Abstract—Service Oriented Architecture (SOA) development practices typically lack a systematic framework for managing variability in service requirements and architectures. This paper addresses this gap by applying software product line (SPL) concepts to model SOA systems as service families. The approach is to model SOA variability with a multiple-view service model and a corresponding meta-model. We integrate SPL concepts of feature modeling and commonality/variability analysis with multiple service requirements and architectural views by using UML and the Service Oriented Architecture Modeling Language (SoaML). The paper describes an automated framework for service-oriented SPL engineering that allows modelers to design, deploy, and execute service-oriented SPLs.

Keywords—Software Product Lines, Service Oriented Architecture, Software Frameworks, Model-Driven Engineering, Variability Management, Service Families.

I. INTRODUCTION

Service Oriented Architecture (SOA) is an architectural style [1] for distributed computing that promotes flexible deployment and reuse. Services are consumed by many clients that have different requirements. Thus, clients that consume the same services usually exhibit varying requirements needs. Varying requirements usually necessitate varying software architectures. Therefore, both requirements and architectures have intrinsic variability characteristics.

Software Product Lines (SPL) are families of software applications that share common functionality, where each family member has variable functionality [4]. The SPL’s PLUS methodology [4] models variability in multiple views [1]. Since SOA systems vary in much the same as application families, i.e. they have common and variable features, this paper describes the use of SPL concepts to model the variability concerns of SOA systems.

Previous papers have described the multiple views of service oriented SPLs [2] and the underlying integrated multiple-view meta-model [3]. This paper describes an automated model-driven framework for the development of variable service-oriented product lines. The framework integrates feature meta-modeling [4], [5] with service views using UML and SoaML [6]. Such a framework facilitates variability modeling of service family architectures in a systematic and platform independent way. At the heart of the framework is a meta-model [3] that describes requirements and architectural meta-views of service oriented systems. The framework allows modelers to design, develop, execute, and deploy service-oriented software product lines.

The rest of the paper is structured as follows. Section II describes variability management challenges for service-oriented SPLs. Section III briefly describes our multiple-view modeling and meta-modeling approach that underpins our framework. Section IV describes our automated service-oriented SPL engineering framework. Section V describes how we validated our framework. Section VI describes related work, and section VII concludes the paper.

II. SERVICE-ORIENTED PRODUCT LINES

A. Modeling Variability in SOA

In SOA, service providers are usually decoupled from service requesters, thus requesters and providers vary independently of each other. Thus, variability modeling is necessary to manage the inherent complexity of service-oriented systems.

To manage variability in SOA, multiple-view modeling techniques can be applied. SOA systems can be modeled using multiple views such as contract/ business workflow requirements views, and service interface/ coordination architectural views. It would also be beneficial to have a framework that manages the aforementioned SOA variability concerns in a unified and platform-independent manner. However, current SOA variability management practices lack a systematic approach for managing variability and are typically platform-dependent. Furthermore, existing SOA variability management approaches [7], [8], [9], [10] do not address the multiple-view nature of variability in SOA in a unified and platform-independent manner.

B. Modeling Variability in SPLs

SPL development consists of two main processes [4]: a) Domain Engineering. A SPL multiple-view model, SPL architecture, and reusable components are developed and stored in the SPL reuse library. b) Application Engineering. The application developer selects the required features for the individual SPL member. Given the features, the SPL model and architecture are adapted and tailored to derive the application architecture.
C. Model Driven Engineering for SPLs

Our framework exploits the model driven engineering (MDE) principles to achieve the following goals:

- Development of model-driven techniques to design service-oriented SPLs.
- Automation of service-oriented product line engineering – this includes SPL Domain and Application Engineering.
- Development of a model-driven service-oriented prototype that realizes the aforementioned framework.

