

VARIABILITY MODELING AND META-MODELING FOR
MODEL-DRIVEN SERVICE-ORIENTED ARCHITECTURES

by

Mohammad Ahmad Abu-Matar

A Dissertation

Submitted to the

Graduate Faculty

of

George Mason University

in Partial Fulfillment of

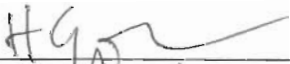
The Requirements for the Degree

of

Doctor of Philosophy

Information Technology

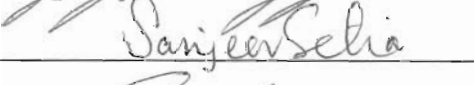
Committee:



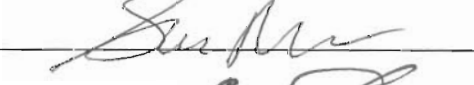
Dr. Hassan Gomaa, Dissertation Director



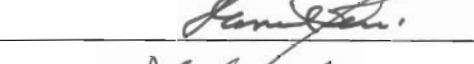
Dr. Jeff Offutt, Committee Member


Sanjeev Setia

Dr. Sanjeev Setia, Committee Member



Dr. Sam Malek, Committee Member


Daniel Menascé

Dr. Daniel Menascé, Senior Associate Dean



Dr. Lloyd J. Griffiths, Dean, Volgenau
School of Engineering

Date:

11/18/11

Fall Semester 2011

George Mason University

Fairfax, VA

Variability Modeling and Meta-Modeling for
Model-Driven Service-Oriented Architectures

A Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University

By

Mohammad Ahmad Abu-Matar
Master of Science
Regis University, 2004
Bachelor of Science
Wright State University, 1993

Director: Hassan Gomaa, Professor
Department of Computer Science

Fall Semester 2011
George Mason University
Fairfax, VA

Copyright: 2011 Mohammad Ahmad Abu-Matar
All Rights Reserved

DEDICATION

I dedicate this work to:

My dear Parents, Ahmad and Husnieh, for their limitless and selfless support and prayers.

My beloved Wife, Dima, for her patience, support, and for always being there for me.

My Children, Aya, Tuqa, Hamza, and Teeba, for their understanding and beautiful smiles.

ACKNOWLEDGEMENTS

All praise be to Allah (God) the Compassionate the Merciful. “O My Lord! Increase me in knowledge”, The Holy Quran 20:114.

I would like to thank my parents, Ahmad and Husnieh, for their continuous and unlimited support and prayers.

I would like to thank my wife, Dima, for her support, patience, and sacrifice.

I would like to thank my dissertation director, Dr. Hassan Gomaa, for his scholarly teaching, encouragement, and support.

I would like to thank my committee member, Dr. Jeff Offutt, for his pointed advice, teaching, and willingness to listen.

I would like to thank Dr. Daniel Menascé, Senior Associate Dean of the Volgenau School of Engineering, for his generous support, in the form of PhD Fellowships, and his scholarly direction.

I would like to thank the George Mason University Provost’s office for their generous support in the form of the PhD Completion Grant.

I would like to thank my friend and school mate, Dr. Ahmad El-Khodary for his good company, intellectual discussions, and help.

Last but not least, I would like to thank my longtime friends Amjad Al-Ramahi, Ayman Hajmoussa, Loai Alsalti, and Yaser Zaatreh for their selfless help and continuous advice.

TABLE OF CONTENTS

	Page
LIST OF TABLES.....	XI
LIST OF FIGURES	XII
ABSTRACT.....	XIV
1. INTRODUCTION.....	1
1.1. BACKGROUND.....	1
1.2. MOTIVATION	2
1.3. GLOSSARY OF RELEVANT TERMS	5
1.4. PROBLEM STATEMENT	6
1.5. RESEARCH STATEMENT.....	7
1.6. RESEARCH APPROACH	7
1.7. IMPORTANCE AND RATIONALE FOR THIS RESEARCH	9
1.8. CONTRIBUTIONS	9
1.9. ORGANIZATION.....	11
2. RELATED RESEARCH.....	12
2.1. SERVICE ORIENTED COMPUTING (SOC).....	12
2.2. SERVICE ORIENTED ARCHITECTURE (SOA)	13
2.3. SERVICE COMPOSITION	14
2.4. SOA VARIABILITY MODELING	15
2.4.1. SERVICE REQUIREMENTS VARIABILITY MODELING	15
2.4.2. SERVICE DESIGN VARIABILITY MODELING	17
2.4.3. VARIATION BY PARAMETERIZED SERVICES	18
2.4.4. VARIATION BY SERVICE COMPOSITION.....	20
2.5. SELF-ARCHITECTING SERVICE ORIENTED SYSTEMS	22

