Product-Line-Based Requirements Customization for Web Service Compositions

Hongyu Sun[1], Robyn Lutz[1,2] and Samik Basu[1]
[1]*Department of Computer Science, Iowa State University*
Ames, IA, 50011-1040, USA
[2]*Jet Propulsion Laboratory/Caltech*
{sun, rlutz, sbasu}@cs.iastate.edu

Abstract

Customizing web services according to users' individual functional and non-functional requirements has become increasingly difficult as the number of users increases. This paper introduces a new way to customize and verify composite web services by incorporating a software product-line engineering approach into web-service composition. The approach uses a partitioning similar to that between domain engineering and application engineering in the product-line context. It specifies the options that the user can select and constructs the resulting web-service compositions. By first creating a web-service composition search space that satisfies the common requirements and then querying the search space as the user selects values for the parameters of variation, we provide a more efficient way to customize web services. A decision model, illustrated with examples from an emergency-response application, is created to interact with the customers and ensure the consistency of their specifications. The capability to reuse the composition search space may also help improve the quality and reliability of the composite services and reduce the cost of re-verifying the same compositions.

1. Introduction

Commercial web services often have a very large user base. How to customize these web services according to the users' individual requirements becomes increasingly difficult as the number of users increases. This paper introduces a new way to customize composite web services to users' requirements by applying a software product line engineering approach to the web service composition domain.

Most existing practical mechanisms for synthesizing composite services have been deployed taking into consideration their desired functional requirements [1, 7]. Functional requirements (FRs) describe how a system should behave during operation, while non-functional requirements (NFRs) describe constraints on the quality attributes of the system's operation. NFRs can be broadly classified as soft and hard constraints. Hard constraints refer to the set of NFRs that must be satisfied by a composite service, while soft constraints deal with user preferences and trade-offs among NFRs [6]. In this paper, we are concerned with hard constraints.

In order to customize a web service in a specific domain application (e.g. a travel agency service), FRs and NFRs may both vary somewhat among individuals. With existing methods of composing web services on FRs and NFRs [2, 7], the verification of the variations in the services tends to incur a heavy overhead that can reduce the performance of the composed services. We instead seek to exploit the reuse of certain web service compositions, where their composition has already been verified. The goal is for each small variation to no longer trigger an entirely new verifiable composition process. Thus, for customizable web services, we need a lightweight and low-overhead solution to generating web service compositions satisfying users' customized requirements.

A web service composition shares several similarities with a product line. A commercial web service provider has a specified, shared set of functional and non-functional requirements that should be fulfilled by every possible service composition provided to the user. For example, a travel agency web service has to include a membership service and a payment service as common functionalities. Secured web service communication is also a common NFR that exists in all compositions.

These common FRs and NFRs can be mapped to commonalities in a product line by taking all possible compositions that satisfy the common FRs and NFRs as products of a product line. Each user of the web service can impose customized FRs and NFRs, according to his or her own preference, within the variations provided by, e.g., the travel-agency service. Thus, a user may require flights to be on a specific airline or airplane model for the flight-booking service. These variations in FRs and NFRs, if predictable by the service provider, can be specified as variabilities of a product line. The user then only needs to decide the value for each variation point in order to generate a customized service composition.

In previous work [2], we have separated the verification of NFRs for a web service composition from the verification of FRs in order to reduce the complexity of requirements verification. In pursuing a two-stage solution we have subsequently tried to isolate the requirements likely to be changed, or customized, by the users. Product line engineering offers a strong framework for this purpose.

Software product line engineering (SPLE) identifies the common assets of a series of products as commonalities and views the optional and alternative assets as variabilities. Taking advantages of these two concepts, SPLE reduces the development cost and time by reusing the commonalties and the variabilities across the product line [3]. We build in this work on the FAST (Family-Oriented Abstraction, Specification, and Translation) approach to constructing a product line. FAST uses three key artifacts to enable rapid generation of a product given a partially ordered specification from the customer [3][4]:

1. The *commonality and variability analysis* (CVA), which identifies the commonalities and variabilities for the product line and the dependencies and constraints among them.
2. The *mapping relations* between the values of the variations and the modules of the product lines (which may be one-to-one, one-to-many, or many-to-many mappings) and the *uses-relations* of the modules and their implementations (which describe the other modules used if one module is selected). For example, one module may have different ways of being implemented under the constraints of performance or platforms.
3. The *decision model* with which the customer interacts to specify and construct a new product in the product in the product line.

We extend the process by adapting the workflow used to compositionally generate a product-line system to the customized composition of web services:

- The user decides the values of the variabilities of a product in either a random or preferred sequence.
- The values of these variabilities are used to prune and determine other variabilities by following the dependencies and constraint rules.
- A consistent value set for the variabilities is thus obtained. By tracing the mapping between the values of variabilities and the components of the product line, a component set is selected.
- By tracing the uses-relations of the selected component set, all necessary components for a product are selected.
- The implementation of the necessary component set is gathered, compiled and published as the final product.

Web service composition takes advantage of SOA (Service Oriented Architecture) [5] to achieve complex functional and non-functional requirements by selecting and composing qualified component services provided by service providers. A service broker (UDDI) is responsible for the registration and look-up of all component services with descriptions.

