEL generation time which is expected

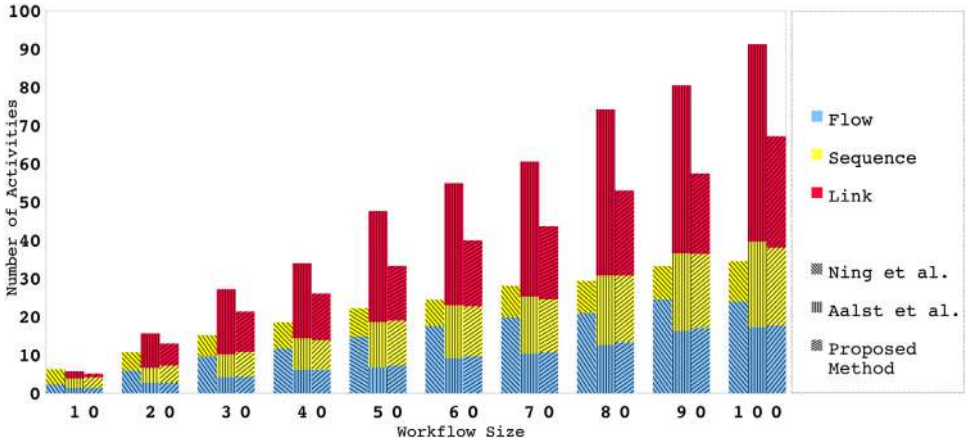

Fig. 13. Comparison between different BPEL generation methods in terms of element distribution.

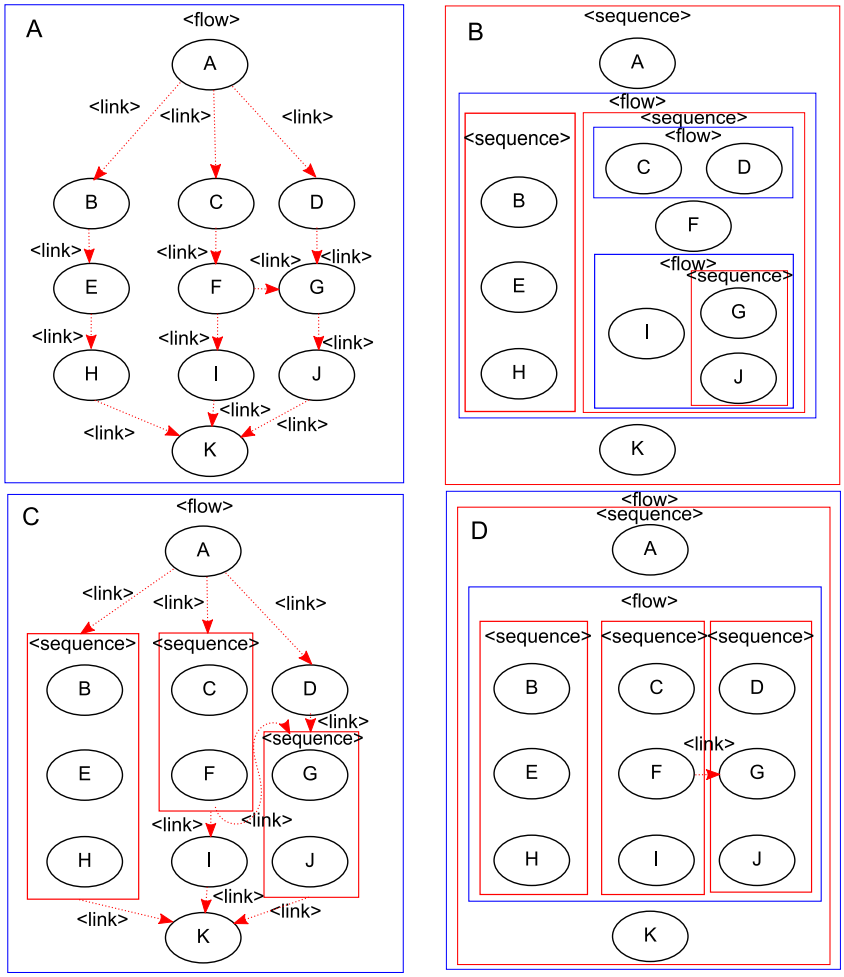

Fig. 15. Possible BPEL representation of workflow in Fig. 4.

considering that it requires depth-first traversal over the workflow graph in each iteration. Still, the time required for generating BPEL using this method remains acceptable (<200 ms) for the processes sizes tested in our experiments.