The paper details how traditional MDE concepts are adapted to cater for service-oriented SPL engineering. In MDE, developers create requirements and design models first without worrying about coding and platform concerns. These models are referred to as Platform-Independent Models (PIM) [14]. Once the PIMs are verified to match business requirements, they are transformed to Platform-Specific Models (PSM) [14], which are geared towards technology platforms. Finally, code is generated from the PSMs for a targeted technology platform such as Java, or .NET.

In typical MDE approaches [12], [13], [14] only one PIM is constructed for the entire application. Since SPLs are family of applications, this paper proposes the construction of two types of PIMs:

- A SPL PIM (splPIM) – this PIM models the entire service-oriented software family with all variability information. The SPL PIM is a multiple-view PIM since it is based on the multiple-view service variability model explained in our previous work [2].
- Software member application PIMs (memberPIM) – these PIMs model the derived member applications of the service-oriented SPL.

III. SERVICE VARIABILITY MODELING

In this section, we briefly describe our service variability model and meta-model, which were developed in our earlier work [2], [3]. The different SOA perspectives are grouped into multiple Requirements and Architectural views, which are formalized by multiple-view service variability meta-model. In addition, the feature model acts as a unifying view that provides the added dimension of modeling variability. The approach uses the SPL multi-view modeling approach advocated by the PLUS method [2], but defines three new views which are needed for service-oriented product lines, namely the service contract view and business process view for SOA requirements modeling, and the service coordination view for SOA design modeling.

A. Multiple View Service Variability Model

The multiple-view model consists of two Requirements views, the Service Contract Variability View and the Business Process Variability View, and two Architectural views, the Service Interface Variability View and the Service Coordination Variability View.

The Service Contract Variability View is a Requirements view that describes service contracts, which are prescribed by collaborating organizations in order to govern and regulate their interactions. Service contracts are modeled by SoaML’s ServiceContract element. This view also contains SoaML’s Participant elements that model providers or consumers of services. We categorize Service Contracts and Participants as kernel, optional, or alternative. Kernel elements are required by all members of an SPL, whereas optional elements are required by only some members. SPL members may select from mutually exclusive alternative elements.

The Business Process Variability View is a Requirements view that models the workflow of business processes. Participants may define internal business processes to conduct their business. Neither SoaML nor UML explicitly model business process workflow. Therefore, we define a Business Process Variability View that consists of one or more business processes, by using UML Activity diagrams. From a SPL perspective, each activity is categorized as kernel, optional, or alternative.

Services expose their capabilities through interfaces only. The Service Interface Variability View is an architectural view that models service interfaces by using SoaML’s ServiceInterface class. ServiceInterface specifies provided and required interfaces by Participants. A ServiceInterface is categorized as kernel, optional, or variant.

The Service Coordination Variability View is an architectural view that models the sequencing of service invocations. Services should be self-contained and loosely coupled in order to have a high degree of reuse; dependencies between services should therefore be kept to a minimum [5]. Hence, coordinators are used in situations where access to multiple services needs to be coordinated and/or sequenced. This view consists of coordinators which are modeled as objects with a <<Service Coordinator>> stereotype. Service coordinators are categorized as kernel, optional, or variant.

With the above service modeling views, it is possible to define the variability in each view and how it relates to other views. However, it is difficult to get a complete picture of the variability in the service architecture, because it is dispersed among the multiple views. The Feature View is a unifying view that focuses on service family variability and relates this variability to other service views. The Feature View is based on feature modeling [4] and is used to manage commonalities and differences among family members in a SPL. Features are analyzed and categorized as common, optional, or alternative. Related features can be grouped into feature groups, which constrain how features are used by a SPL member.

B. Multiple View Service Variability Meta-Model

The multiple-view variability modeling approach is based on a meta-model that precisely describes all views and views relationships. Each view in the multiple-view model is described by a corresponding meta-view in the meta-model (Fig. 1). There are two Requirements meta-views, Service Contract and Business Process, and two Architecture meta-views, Service Interface and Service Coordination. Our meta-modeling approach builds on
previous work in SPL multiple-view modeling and metamodeling [11].