By considering a customizable web service composition as a product line, we are better able to handle the customized requirements from the users. Normally, web service compositions involve verification of the functional and non-functional requirements on demand. This means that whenever there is a variation, change, or new requirement for the services, the composition algorithm needs to re-compose and re-verify the web services against the variation, change or new requirement.

Using the SPLE approach, we show how we can successfully divide the composition and verification process into two stages. The first stage treats the construction of the functional *commonalities* of the web service compositions and verifies them against the common NFRs. The second stage takes as input the verified compositions from the first stage and then verifies the candidate composition in the result set to meet the *variable* functional and non-functional requirements from the user.

Since part of the computation has already been done in the first stage as pre-processing, the verification overhead caused by the customizations in the mass-user-based web services is much lower than the traditional one-stage composition and verification process. The two-stage SPLE approach has better efficiency because, for a customization requirement, the second stage acts like a service composition verifier rather than a service composition generator. It essentially hides the large computation overhead in the first stage when constructing the search space for the second stage.

The rest of the paper is organized as follows. Section 2 introduces related work that helps in understanding our work. Section 3 describes our approach to applying SPLE to web service composition and illustrates it by application to an emergency-response service. Section 4 discusses the advantages and limitations of our approach and describes the steps needed for future evaluation. Section 5 provides a brief conclusion.

2. Related Work

Two types of web service composition mechanism are widely used: the choreography-based composition and the orchestration-based composition [1]. The choreography-based composition assumes that there is a central controller, the choreographer, to interact with each component services and the service user. The orchestration-based composition is more like a peer-to-peer network. The component services directly interact with each other without a center that manages all the transactions. In BPEL4WS, this is realized by automated execution of the service workflow. In this work we use choreography-based composition. More information on web service composition can be found in the surveys by Duster and Schreiner [1], Kohler and Srivastava [7] and Milanovic and Malek [8].

Web service composition can be divided into two problems from the viewpoint of the requirements: how to compose services with the functional requirements and how to compose services with the NFRs. For web service composition on functional requirements, Foster, Uchitel, Magee and Kramer [9] introduced a model-based approach (LTSA-WS) to verifying the composition implementations. All coordination and obligation specifications were modeled in message sequence chart (MSC) and then translated into finite state process (FSP) algebra. The verification mechanism was a trace equivalence check [10].

Pathak, Basu and Honavar [11] introduced a tool-supported approach (MoSCoE) for web service compositions. It uses a forward-backward web service composition algorithm to verify functional requirements and some non-functional properties. The functional requirements are represented by the goal automaton. Later work [2] further explored the verification of NFRs by also modeling the NFRs as automata. The NFRs were verified by composing the property automata with the service automata.

By treating web service compositions as a product line, we can apply approaches in software product line engineering (SPLE) to the web service composition domain. SPLE takes advantage of the concept of commonalities in a product family to form a product line. By distinguishing the commonalities and variabilities, a product in the product line can be viewed as reuse of the common assets with different variable assets.

The SPLE process consists of two phases: domain engineering phase and application engineering phase. In order to simplify the product generation process and hide the background complexity, a decision model is created during the domain engineering phase and reused during the application engineering phase. We here use the fully constructed decision model introduced in [4] with all its background relation models to generate verifiable compositions of customizable web service families.

Several researchers have applied product line engineering (PLE) to the web service domain in different ways. Karam. Dascalu, Safa, Santina and Koteich, [12] incorporated PLE into web service-based applications (WSbWAs)s. The web applications benefited from the reconfigurable, reusable pages, workflows and web services (WebPads as composite web services). These supported the common artifacts of the web development domain and the particular aspects of the application in the domain. Their work focused on the reuse of functionalities during product evolution rather than on the NFRs of the web applications.

Balzerani, Di Ruscio and Pierantonio [13] followed the FODA (Feature-Oriented Domain Analysis) [14] approach to SPLE to construct a reusability-oriented web application architecture. It took the bounded input parameters of the functional methods as variation points. New requests from the user were considered as different variation values for the product line to enable both design time reconfiguration and runtime reconfiguration. However, this work lacked a clear explanation of the domain engineering phase to distinguish the commonalities of the web applications from the variabilities.

Capilla and Topaloglu [15] introduced a way of applying SPLE into web service composition. The authors identified types of variation points that can be used in a web service based product line: the order of the composition in an orchestration composition, the flow conditions in a message path, the service alternatives, the exception handling possibilities and the quality of service choices. These types of variations served to customize web services during the design and implementation phases.

Our work continues the idea of applying SPLE approaches to solve existing problems in web service composition. The goal here is to improve the customization efficiency of domain-specific, mass-user based, customizable web services. By constructing a decision model [3, 4], we can hide the complexity of domain and application knowledge from the user and give the user a trouble-free way of generating a customized web service. By distinguishing the commonalities and the variabilities of the web services, we can successfully divide the web composition into two stages: the preparation stage (to construct all commonalities) and the customization stage (to set all variabilities). We thus draw most of the computation overhead into the first stage during the design to enable improved runtime efficiency during the second stage.