6.4. Threats to validity

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

6.4.1. Internal validity

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

- All the models which are used in these experiments are synthetically generated. While the models are synthetic, we have been rigorous in capturing and reporting all of the statistical distributions that were used in our tools for generating these models. Despite this, there can still be some unconfigurable parameters in these tools which could result in bias when creating the synthetic models, hence indirectly impacting the experimental results.
- In Experiment 2.1, planning was performed using the FF planner, which is a highly optimized planner written in C; however, for the sake of consistency, we have implemented the optimization logic using the Java programming language. The programming language as well as the optimization method can affect time-to-completion. Therefore, the reported values could change based on a different implementations.
- The implementation for the two methods used as baseline for BPEL process generation has not been publicly released by the authors. Therefore, we have implemented these two methods based on the algorithms provided in their corresponding papers. These implementations may have minor differences with the actual implementation of the authors, which may result in slightly different outputs. We have made the implementation of the baselines as well as the implementation of our proposed method available on the magus.online source repository for replication purposes.

6.4.2. External validity

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

- The results of our experiments may be different when applied on real-world domains considering the fact that we used synthetic domain models in our experiments, which may not truly capture the properties of a real problem domain. Therefore, evaluating the proposed method by addressing real-world problems has to be considered as a next step for our work.
- Experiment 2.2 assumes that the time-to-completion of all services has the same distribution. However, this may not be the case in real-world scenarios, which may affect the efficiency of the optimization method. Similar to the previous issue, evaluating the optimization method on a workflow defined over actual services may be able to better capture the efficiency of this method. Furthermore, the

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

7. Concluding remarks

In this paper, we have proposed an effective approach for the automated generation of service compositions for variability-intensive domains. Our work is positioned within the intersection of service-oriented architecture and software product line engineering. We benefit from the formal representation of feature models from the software product line engineering domain to represent the possible requirements space of the end users and feature model configurations to represent exact user requirements. We have benefited from the integration of feature models and services to translate functional requirements expressed as feature model configurations into service selections. The selected services are then automatically composed based on AI planning so that a workflow is generated. The generated workflow is not guaranteed to have desirable properties such as execution time. For this purpose, we have proposed a method that introduces parallelism into the workflow while maintaining its validity. The optimized workflow is then converted into WS-BPEL that is directly executable.

7.1. Future work

In the following, we discuss potential areas for our future work and the limitations that each area will be addressing.

- As a next step, we are planning to apply the proposed method in real-world problem domains. This includes the generation of the domain models based on the requirements of the environment and the automated generation of processes for specific application domains. This will enable us to perform more extensive evaluation of the practicality and usefulness of the proposed method, which can point to the limitations of our proposed method.
- One of the limitations of the proposed approach is that the process of linking services and features with appropriate annotations can be a complex and error prone process. As a next step, we will investigate the possibility of developing automated or semi-automated tools, which use existing information such as textual descriptions of the features to aid domain designers in developing domain models.
- Another limitation of the proposed method is that it assumes that the impact of features and services on the environment can be captured using a set of predicates, which may not always be the case. By applying the proposed method on real-world problems, we can evaluate how this assumption can affect the applicability of the proposed method. The result of such evaluation can be used to improve the proposed method.
- Given the fact that our work is based on software product line feature models, in our future work, we are interested in exploring the possibility of re-generating the workflow at runtime to support dynamic runtime self-adaption of the composition. This would provide the means to ensure that the workflow does not only respect functional requirements but is also aware of non-functional and quality of service requirements. We will investigate how runtime reconfiguration of the feature model can enable us to find alternative service compositions that satisfy functional and non-functional requirements.

References

Aalst, W.M.V.D., Lassen, K.B., 2008. Translating unstructured workflow processes to readable BPEL: theory and implementation. Inf. Softw. Technol. 50 (3), 131–159.

Agarwal, V., Chafle, G., Dasgupta, K., Karnik, N., Kumar, A., Mittal, S., Srivastava, B., 2005. Synthy: a system for end to end composition of web services. Web Semant. Sci. Serv. Agents World Wide Web Vol. 3, 311–339.

Alferez, G., Pelechano, V., Mazo, R., Salinesi, C., Diaz, D., 2014. Dynamic adaptation of service compositions with variability models. J. Syst. Softw. 24–47.

Asadi, M., Mohabbati, B., ner, G.G.A., Gasevic, D., 2014. Development and validation of customized process models. J. Syst. Softw. 96, 73–92.

Backstrom, C., 1998. Computational Aspects of Reordering Plans. J. Artif. Intell. Res. 9, 99–137.

Baresi, L., Guinea, S., Pasquale, L., 2012. Service-oriented dynamic software product lines. Comput. (Long Beach Calif) 45 (10), 42.