SoaML's ServiceContract meta-class is used to model elements in the Service Contract meta-view (Fig. 1). To model variability, ServiceContract meta-classes are categorized as kernel, optional, or alternative. This meta-view also models contracts’ participants by using SoaML’s Participant meta-class. UML Activity meta-classes are used to model activities in the Business Process meta-view. Service interfaces are modeled by SoaML’s ServiceInterface meta-classes. The Service Coordination Meta-View consists of ServiceCoordinators, which are modeled by extending the UML Class meta-class. Messages sent and received by ServiceCoordinators are modeled as SoaML’s MessageType meta-classes.

To get a full understanding of the variability in service architectures, it is necessary to have one view that focuses entirely on variability and defines dependencies in this variability, which is the purpose of the Feature meta-modeling View (Fig. 1). Features are modeled by UML meta-classes that extend UML Class meta-class.

The meta-model also describes relationships between the aforementioned service meta-views. Object Constraint Language (OCL) [24] consistency checking rules are provided [3] for the relationships that cannot be explicitly described in the meta-model. The relationships in the meta-model consist of intra-view, inter-view, and feature to service variability views relationships [3]. The feature meta-view serves as the unifying meta-view that ties all the other meta-views together.

IV. SERVICE-ORIENTED SPL FRAMEWORK

To realize the model-driven service-oriented SPL engineering approach described in this paper, we developed a proof-of-concept prototype of an automated framework for model-driven Service-Oriented SPL Engineering (SoaSPL). The goals of the prototype are:

- Demonstrate the feasibility of the automated service-oriented SPL engineering framework.
- Ensure the consistency of multiple views of the multiple-view model.
- Model multiple-view service-oriented variability SPLs.
- Derive member service applications from the SPLs.
- Deploy, execute, and test member applications.

This section describes the design and implementation of the framework prototype utilizing the Eclipse Modeling Framework (EMF) [15]. EMF is an open-source meta-modeling framework and code generation facility for building tools and languages based on a structured metamodel called Ecore. It should be noted that EMF provides runtime support where instances of the meta-models, i.e. models, can be manipulated by Java code. Our framework utilizes EMF as a design environment. In other words, the Ecore meta-modeling language is used to create all model elements as first-class modeling elements. By representing the modeling elements of the service meta-views as first-class elements, high flexibility is gained, which facilitates precise modeling and consequently helps achieve the goals mentioned above.

A. SoaSPL Framework Development Process

The SPL member application derivation process is depicted in Fig. 2. Using SoaSPL, modelers and business analysts model the commonality and variability of an SPL domain by creating a feature model. Consequently, a multiple-view service variability model for the SPL, i.e.
An application developer selects features for the application and uses SoaSPLE to derive an SPL member application, i.e., memberSPL, utilizing our Java-based derivation rules. Each memberSPL is transformed into code utilizing SoaSPLE’s transformation facility, which executes the desired transformation rules, e.g., Java or .NET. During the modeling phases, SoaSPLE detects invalid modeling steps and inconsistencies and reports them to the modelers. Consequently, the modelers can take corrective actions to produce valid models.

B. Feature Meta-View to Service Meta-Views Mapping

This framework utilizes an Ecore meta-model that realizes the meta-model by establishing the mapping relationships among the Service Meta-Views and the Feature Meta-View. The mapping relationships were modeled as meta-class associations. In other words, the Feature meta-class from the Feature Meta-View has an association with each meta-class of the Service Meta-Views. In addition, the service views’ meta-classes have associations with each other. The end result is one Ecore meta-model that represents the multiple-view service variability meta-model [3] described in Section 3.2. In addition, this framework provides a Mapping Facility within the prototype that enables modelers to link features to service views. This Mapping Facility is governed by the multiple-view service variability meta-model.