3. Approach

Figure 1 shows an overview of our approach, which is described briefly here. Subsequent subsections provide a more detailed account of each of the major steps in Fig. 1. The steps are labeled in the figure with the number of the subsection that describes that step.

Our approach contains two workflows, one for the web service product developer (shown at the top of the figure) and one for the user (shown at the bottom). This approach provides a two-stage process. From the point of view of product line development, the development process is divided into the domain engineering phase and the application engineering phase. From the point of view of web-service design, the process is divided into the preparation stage (before implementation) and the customization stage (after implementation).

At the top half of the figure, i.e., the domain engineering phase, the developer performs a commonality and variability analysis (CVA) of the system requirements. The results of the CVA include a formalized specification of all the common and variable functional and non-functional properties. Each variability has some associated parameters to configure, called *parameters of variation*. Some of these parameters may have dependencies or tradeoffs with other parameters of variation. We model this dependency of variations in a *variability dependency graph*.

Each commonality and parameter of variation is also associated with a set of service compositions that satisfy them. We call this relation the *mapping-relation*. The results of the CVA are captured in a product-line decision model. Given the system requirements, the developer can identify the component web services that are relevant to the domain and the system.

For the application engineering phase, the services retrieved from the service broker are composed according to the common functional requirements by means of a modified version of the goal model and a functional service composition algorithm from [16]. The algorithm outputs all possible service compositions that satisfy the common functional requirements. By verifying all the compositions of this set against the common NFRs, we prune the composition set into a smaller set in which each composition satisfies all commonalities. This set serves as the composition search space for the variability, and we call it the *commonality composition set* (subset).

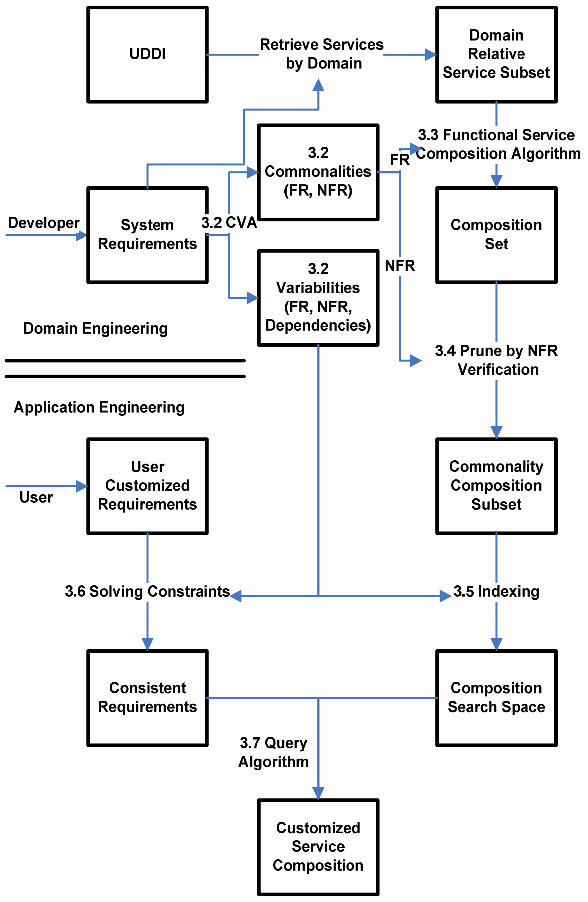
UDDI
Retrieve Services by Domain
Domain Relative Service Subset
3.2 Commonalities (FR, NFR)
3.3 Functional Service Composition Algorithm
FR
Developer
System Requirements
3.2 CVA
Composition Set
3.2 Variabilities (FR, NFR, Dependencies)
NFR
Domain Engineering
3.4 Prune by NFR Verification
Application Engineering
User
User Customized Requirements
Commonality Composition Subset
3.6 Solving Constraints
3.5 Indexing
Consistent Requirements
Composition Search Space
3.7 Query Algorithm
Customized Service Composition

Figure 1. Overview of Approach

In order to improve the verification efficiency for the runtime customization, we do an indexing on each parameter of variation mapping to a subset of the search space. A composition subset associates with a parameter of variation if and only if all compositions in this subset satisfy the variability set by this parameter. The result of this preparation phase is the composition search space that is ready for runtime user customization.

After implementation of the web services, the user can access a default web service with basic functionalities. The user can then customize the functionalities and non-functional properties by setting all the parameters of variation in the decision model. By interacting with the decision model, the input requirements from the user are always kept consistent through solving the constraints in the variability dependency graph. The consistent specification of the variabilities is fed into a query algorithm to search valid compositions in the composition search space.

The output of this algorithm is either the customized service composition satisfying all the requirements from the user or a report to the user of a failed composition attempt.

3.1 Illustrative Example

We constructed a small system, the Emergency Management System (EMS), based on [17], to illustrate the basic concepts of our approach.