2.6.	SOFTWARE PRODUCT LINES	23
2.6.1.	PRODUCT LINE UML-BASED SOFTWARE ENGINEERING (PLUS).....	23
2.6.2.	FEATURE ORIENTED DOMAIN ANALYSIS (FODA).....	24
2.6.3.	REUSE-DRIVEN SOFTWARE ENGINEERING METHOD (RSEB).....	25
2.6.4.	FAMILY-ORIENTED ABSTRACTION, SPECIFICATION, AND TRANSLATION (FAST).....	26
2.6.5.	KOBRA.....	27
2.7.	FEATURE MODELING	27
2.8.	MULTIPLE-VIEW MODELING.....	31
2.9.	META-MODELING	32
2.10.	MODEL DRIVEN ARCHITECTURE (MDA).....	33
3.	RESEARCH APPROACH.....	35
3.1.	INTRODUCTION.....	35
3.2.	RESEARCH APPROACH	37
3.3.	RELATION TO EXISTING RESEARCH APPROACHES.....	38
3.4.	FEATURE MODELING AND META-MODELING.....	39
3.5.	MULTIPLE VIEW SERVICE VARIABILITY MODELING.....	39
3.6.	MULTIPLE VIEW SERVICE VARIABILITY META-MODELING	40
3.7.	CONSISTENCY CHECKING AND MAPPING RULES.....	40
3.8.	MODEL-DRIVEN SERVICE-ORIENTED PRODUCT LINE ENGINEERING FRAMEWORK.....	41
3.9.	PROOF-OF-CONCEPT TOOL PROTOTYPE	41
3.10.	APPROACH VALIDATION.....	41
4.	MULTIPLE-VIEW SERVICE VARIABILITY MODEL	43
4.1.	INTRODUCTION.....	43
4.2.	USING SPL CONCEPTS TO MODEL SOA VARIABILITY.....	44
4.3.	SERVICE CONTRACT VARIABILITY VIEW.....	46
4.4.	BUSINESS PROCESS VARIABILITY VIEW.....	47
4.5.	SERVICE INTERFACE VARIABILITY VIEW	48
4.6.	SERVICE COORDINATION VARIABILITY VIEW.....	49
4.7.	FEATURE VIEW	51

4.8.	MULTIPLE-VIEW SERVICE VARIABILITY MODEL RELATIONSHIPS..	52
4.8.1.	INTRA-VIEW RELATIONSHIPS	53
4.8.2.	INTER-VIEW RELATIONSHIPS	55
4.8.3.	FEATURE TO SERVICE CONTRACT VIEW RELATIONSHIPS	57
4.8.4.	FEATURE TO BUSINESS PROCESS VIEW RELATIONSHIPS	57
4.8.5.	FEATURE TO SERVICE INTERFACE VIEW RELATIONSHIPS.....	58
4.8.6.	FEATURE TO SERVICE COORDINATION VIEW RELATIONSHIPS ..	59
4.8.7.	FEATURE DEPENDENCY TO SERVICE VIEWS RELATIONSHIPS..	60
5.	MULTIPLE-VIEW SERVICE VARIABILITY META-MODELING.....	61
5.1.	INTRODUCTION.....	61
5.2.	SERVICE CONTRACT VARIABILITY META-VIEW	63
5.3.	BUSINESS PROCESS VARIABILITY META-VIEW	64
5.4.	SERVICE INTERFACE VARIABILITY META-VIEW	65
5.5.	SERVICE COORDINATION VARIABILITY META-VIEW	66
5.6.	FEATURE META-VIEW.....	66
5.7.	SERVICE VARIABILITY META-MODEL RELATIONSHIPS	67
5.8.	INTRA META-VIEW RELATIONSHIPS	68
5.9.	INTER META-VIEW RELATIONSHIPS	69
5.10.	FEATURE TO SERVICE CONTRACT META-VIEW RELATIONSHIPS.	70
5.11.	FEATURE TO BUSINESS PROCESS META-VIEW RELATIONSHIPS.	71
5.12.	FEATURE TO SERVICE INTERFACE META-VIEW RELATIONSHIPS.	72
5.13.	FEATURE TO SERVICE COORDINATION META-VIEW RELATIONSHIPS.....	72
6.	MODEL-DRIVEN SERVICE-ORIENTED PRODUCT LINE ENGINEERING FRAMEWORK	74
6.1.	INTRODUCTION.....	74
6.2.	PLATFORM INDEPENDENT MODEL (PIM)	75
6.3.	PLATFORM SPECIFIC MODEL (PSM)	79
6.4.	FEATURE BASED SERVICE APPLICATION DERIVATION	80
6.5.	SPLPIM TO MEMBERPIM DERIVATION ALGORITHM.....	81
6.6.	SERVICE-ORIENTED SPL ENGINEERING TOOL PROTOTYPE	82