Basile, D., Beek, M.H.t., Giandomenico, F.D., Gnesi, S., 2017. Orchestration of dynamic service product lines with featured modal contract automata. Proceedings of the Twenty-First International Systems and Software Product Line Conference-Volume B. pp. 117–122.

Benavides, D., Segura, S., Trinidad, P., Ruiz-Cortes, A., 2007. Fama: tooling a framework for the automated analysis of feature models. Proceedings of first International Workshop on Variability Modelling of Software-Intensive Systems. pp. 129–134.

Bertoli, P., Pistore, M., Traverso, P., 2010. Automated composition of web services via planning in asynchronous domains. Artif. Intell. 174 (3), 316–361.

Cappiello, C., Matera, M., Picozzi, M., Sprega, G., Barbagallo, D., Francalanci, C., 2011. Dashmash: a mashup environment for end user development. Web Engineering. Springer, pp. 152–166.

Cesari, L., Pugliese, R., Tiezzi, F., 2010. A tool for rapid development of WS-BPEL applications. ACM SIGAPP Appl. Comput. Rev. 11 (1), 27–40.

Chafle, G., Das, G., Dasgupta, K., Kumar, A., Mittal, S., Mukherjea, S., Srivastava, B., 2007. An integrated development environment for web service composition. Proceedings of the IEEE International Conference on Web Services. IEEE, pp. 839–847.

Cremaschi, M., De Paoli, F., 2017. Toward automatic semantic api descriptions to support services composition. Proceedings of the European Conference on Service-Oriented and Cloud Computing. pp. 159–167.

Daniel, F., Casati, F., Benatallah, B., Shan, M.C., 2009. Hosted Universal Composition: Models, Languages and Infrastructure in Mashart, Conceptual Modeling-ER 2009. Springer, pp. 428–443.

Deng, S., Hongyue, W., Daning, H., Zhao, J.L., 2016. Service selection for composition with QoS correlations. IEEE Trans. Serv. Comput. 9 (2), 291–303.

Fikes, R.E., Nilsson, N.J., 1972. Strips: a new approach to the application of theorem proving to problem solving. Artif. Intell. 2 (3), 189–208.

Fujii, K., Suda, T., 2006. Semantics-based dynamic web service composition. Int. J. Cooperat. Inf. Syst. 15 (03), 293–324.

Gibb, B.K., Damodaran, S., 2002. EbXML: Concepts and Application. John Wiley and Sons, Inc.

Gröner, G., BošKović, M., Parreiras, F.S., GašEvić, D., 2013. Modeling and validation of business process families. Inf. Syst. 38 (5), 709–726.

Gtz, M., Roser, S., Lautenbacher, F., Bauer, B., 2009. Token analysis of graph-oriented process models. Proceedings of the Thirteenth IEEE Enterprise Distributed Object Computing Conference Workshops, EDOCW. pp. 15–24.

Hatzi, O., Vrakas, D., Bassiliades, N., Anagnostopoulos, D., Vlahavas, I., 2013. The porsce ii framework: using ai planning for automated semantic web service composition. Knowl. Eng. Rev. 28 (02), 137–156.

Haugen, O., Moller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A., 2008. Adding standardized variability to domain specific languages. Proceedings of the International Software Product Line Conference (SPLC). IEEE, pp. 139–148.

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

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

Jiang, W., Zhang, C., Huang, Z., Chen, M., Hu, S., Liu, Z., 2010. Qsynth: a tool for qos-aware automatic service composition. Proceedings of the International Conference on Web Services. IEEE, pp. 42–49.

Johnson, R., Pearson, D., Pingali, K., 1994. The program structure tree: computing control regions in linear time. Proceedings of the ACM SigPlan Notices, Vol. 29. ACM, pp. 171–185.

Klusch, M., Gerber, A., Schmidt, M., 2005. Semantic web service composition planning with owls-xplan. Proceedings of the AAAI Fall Symposium on Semantic Web and Agents.

Lassen, K.B., van der Aalst, W.M., 2009. Complexity metrics for workflow nets. Inf. Softw. Technol. 51 (3), 610–626.

Lee, J., Kotonya, G., 2010. Combining service-orientation with product line engineering. Software. Vol. 27. IEEE, pp. 35–41.