EMS consists of several different units: the Field Officer Service, the Request Dispatch Service (Dispatcher for short) and services for emergency handling, including an Ambulance Dispatch Service, a Fire Station Dispatch Service and a Police Dispatch Service. The functional requirements are to dispatch ambulance(s), fire truck(s) and police to a location upon request. These requests are specified and sent by the Field Officer through a service in a mobile terminal or a PDA.

The Dispatcher has three types: the normal dispatch service, which is responsible for routine situations; the speed-line dispatch service, which has low communication delay, compared to the normal dispatcher and is used for urgent dispatches; and the highly-secured dispatcher, which is used for national security related cases. The dispatcher service can also invoke a GPS-MAP service and other third party services. Figure 2 shows a sample structure for EMS service composition. The services connected by the dashed lines represent the optional services in a valid composition.

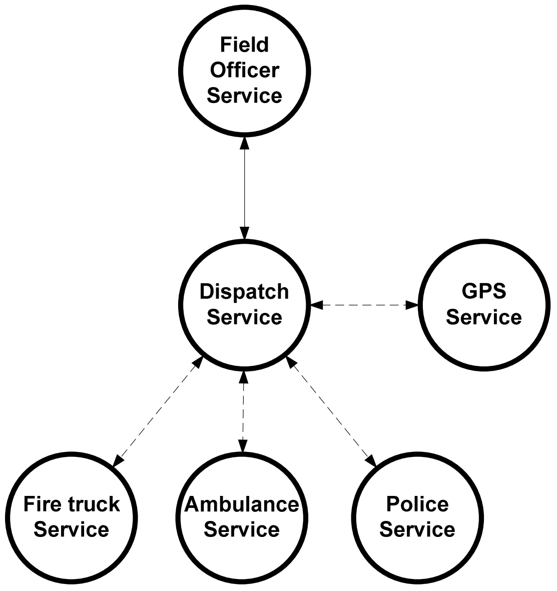

Figure 2. Sample Composition for EMS

3.2 Commonality and Variability Analysis

The commonalities that are shared across all of the EMS service compositions are listed according to their label, type (Functional requirements as "FR" and Non-functional requirements as "NFR"), their description and their service mappings.

Commonalities:

- C1
  - FR
  - There must be a Field Officer Service.
  - Field Officer Service = any service in Field Officer Service set {FO1, FO2, FO3…}.
- C2
  - FR
  - There must be a Dispatcher Service.
  - Dispatch Service = any service in normal Dispatcher Service set {DSN1, DSN2, …} or speed-line Dispatcher Service set {DSSL1, DSSL2, …} or Highly-secured Dispatcher Service set {DSHS1, DSHS2, …}.
- C3:
  - FR
  - There must be one or more emergency services.
  - An emergency service = any service in the Fire Station Dispatch Service set {FS1, FS2, …} or in the Ambulance Dispatch Service set {AS1, AS2, …} or in the Police Dispatch Service set {PS1, PS2, …}.
- C4:
  - NFR
  - All services must support at least 128-bit encryption in its service description.

The variabilities of the EMS are listed according to their label, type, description, parameter(s) of variation and any dependencies among these parameters.

Variabilities:

- V1:
  - FR
  - Type of dispatch service
  - {Normal, Speed-line, Highly-secured}.
  - If V1 is Highly-secured, then V6 is High. If V1 is Speed-line, V7 is Low.
- V2:
  - FR
  - Existence of Fire Station Dispatch service
  - {True, False}
  - If V2 is False, (V3 or V4) is True
- V3:
  - FR
  - Existence of Ambulance Dispatch service,
  - {True, False}.
  - If V3 is False, (V2 or V4) is True

- V4:
  - FR
  - Existence of Police Dispatch service
  - {True, False}.
  - If V4 is False, (V2 or V3) is True
- V5:
  - FR
  - Existence and type of a third-party service.
  - {N/A, GPS-MAP, TP1, TP2,…}.
  - If V5 is N/A, V8 is N/A.
- V6:
  - NFR
  - Security Level
  - {Medium, Med-High, High}.
  - If V6 is High, V1 is Highly-Secured and V9 is 256. If V6 is Med-High, V9 is 256. If V6 is Medium, V9 is 128.
- V7:
  - NFR
  - Delay of service communication
  - {Low, Med-Low, Medium}.
  - If V7 is Low, V1 is Speed-line.
- V8:
  - NFR
  - Range constraint between the field officer and the emergency service location
  - {N/A, Near, Medium, Far}.
  - If V8 is not N/A, V5 is GPS-MAP.
- V9:
  - NFR
  - Type of encryption
  - {128, 256}. If V9 is 128, V6 is Medium. If V9 is 256, V6 is Med-High or High.

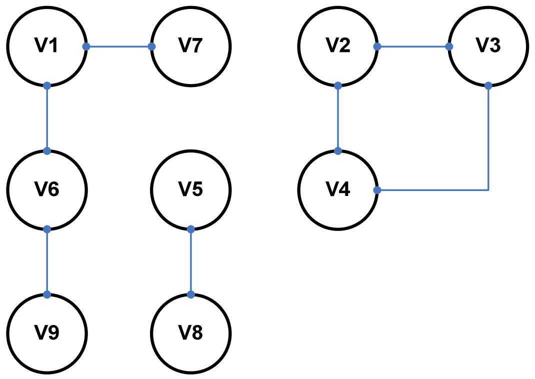

Figure 3. Dependency graph of variabilities

We model the dependencies of the parameters in the CVA in a dependency graph (see Figure 3). In the dependency graph, each node represents a variability and contains the information of all the parameters related to this variability. The edges between the nodes represent and contain the constraints between the two variabilities. The graph may not be fully connected. A graph-walk algorithm (described in Section 3.6) is used to traverse the sub-graphs in a random order or a user-preferred order to solve all the constraints.