6.7.	FEATURE META-VIEW	83
6.8.	SERVICE META-VIEWS	84
6.9.	FEATURE META-VIEW TO VARIABLE SERVICE META-VIEWS MAPPING	85
6.10.	CONSISTENCY CHECKING RULES	86
6.11.	SERVICE MEMBER APPLICATION DERIVATION	88
6.12.	CODE GENERATION	88
6.13.	DEPLOYMENT AND EXECUTION	89
7.	RESEARCH VALIDATION	90
7.1.	VALIDATION APPROACH	90
7.2.	SYSTEM TESTING APPROACH	92
7.3.	UNIT TESTING	93
7.4.	UNIT TEST CASES	93
7.4.1.	FEATURE META-VIEW UNIT TESTS	94
7.4.2.	SERVICE CONTRACT META-VIEW UNIT TESTS	96
7.4.3.	BUSINESS PROCESS META-VIEW UNIT TESTS	96
7.4.4.	SERVICE INTERFACE META-VIEW UNIT TESTS	97
7.4.5.	SERVICE COORDINATION META-VIEW UNIT TESTS	97
7.4.6.	INTER-VIEW RELATIONSHIPS UNIT TESTS	98
7.4.7.	CONSISTENCY CHECKING RULES UNIT TESTS	99
8.	ELECTRONIC COMMERCE SERVICE-ORIENTED SOFTWARE PRODUCT LINE CASE STUDY	100
8.1.	CASE STUDY OBJECTIVES	100
8.2.	CASE STUDY VALIDATION APPROACH	101
8.3.	ELECTRONIC-COMMERCE CASE STUDY PROBLEM DESCRIPTION 101	
8.4.	FEATURE VIEW MODELING	102
8.5.	SERVICE CONTRACT VARIABILITY VIEW MODELING	105
8.6.	BUSINESS PROCESS VARIABILITY VIEW MODELING	106
8.7.	SERVICE INTERFACE VARIABILITY VIEW MODELING	107
8.8.	SERVICE COORDINATION VARIABILITY VIEW MODELING	108
8.9.	FEATURE VIEW TO SERVICE VIEWS MAPPING	109

8.10.	MEMBER APPLICATIONS DERIVATION.....	112
8.10.1.	BASIC E-COMMERCE APPLICATION	112
8.10.2.	ENHANCED E-COMMERCE APPLICATION.....	115
8.11.	CASE STUDY CONCLUSION.....	117
9.	HOTEL RESERVATION SERVICE-ORIENTED SOFTWARE PRODUCT LINE CASE STUDY	118
9.1.	CASE STUDY OBJECTIVES.....	118
9.2.	CASE STUDY VALIDATION APPROACH	119
9.3.	HOTEL RESERVATIONS CASE STUDY PROBLEM DESCRIPTION..	119
9.4.	FEATURE VIEW MODELING.....	121
9.5.	SERVICE CONTRACT VARIABILITY VIEW MODELING	122
9.6.	BUSINESS PROCESS VARIABILITY VIEW MODELING	124
9.7.	SERVICE INTERFACE VARIABILITY VIEW MODELING.....	125
9.8.	SERVICE COORDINATION VARIABILITY VIEW MODELING	126
9.9.	FEATURE VIEW TO SERVICE VIEWS MAPPING	127
9.10.	MEMBER APPLICATIONS DERIVATION.....	129
9.10.1.	CONVENTIONAL ROOMS APPLICATION	129
9.10.2.	RESIDENTIAL ROOMS APPLICATION.....	131
9.11.	CASE STUDY CONCLUSION.....	132
10.	CONCLUSIONS	133
10.1.	INTRODUCTION.....	133
10.2.	RESEARCH CONTRIBUTIONS.....	134
10.2.1.	MULTIPLE-VIEW SERVICE VARIABILITY META-MODEL.....	134
10.2.2.	MULTIPLE-VIEW SERVICE VARIABILITY MODEL	136
10.2.3.	CONSISTENCY CHECKING AND MAPPING RULES	136
10.2.4.	MODEL DRIVEN FRAMEWORK FOR SERVICE ORIENTED SPLS	136
10.2.5.	SERVICE MEMBER APPLICATIONS DERIVATION RULES	137
10.2.6.	SOAML VARIABILITY NOTATION.....	137
10.2.7.	EXPLICIT MODELING OF SERVICE COORDINATION VARIABILITY 138	
10.2.8.	PROOF-OF-CONCEPT TOOL PROTOTYPE.....	138