Lee, K., Kang, K., Lee, J., 2002. Concepts and guidelines of feature modeling for product

line software engineering. Softw. Reuse Methods Tech.Tools 62–77.
Lemos, A.L., Daniel, F., Benatallah, B., 2016. Web service composition: a survey of techniques and tools. ACM Comput. Surv. 48 (3), 33.
Leymann, F., Web Services Flow Language (wsfl 1.0), Tech. rep., IBM Corporation (2001). URL http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.
Lian, X., Zhang, L., Jiang, J., Goss, W., 2018. An approach for optimized feature selection in large-scale software product lines. J. Syst. Softw. 137, 636–651.
Liu, X., Hui, Y., Sun, W., Liang, H., 2007. Towards service composition based on mashup. Proceedings of 2007 IEEE Congress on Services. IEEE, pp. 332–339.
Mahdi, B., Ebrahim, B., Weichang, D., 2016. Automated composition of service mashups through software product line engineering. Proceedings of the Fifteenth International Conference on Software Reuse. pp. 20–38.
Mahdi, B., Ebrahim, B., Weichang, D., 2017. Self-healing in service mashups through feature adaptation. Proceedings of the Twenty-First International Systems and Software Product Line Conference, SPLC 2017, Volume A. pp. 94–103.
Mayer, S., Verborgh, R., Kovatsch, M., Mattern, F., 2016. Smart configuration of smart environments. Proceedings of the IEEE Transactions on Automation Science and Engineering. Vol. 13. pp. 1247–1255.
McAllester, D., Rosenblatt, D. Systematic nonlinear planning. MIT AI Memo, No 1339, Dec 1991.
Mcdermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D., 1998. Pddl - the planning domain definition language, Tech. Rep., CVC TR-98-003/DCS TR-1165, Yale center for computational vision and control.
McDermott, D.V., 2002. Estimated-regression planning for interactions with web services. Proceedings of the International Conference on Artificial Intelligence Planning and Schedule Systems. 2. pp. 204–211.
Medeiros, F.M., De Almeida, E.S., de Lemos Meira, S.R., 2009. Towards an approach for service-oriented product line architectures. Proceedings of the Workshop on Service-oriented Architectures and Software Product Lines. pp. 1–7.
Menasc, D.A., Casalicchio, E., Dubey, V., 2008. A heuristic approach to optimal service selection in service oriented architectures. Proceedings of the Seventh International Workshop on Software and Performance. ACM, pp. 13–24.
Mendonca, M., Branco, M., Cowan, D., 2009. Splot: software product lines online tools. Proceedings of the Twenty-Fourth ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and applications. ACM, pp. 761–762.
Microsoft. Windows workflow foundation. URL http://msdn.microsoft.com/en-us/library/jj684582.aspx.
Minton, S., Bresina, J.L., Drummond, M., 1994. Total-order and partial-order planning: a comparative analysis. J. Artif. Intell. Res. 2, 227–262.
Narwane, G.K., Galindo, J.A., Krishna, S.N., Benavides, D., Millo, J.V., Ramesh, S., 2016. Traceability analyses between features and assets in software product lines. Entropy 18 (8), 269.
Ngu, A.H.H., Carlson, M.P., Sheng, Q.Z., Paik, H.y., 2010. Semantic-based mashup of composite applications. IEEE Trans. Serv. Comput. 3 (1), 2–15. ID: 1
Nguyen, X., Kambhampati, S., 2001a. Reviving partial order planning. Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence-Volume 1. Morgan Kaufmann Publishers Inc., pp. 459–464.
Nguyen, X., Kambhampati, S., 2001. Reviving partial order planning. Int. Joint Conf. Artif. Intell. 1, 459–464.
Ning, G., Zhu, Y., Lu, T., Wang, F., 2007. Bpelgen: an algorithm of automatically converting from web services composition plan to BPEL4WS. Proceedings of the Second International Conference on Pervasive Computing and Applications. IEEE, pp. 600–605.
Noorian, M., Bagheri, E., Du, W., 2017. Toward automated qualitycentric product line configuration using intentional variability. J. Softw. Evol. Process 29 (9). https://doi.org/10.1002/smr.1870.
OMG, 2009. Business process modeling notation specification, version 1.2, Tech. Rep., OMG, formal/09-01-03 (January 2009). URL http://www.omg.org/spec/BPMN/1.2.
Ouyang, C., Dumas, M., 2006. From BPMN process models to Bpel web services. Proceedings of the IEEE International Conference on Web Services (ICWS'06). IEEE, pp. 285–292.
Ouyang, C., Dumas, M., Hofstede, A.H.T., Aalst, W.M.V.D., 2008. Pattern-based translation of BPMN process models to Bpel web services. Int. J. Web Serv. Res. 5 (1), 42.
Peer, J., 2005. Web Service Composition as AI Planning–A Survey. University of St. Gallen, Switzerland, pp. 1–63.
Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P., 2004. Planning and monitoring web service composition. Artificial Intelligence: Methodology, Systems, and Applications. Springer, pp. 106–115.
Pohl, K., Bockle, G., Linden, F.V.D., 2005. Software product line engineering: foundations. Principles and Techniques, Vol. 10. Springer.
Rao, J., Kungas, P., Matskin, M., 2004. Logic-based web services composition: from service description to process model. Proceedings of the IEEE International Conference on Web Services. IEEE, pp. 446–453.
Reijers, H.A., Mendling, J., 2011. A study into the factors that influence the understandability of business process models. Proceedings of the IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans. Vol. 41. pp. 449–462.
Reijers, H.A., Mendling, J., Dijkman, R.M., 2011. Human and automatic modularizations of process models to enhance their comprehension. Inf. Syst. 36 (5), 881–897.
Rodriguez-Mier, P., Manuel, M., Manuel, L., 2017. Hybrid optimization algorithm for large-scale QoS-aware service composition. IEEE Trans. Serv. Comput. 10.4, 547–559.
Rodriguez-Mier, P., Mucientes, M., Lama, M., 2011. Automatic web service composition with a heuristic-based search algorithm. Proceedings of the IEEE International Conference on Web Services. IEEE, pp. 81–88.
Rosa, M.L., Van Der Aalst, W.M.P., Dumas, M., Milani, F.P., 2017Rosa, van der Aalst, Dumas, Milani. Business Process Variability Modeling: A Survey. ACM Comput. Surv. 50 (1). https://doi.org/10.1145/3041957. Article 2, 45 pages.
Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S., Xu, X., 2014. Web services composition: a decades overview. Inf. Sci. (Ny) 280, 218–238.
Sheth, A.P., Gomadam, K., Lathem, J., 2007. Sa-rest: semantically interoperable and easier-to-use services and mashups. IEEE Internet Comput. 11 (6), 91–94.
Siddiqui, F.H., Haslum, P., 2013. Plan quality optimisation via block decomposition. Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. AAAI Press, pp. 2387–2393.
Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D., 2004. Htn planning for web service composition using shop2. Web Semant. Sci. Serv. Agents World Wide Web 1 (4), 377–396.
Soltani, S., Asadi, M., Gasevic, D., Hatala, M., Bagheri, E., 2012. Automated planning for feature model configuration based on functional and non-functional requirements. Proceedings of the Sixteenth International Software Product Line Conference-Volume 1. ACM, pp. 56–65.
Syu, Y., FanJiang, Y.Y., Kuo, J.Y., Ma, S.P., 2011. Towards a genetic algorithm approach to automating workflow composition for web services with transactional and Qos-awareness. Proceedings of the IEEE World Congress on Services (SERVICES). IEEE, pp. 295–302.
Syu, Y., Ma, S.P., Kuo, J.Y., FanJiang, Y.Y., 2012a. A survey on automated service composition methods and related techniques. Proceedings of the IEEE Ninth International Conference on Services Computing (SCC). IEEE, pp. 290–297.
Syu, Y., Ma, S.P., Kuo, J.Y., FanJiang, Y.Y., 2012b. A survey on automated service composition methods and related techniques. Proceedings of the Ninth IEEE International Conference on Services Computing (SCC). IEEE, pp. 290–297. ID: 1
Thatte, S., 2001. XLANG: web services for business process design. Microsoft Corp Technical Report.
Thiagarajan, R.K., Srivastava, A.K., Pujari, A.K., Bulusu, V.K., 2002. Bpml: a process modeling language for dynamic business models. Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, 2002. IEEE, pp. 222–224.
Tizzei, L.P., Marcelo, N., Segura, V.A.-c.C., Cerqueira, R.F.G., 2017. Using microservices and software product line engineering to support reuse of evolving multi-tenant saas. Proceedings of the Twenty-First International Systems and Software Product Line Conference-Volume A. pp. 205–214.
Traverso, P., Pistore, M., 2004. Automated composition of semantic web services into executable processes. The Semantic WebISWC 2004. Springer, pp. 380–394.
van der Aalst, W.M., Argensen, J.B.J., Lassen, K.B., 2005. Lets go all the way: from requirements via colored workflow nets to a Bpel implementation of a new bank system. Proceedings of the OTM Confederated International Conferences on the move to Meaningful Internet Systems. Springer, pp. 22–39.
Yuan, P., Jin, H., Yuan, S., Cao, W., Jiang, L., 2008. Wftxb: a tool for translating between Xpdl and Bpel. Proceedings of the Tenth IEEE International Conference on High Performance Computing and Communications, 2008, IEEE. pp. 647–652.

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

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

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