To reduce the workload for the later verification process, we typically design the parameters of variation by translating integer or real number values into enumerated parameters, as is done with the Range Constraint V8, above. The range can be an integer of miles from 1 to 100, but is represented as only three values: Near (within 10 miles), Medium (10 to 50 miles) and Far (51 to 100 miles).

3.3 Goal Model of the Common Functionalities

In order to generate all the service compositions that satisfy the commonalities, we need to represent the common functionalities, here C1, C2 and C3, in the goal model.

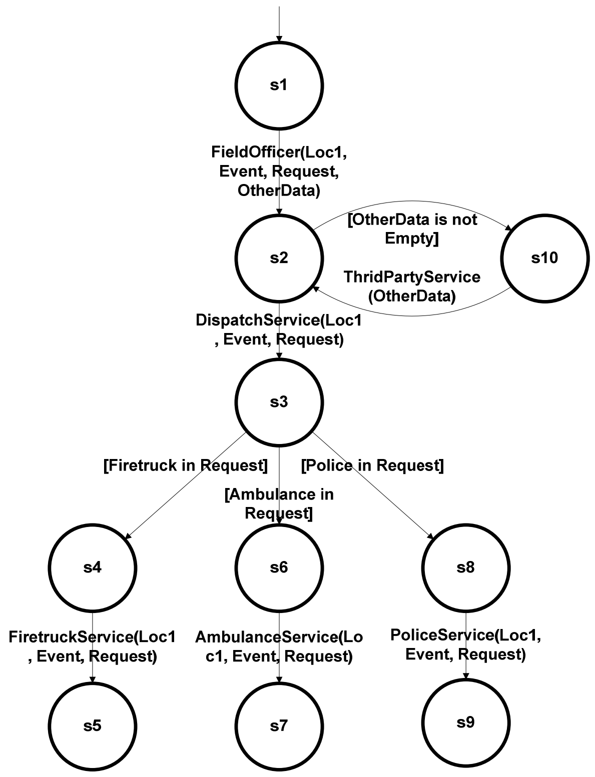
s1
FieldOfficer(Loc1, Event, Request, OtherData)
s2
[OtherData is not Empty]
ThridPartyService (OtherData)
s10
DispatchService(Loc1 , Event, Request)
s3
[Firetruck in Request]
[Ambulance in Request]
[Police in Request]
s4
s6
s8
FiretruckService(Loc1 , Event, Request)
AmbulanceService(Lo c1, Event, Request)
PoliceService(Loc1, Event, Request)
s5
s7
s9

Figure 4. Goal Model for Common Functionalities

Figure 4 shows the goal model for the common functionalities in EMS. Note that we have preserved the possibility of using third-party services to implement the functionality ThirdPartyService (OtherData) in the goal model. The functional composition process finds appropriate component services to implement these abstract methods in the goal model in accordance with the service mapping table. This service mapping table is constructed together with the goal model by looking up the CVA results. For example, a mapping is: ThirdPartyService = any element from {N/A, GPS-MAP, TP1, TP2, …}.

We apply a variant of the choreography-based web service composition algorithm from our previous work [16] on this goal model. The change is that we here take advantage of the availability of the service mapping table to generate all the compositions that satisfy the FR commonalities rather than generating just one such composition. The result of this step, as shown in Fig. 1, is a web service Composition Set containing all compositions satisfying the common functional requirements.

3.4 Verification Algorithm of Non-functional Properties

The next step toward constructing a search space is to verify the common NFRs. The automation in Figure 5 shows an example of an NFR that requires that any ambulance service in the composition shall respond within 600 seconds.

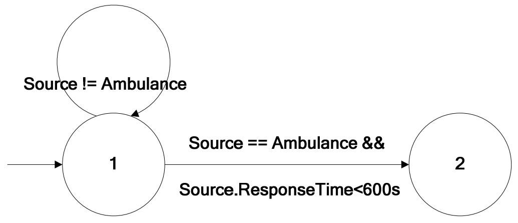

Figure 5. Transition guard with constraint on the service attributes

In previous work we have described how to model both a service composition and a non-functional property as automata and how to verify the NFR property by composing the two automata [2]. The verification is an automata equivalence check. This verification method unifies the property models by converting all liveness properties into safety properties. It also handles cases where there are multiple properties to be verified.

We define a finite state automata as a tuple FSA = (S, s_0, Δ, P, F) where S is the finite set of states, $s_0 \in S$ is the start state, and $\Delta \subseteq S \times 2^P \times S$ is the transition relation of the form $s - \phi -> s'$ such that $s, s' \in S$, and $\phi \in 2^P$ is a subset of propositions P. Finally, $F \subseteq S$ is the set of final states.