C. Consistency Checking Rules

To ensure the consistency of the multiple views of service-oriented application families, the framework applies consistency checking rules [3] to the multiple-view service variability meta-model. The OCL [24]-based consistency checking rules were added to the multiple-view service variability meta-model as Ecore annotations [15]. SoaSPLE executes the consistency rules when service models, based on the multiple-view service variability meta-model, are created. If the service models violate these consistency checking rules, SoaSPLE emits popup messages indicating the violation. The following is an example consistency rule:

\[
\text{A Kernel ServiceContract can only support a kernel Feature}
\]
\[
\text{context Feature inv: reuseStereotype = ’kernel’ implies servicecontract->size() } \geq 1 \text{ and servicecontract.reuseStereotype = ’kernel’}
\]

D. Service Member Application Derivation

Member application derivation is based on feature selection from the feature model, e.g., Fig. 3. It should be noted that the derived model is an application multiple-view model that is based on the multiple-view service variability model and meta-model described above.

To automatically derive member applications, our framework defines a member application derivation algorithm that derives member application PIMs from the SPL PIM. The derivation algorithm is implemented in JAVA code that takes the SPL PIM and feature selection as inputs and produces member application PIMs. The algorithm de-selects SOA elements that are not mapped to any selected feature. Then, the algorithm traverses the selected features and constructs the derived member application by following the ‘supportedBy’ attributes of each selected feature. These attributes refer to SOA elements from the multiple service views. The output of the algorithm is a multiple-view service-oriented member application.

E. Code Generation

SoaSPLE is designed to use any PIM-to-PSM transformation rules [14] based on the desired target platform. In this paper, SoaSPLE uses Java/Web Services PIM-to-PSM transformation rules to derive the member PSMs. If a different target language is desired, say .NET instead of Java, SoaSPLE could use .NET PIM-to-PSM transformation rules and regenerate the member PSMs.
**F. Deployment and Execution**

SoaSPLE employs several technologies for deploying and executing service member applications:

- **Eclipse runtime support environment** [15].

- **Apache ODE** [20] – ODE is an open source BPEL engine. The generated BPEL code is compiled and deployed to ODE. The BPEL code invokes services based on WSDL files.

- **Apache CXF** [21] – CXF is an open-source web-services framework that supports standard APIs such as JAX-WS and JAX-RS as well as WS standards including SOAP, and WSDL.

- **Eclipse Swordfish** [22] – Swordfish is an open-source extensible Enterprise Service Bus (ESB).

**V. EVALUATION OF FRAMEWORK**

To evaluate the prototype framework, we developed an e-commerce service-oriented SPL case study using the SoaSPLE. The objective of the case study was to evaluate our framework with regard to the following properties:

1. The multiple views of the service-oriented SPL are consistent with each other.

2. The multiple-view service variability model is compliant with the underlying multiple-view service variability meta-model, i.e., each element in the multiple-view model is a valid instantiation of its corresponding meta-class in the meta-model and each association in the multiple-view model is a valid instantiation of its corresponding meta-association in the meta-model.

3. Derived software product line member applications are valid service-oriented SPL members. Hence, the evaluation validates that the derived member applications are valid members of the SPL family, and the derived SPL member applications are valid service-oriented applications, i.e. they execute within a service-oriented environment.

To achieve these objectives, the evaluation procedure was divided into two main testing tasks:

1. **Unit Testing** – tests each element and relationship in the multiple-view service variability meta-model. Unit testing is needed because the case study may not exercise every part of the meta-model. Unit testing validates the first and second properties above by demonstrating that each element in the multiple-view service variability meta-model can be instantiated as a modeling element. Unit testing also demonstrates that each association and consistency rule in the meta-model can be satisfied between elements of the instantiated model. Unit testing is accomplished via SoaSPLE and defined using Junit [16].

2. **System Testing** – tests the running service-oriented applications of the SPL end-to-end. System testing is needed to ensure that the automated framework can produce valid SPL and service-oriented applications. System testing validates the third property above.