Given two automata, $FSA_i = (S_i, s_{0i}, \Delta_i, P_i, F_i)$, for $i \in \{1, 2\}$, their product is another FSA denoted by $FSA_1 \times FSA_2 = (S_{12}, s_{012}, \Delta_{12}, P_{12}, F_{12})$, where $S_{12} \subseteq S_1 \times S_2$, $s_{012} = (s_{01}, s_{02})$, $P_{12} = P_1 \cup P_2$, $F_{12} = \{(s_1, s_2) \mid s_1 \in F_1, s_2 \in F_2\}$. Finally, $s_1 - \phi_1 -> s'_1 \in \Delta_1$ and $s_2 - \phi_2 -> s'_2 \in \Delta_2$ and $(s_1, s_2) - \phi_1 \wedge \phi_2 -> (s'_1, s'_2) \in \Delta_{12}$.

Figure 6 shows an example of the product of two automata with the service composition automaton on the left, the property automation in the middle and their product on the right. During the calculation of the automata product, if a trap state of the safety property is reached, the verification has failed. Otherwise, the property is satisfied by the candidate composition.

For example, by applying this verification technique for the common NFR C4 to every composition in the Composition Set retrieved in 3.3, we prune out any compositions with service that violates the requirement for at least 128-bit encryption. The result of performing this verification on the other NFRs, as well, is a set of candidate compositions that satisfies all commonalities, both functional and non-functional, labeled the Commonality Composition Subset in Fig. 1.

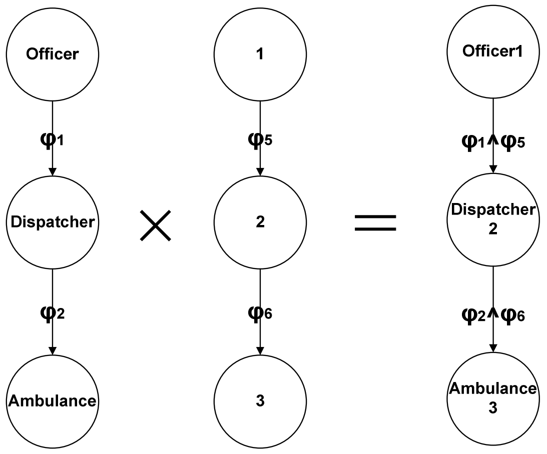

Figure 6. Product of a service composition and a property

3.5 Search Space Construction by Indexing

In order to construct the composition search space from the Commonality Composition Subset, we next need to index every parameter of variation. By indexing, we mean the creation of a mapping from each parameter of variation to a further subset of the Commonality Composition Subset such that that any composition in this subset satisfies the variability set by this parameter. To do this, we apply the verification technique introduced in 3.4 to prune the common set to this subset. The results are stored in the Search Space table for each parameter of variation. If the value for a parameter of variation depends on other parameters of variation, we also check those dependencies, as shown in the following example.

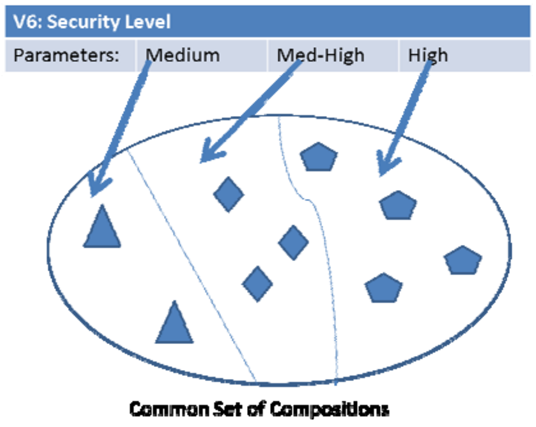

Figure 7. Indexing between parameters of variations and subsets of the Commonality Composition Subset

Figure 7 shows an example of the mapping relation for variability V6 created by this indexing process. The user-selectable parameter of Medium security level maps to the commonality composition subset shown with two triangles, each of which represents a candidate service composition. In order to find these triangle compositions, we first verify the security level to find those compositions with a Medium security level tag in all their services. Next, according to the dependency graph, a Medium security level requires an encryption length of 128 bits, so we verify whether all these compositions also have 128-bit encryption.

After creating the indexing for the parameters, the construction of the composition search space is completed and the application-engineering phase, i.e., the generation by the user of a service composition that verifiably satisfies his/her customized requirements, begins.

3.6 Solving Constraints

The user customizes the service composition's functional and non-functional requirements by setting the parameters of variation. However, since the user cannot be expected to handle the possibly complex dependencies among the variabilities, a specification from the user inputs may not guarantee consistency among the parameters. For example, an inconsistent specification is shown in Table 1. A Low parameter in the communication delay variability V7 constrains the choice of the Dispatch Service to be Speed-line rather than a normal one. In this case, the specification cannot result in a valid service composition.

Instead, in order to ensure the consistency of the user's choices, a dependency graph walking algorithm is used (here, to Fig. 3) to interact with and guide the user in solving the constraints. The constraint solving algorithm is as follows:

1. User picks a variability to start the process
2. User decides the parameter for this variability.
3. Locate the node of this variability in the dependency graph.
4. Check and apply the constraints on all the edges of current node. If the constraints force a variation value on any other node(s), mark them as explored.
5. Walk to the next unexplored node in the sub-graph and iterate from step 2.
6. If all nodes of the current sub-graph have been explored, pick a next sub-graph and start from step 2.
7. If all sub-graphs have been explored, then all constraints in the dependency graph have been solved.

Table 1: Excerpt of inconsistent specification

Variability	Value	Constraints
V1	Normal	If V1 is Highly-secured, V6 is High. If V1 is Speed-line, V7 is Low.
V7	Low	If V7 is Low, V1 is Speed-line

Without the product line concept of a dependency graph, the constraints among the different properties would have to be solved later during the properties' verification using higher-overhead verification techniques such as the one introduced in section 3.4. Solving the constraints using a dependency graph is more efficient than detecting the inconsistency in the later verification phase. Moreover, we will need this consistent specification to apply the query algorithm to the composition search space in the next step.

3.7 Query the Composition Search Space

We now query the composition search space to find the subset of compositions that satisfies each user-selected variation parameter. Because the construction of the search space in the previous step has verified that the remaining compositions satisfy the common requirements and that the user's selection of variabilities is consistent, we need only to perform a simple look-up in the Search Space Table. Table 2 continues our example, showing two queries on the parameters of V1 and V7.

Table 2: Sample query in the composition search space

Variability	Value	Composition Subset
V1	Speed-line	CSet1
V7	Low	CSet2

We note that since (V1=Speed-line) and (V7=Low) have already been shown to be consistent by the constraint solving in section 3.6, if any element of CSet1 satisfies V1=Speed-line and any element of CSet2 satisfies V7=Low, then any element of CSet1∧CSet2 satisfies (V1=Speed-line) and (V7=Low). More generally: *For any two properties P1 and P2, if P1 and P2 are consistent or independent, and any composition in SetA satisfies P1 and any composition in SetB satisfies P2, then any composition in SetA ∧ SetB satisfies both P1 and P2.*

3.8 User Customization

The user can customize the web service functionalities and its NFRs by setting the values for the variabilities. However, we need to make sure these values of the parameters are consistent with each other and do not violate any constraint identified in the CVA. To do this, we use a decision model. The structure of the decision model is shown in Figure 8. A decision model [4] is a user-friendly front-end model that consists of the commonalties, the variabilities, the parameters of variation together with their constraints/dependencies, the mapping relations and the algorithms to handle these complex relations. In both the product line and web-service context it interacts with the user and generates the user preferred product.

To customize the service composition, using prompts from the constraint solving algorithm in Sect. 3.6, the user successively selects values for the parameters of variation until all variabilities have been decided. The workflow for making decisions using the decision model is:

1. Apply constraint solving algorithm (3.6) to interact with the user.
2. Report failure if no valid consistent specification exists with the current user decisions and starts over.
3. If a consistent value set of the variabilities is obtained, query the search space with these parameters to retrieve a set of composition subsets.
4. Compute the conjunction of the composition subsets. If the conjunction yields a non-empty set, *any composition in this set will satisfy the user's common and variable (customized) FRs and NFRs.* If the conjunction is an empty set, it means no composition satisfies all the user's requirements.
5. Output the result. In the case of a non-empty set, we use the composition with the least services to avoid redundant services.

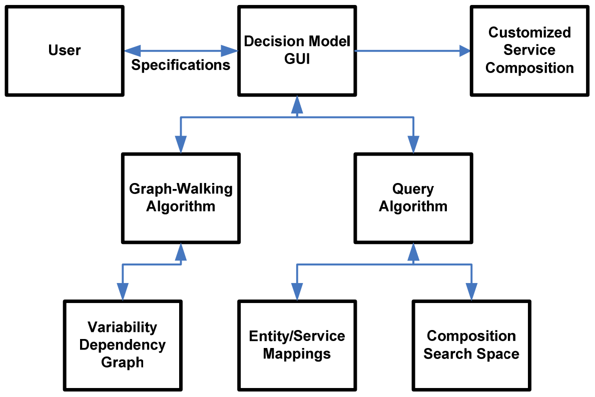

Figure 8. Structure of the decision model

4. Discussion

The approach in this paper introduced the FAST process for developing software product lines into the web-service composition domain. For customizable compositions of web services, the partitioning of the service composition into a domain engineering and an application engineering phase, as is done in product line development, allows us to first compose services that verifiably meet the common FRs and NFRs needed by all the customers of the service, and then to add on and similarly verify each customer's selected set of variations. The construction of the compositions as product-line assets supports their repeated reuse for commercial, mass-user services.