The system testing approach validates the derived service member applications of an electronic commerce software product line (SPL). This type of testing ensures that each feature of the SPL’s feature model is tested by tracing the sequence of activities for each business process of member applications through testing the sequence of service coordinators and service invocations. The testing results are demonstrated by having an execution trace that shows that the activity model sequence, i.e. the business process, is executed by the service coordinators and service operation invocations.

The case study is conducted via the proof-of-concept prototype, SoaSPLE. The case study involves feature modeling of the SPL, multiple-view variability modeling of the SPL, consistency checking of the SPL’s multiple views, derivation of the service member applications of the SPL, and execution of member applications of the SPL.

**A. Case Study Problem Description**

This case study demonstrates the modeling of an E-Commerce web based SPL by using our prototype framework. The following subsections illustrate the steps taken to conduct the case study.
B. Feature View Modeling

Commonality and variability analysis of the E-Commerce SPL is performed in this step. The result of this step is a feature model, which represents the Feature View of the multiple-view service variability model [2]. The feature model was created in SoaSPLE as shown in Fig. 3. As the modeler was building the feature model, SoaSPLE automatically checked the model against the Feature Meta-View of the multiple-view service variability meta-model [3] by using the embedded consistency checking rules. Violations of these consistency checking rules were detected by SoaSPLE and the modeler had to take corrective actions to create a valid feature model.

C. Service Contract Variability View Modeling

This is a Requirements view that describes service contracts. Based on the feature model and knowledge of the E-Commerce domain, it was decided to model Service Contracts and Participants as shown in the Service Contract View of SoaSPLE in Fig. 4. The model in Fig. 4 consists of Purchasing, Inventory Ordering, Credit Checking, and Sales Tax Service Contract classes. In addition, the model contains Buyer, Seller, Bank, Rating Agency, Book Publisher, Electronic Supplier, and Tax Agency service Participants. Service Contracts and Participants are categorized as kernel, or optional. Kernel elements are required by all members of the E-Commerce SPL, whereas optional elements are required by only some members. Thus, the model describes how service Participants interact with each other through Service Contracts to realize an E-Commerce family of applications. Note that this model is based on the Service Contract Variability Meta-Model described in [3].

D. Business Process Variability View Modeling

This Requirement view models the workflow of the Order Fulfillment business process that is defined by the Seller Participant. This view is based on the Business Process Variability Meta-Model [3]. A UML Activity diagram is constructed with kernel, optional, and alternative activities.

E. Service Interface Variability View Modeling

Upon examining the activities of the Order Fulfillment business process and based on the E-Commerce domain knowledge, it was decided to have the following service interfaces: IOrdering, IInventory, IBooksOrdering, ICreditRating, ISalesTax, IElectronicsOrdering, and IPayment. It should be noted that these service interfaces are provided or required by the service participants that are modeled in the Service Contract View (Fig. 4).

F. Service Coordination Variability View Modeling

This Architectural view models service coordinators that coordinate business workflows [2]. Upon examining the Activities of the Order Fulfillment business process and their corresponding service interfaces, a ServiceCoordinator was designed in SoaSPLE. The Order Fulfillment ServiceCoordinator coordinates the Order Fulfillment business process.

G. Feature View to Service Views Mapping

Once the service variability views were constructed, mapping features to service views commenced. Mapping the feature view to the service variability views was done manually by using the Mapping Facility within SoaSPLE. Here, the modeler constructed a mapping by linking features from the feature view with service elements in the variable service views. It should be noted that this mapping is governed by the underlying multiple-view service variability meta-model [3].