By incorporating product-line-like artifacts, this approach focuses on the reuse of the existing, verified compositions. It appears likely that this reuse will be able to reduce the time required to generate a verifiable user customization. To test this we plan to implement this conceptual approach on a mass-user service model. An experimental evaluation is needed to test this and to explore tradeoffs between the number of user customizations and number of possible variations for which it is advantageous to accept the overhead of creating the product-line artifacts (CVA, dependency graph, and decision model). Apart from improved efficiency, the product-line approach may also promote service quality. Measurement of historical user preferences can be used to predict future usage; measures of user satisfaction can help guide evolution of the services as new variations are introduced.

Future work is also needed to handle recoverable failures. That is, given a user's customized set of functional and non-functional requirements, our approach identifies a candidate set of compositions that satisfies them, if that is possible. While the decision model currently offers the user a single composition, it should also be possible to implement a strategy to remember the alternative compositions as "cold stand-bys". In the case of service failure, an alternative would then be recalled to replace the failed composition, so as to increase the reliability of the service.

5. Conclusion

This paper introduces a new approach to customizing the functional and non-functional requirements of a web service composition by incorporating software product line engineering techniques into the web service domain. By following the product line engineering procedures, the web service compositions in the search space of a commercial service provider can be more readily composed, verified, and reused in the presence of users' personal selections of their preferred variations. .

This approach creates a two-phase solution for efficiently handling mass-user service customization: a preparation phase in which the composition search space is constructed, and an implementation phase in which the web service composition is customized. We anticipate that the preparation phase will occur off-line and that the customization phase will occur at runtime. Most of the computational overhead of verification is here pulled into the preparation stage. The runtime verification for the user's customization requirements (the selection of variations) is thus simplified. A product-line decision model hides the background relations, the dependency models and the verification algorithms from the user. The decision model interacts with the user to maintain a consistent set of customized requirements and to generate a more convenient and efficient experience of service customization.

6. Acknowledgement

This research is supported by NSF grants 0541163 and 0702758.

7. References

[1] S. Dustdar and W. Schreiner, "A survey on web services composition", *Int'l Journal of Web and Grid Services*, Vol. 1, No.1, 2005, pp. 1–30.

[2] H. Sun, S. Basu, R. Lutz and V. Honavar, "Automata-Based Verification of Non-Functional Requirements in Web Service Composition", Dept. of Computer Science Technical Report, Iowa State University, 2009, submitted.

[3] D. M. Weiss and C. T. R. Lai, *Software Product Line Engineering*, Addison Wesley Longman, 1999

[4] D. M. Weiss, J.J. Li, H. Slye, T. Dinh-Trong and H. Sun, "Decision-Model-Based Code Generation for SPLE", *Proc. 12th International Software Product Line Conference (SPLC)*, 2008, pp. 129-138.

[5] T. Erl, *Service-oriented architecture: concepts, technology, and design*, Prentice Hall, 2005.

[6] L. Chung. B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Springer, 1999.

[7] J. Koehler and B. Srivastava, "Web service composition: Current solutions and open problems", *ICAPS Workshop on Planning for Web Services*, 2003, pp. 28-35.

[8] N. Milanovic and M. Malek, "Current Solutions for Web Service Composition", *Internet Computing*, Nov/Dec 2004, Vol. 8, No. 6, pp. 51-59.

[9] H. Foster, S. Uchitel, J. Magee and J. Kramer, "LTSA-WS: a tool for model-based verification of web service compositions and choreography", *28th Int'l Conf on Software Engineering (ICSE)*, Shanghai, China, 2006, pp. 771-774.

[10] H. Foster, S. Uchitel, J. Magee and J.Kramer, "Model-based Verification of Web Service Compositions", *18th IEEE Int'l Conf on Automated Software Engineering (ASE)*, Montreal, Canada, 2003, pp. 152-163.

[11] J. Pathak, S. Basu and V. Honavar, "Modeling Web Services by Iterative Reformulation of Functional and Non-Functional Requirements", *4th International Conference on Service Oriented Computing, Chicago*, USA, December 4-7, 2006, pp. 314-326.

[12] M. Karam, S. Dascalu, H. Safa, R. Santina and Z. Koteich, "A product-line architecture for web service-based visual composition of web applications", *Journal of Systems and Software,* Vol. 81. Issue 6, 2008, pp. 855-867.

[13] L. Balzerani, D. Di Ruscio, A. Pierantonio, G. De Angelis, "A product line architecture for web applications", *Proc. ACM Symposium on Applied Computing (SAC2005)*, pp. 1689–1693.

[14] K. Lee K. C. Kang, M. Kim and S. Park, "Combining feature-oriented analysis and aspect-oriented programming for product line asset development", *Proc. 10th International Software Product Line Conference (SPLC),* 2006, pp. 102-112.

[15] R. Capilla, and N.Yasemin Topaloglu, "Product lines for supporting the composition and evolution of service oriented applications", *Proc. 8th Int'l Workshop on Principles of Software Evolution*, pp. 53–56.

[16] J. Pathak, S. Basu, Robyn Lutz and V. Honavar, "Parallel Web Service Composition in MoSCoE: A Choreography-based Approach", *4th IEEE European Conference on Web Services*, 2006, pp. 3-12.

[17] B. Bruegge and A.H. Dutoit, *Object-oriented Software Engineering: Using UML, Patterns and Java*, Prentice Hall, 2003, pp. 181-196.