H. Member Applications Derivation

At this phase of the case study, a complete multiple-view service-oriented E-Commerce SPL PIM is constructed. Therefore, service member applications of the product line family can be derived based on feature
selection from the feature view. Two service member applications were derived as follows:

1) Basic E-Commerce Application
To specify the first member application, feature selection from the feature model (Fig. 3) is performed. To derive the Basic E-Commerce application, the Member Application Derivation facility of SoaSPLE was invoked. Next, the Code Generation Facility of SoaSPLE was invoked to transform the derived Basic E-Commerce application to Java/Web Services code. Web Services Description Language (WSDL) was produced for the Service Interface View. Business Process Execution Language (BPEL) code was generated for the Service Coordination View. Once code was generated, the Basic E-Commerce member application was deployed and executed using SoaSPLE.

2) Enhanced E-Commerce Application
To specify the second member application, feature selection from the feature model is performed. To derive the Enhanced E-Commerce application, the Member Application Derivation facility was invoked. The derived multiple-view service-oriented Enhanced E-Commerce application features are depicted in Fig. 5. Code generation and deployment was performed in SoaSPLE as in the case of the basic member application described in the previous sub-subsection.

I. Discussion of Framework Evaluation
The modeler in the case study modeled the commonality and variability of an E-Commerce SPL by creating a feature model. Then, multiple-view service variability model was created. Afterwards, the modeler mapped the feature model to the multiple-view service model. Then, two member applications were derived from the SPL. Finally, service member applications were deployed and executed.

During the modeling phases, SoaSPLE detected invalid modeling steps and reported them to the modeler. Consequently, the modeler took corrective actions and produced valid models.

This case study validated our proposed framework by satisfying the three properties set forth in the beginning of section 5. The prototype (SoaSPLE) was used to create the E-Commerce SPL used in the case study and derive member applications from it. System testing was achieved by making sure that each feature of the feature view was exercised by tracing the sequence of activities of the Order Fulfillment business process through testing the sequence of the Service Coordinator and service invocations.

VI. RELATED WORK
This section discusses related work and examines them in light of our work.
Chang and Kim in [17] add variability analysis techniques to an existing service oriented analysis and design method (SOAD). Decision tables are used to record variability types in each phase of the SOAD process.
Capilla and Topaloglu [18] advocate an SPL engineering approach that has a specific phase for service composition in the SPL architecture. However, the authors do not tie service selection to the features required in the SPL.
In [10], the authors used the concept of features to solve variability problems for SOA. However, the authors’ approach assumes the availability of service implementation code, which is not the norm in most SOA scenarios.
Park et al. [19] suggest a feature-based reusable domain service development approach to create reusable domain services. However, the approach in [19] does not consider the relationships between features and services.
The authors in [23] developed a model-driven tool support for SoaML based on EMF. The tool addressed single application, not SPLs, and did not present a framework.

VII. CONCLUSIONS
This paper has introduced an automated framework for variability management of service-oriented software product lines. The framework is based on a multiple-view modeling and meta-modeling approach that addresses variability concerns of service oriented application families in a systematic, unified, and platform-independent manner.
The framework is the result of the integration of SPL concepts of feature modeling and commonality/variability analysis techniques with service views using UML and SoaML. Furthermore, the framework has embedded consistency checking rules [3] that ensure the consistency of the multiple views of service-oriented application families. The framework also consists of mappings between the Requirements views and the Architectural views of the multiple-view service variability meta-model. In addition, the framework has derivation rules to automatically derive service member applications of service application families.

A proof-of-concept prototype (SoaSPLE) was developed to realize the framework. SoaSPLE was used to model a service-oriented E-Commerce SPL, derive and execute its member applications, and to ensure the consistency of the multiple views of the E-Commerce SPL.

Validation of this research consisted of Unit Testing and System Testing. Unit testing checked each element and relationship in the multiple-view service variability meta-model. System Testing tested the derived service-oriented applications of the SPL end-to-end.

The main contributions of this research are the Model Driven Framework for Service Oriented SPLs and a Proof-of-Concept Prototype of the model driven framework.

Future work that stems from this research would extend the automated framework to include more modeling concepts from the SoaML, to enhance SoaSPLE’s user interface, and to add feature-based discovery of service-oriented SPLs.

REFERENCES