10.3. FUTURE RESEARCH.....	139
10.3.1. SERVICE VARIABILITY MEDIATION LAYER	139
10.3.2. EVOLUTION OF SERVICE ORIENTED SPLS	140
10.3.3. FEATURE BASED DISCOVERY OF SERVICE ORIENTED SPLS ...	140
10.3.4. ENHANCEMENTS OF THE TOOL PROTOTYPE (SOASPLE)	140
BIBLIOGRAPHY	142
BIBLIOGRAPHY	143

LIST OF TABLES

Table	Page
Table 2.1 PLUS Feature Stereotypes	30
Table 7.1 Feature Meta-View Unit Tests.....	95
Table 7.2 Service Contract Meta-View Unit Tests.....	96
Table 7.3 Business Process Meta-View Unit Tests	96
Table 7.4 Service Interface Meta-View Unit Tests	97
Table 7.5 Service Coordination Meta-View Unit Tests.....	97
Table 7.6 Inter-View Relationships Unit Tests.....	98
Table 7.7 Consistency Checking Rules Unit Tests.....	99
Table 8.1 Feature to SOA Mapping.....	111

LIST OF FIGURES

Figure	Page
Fig. 4.1 Service Contract View of an E-Commerce SPL	46
Fig. 4.2 Business Process View of an E-Commerce SPL	48
Fig. 4.3 Service Interface View of an E-Commerce SPL	49
Fig. 4.4 Service Coordination View of an E-Commerce SPL	50
Fig. 4.5 Feature View of an E-Commerce SPL	52
Fig. 4.6 E-Commerce SPL Multiple-View Service Variability Model	54
Fig. 5.1 Service Contract Variability Meta-View	63
Fig. 5.2 Business Process Meta-View	64
Fig. 5.3 Service Interface Variability Meta-View	65
Fig. 5.4 Service Coordination Variability Meta-View.....	66
Fig. 5.5 Feature Meta-View	67
Fig. 5.6 Service Variability Meta-Model	68
Fig. 6.1 E-Commerce SPL Service Oriented Multiple-View Platform Independent Model	77
Fig. 6.2 Basic E-Commerce SPL Member Application.....	78
Fig. 6.3 Two Types of PIMs in SoaSPLE	79
Fig. 6.4 splPIM to memberPIM Derivation Process in SoaSPLE	81
Fig. 6.5 Feature View Meta-Model in SoaSPLE	84
Fig. 6.6 Service Contract View Meta-Model in SoaSPLE	85
Fig. 6.7 Multiple-View Service Variability View Meta-Model in EMF	86
Fig. 6.8 Meta-Class OCL Annotation Example.....	87
Fig. 8.1 Feature View of the E-Commerce SPL	104
Fig. 8.2 Service Contract View of the E-Commerce SPL	106
Fig. 8.3 Business Process View of the E-Commerce SPL.....	107
Fig. 8.4 Service Interface View of the E-Commerce SPL.....	108
Fig. 8.5 Service Coordination View of the E-Commerce SPL	109
Fig. 8.6 E-Commerce splPIM	110
Fig. 8.7 Basic Member E-Commerce Feature Model	114
Fig. 8.8 Enhanced Member E-Commerce Feature Model	116
Fig. 9.1 Feature View of the Hotel Reservation SPL	122
Fig. 9.2 Service Contract View of the Hotel Reservation SPL.....	123
Fig. 9.3 Business Process View of the Hotel Reservation SPL	124
Fig. 9.4 Service Interface View of the Hotel Reservation SPL	125
Fig. 9.5 Service Coordination View of the Hotel Reservation SPL	127
Fig. 9.6 Hotel Reservation splPIM.	128
Fig. 9.7 Conventional Rooms Feature Model.....	130

Fig. 9.8 Residential Rooms Feature Model.	131
--	-----

ABSTRACT

VARIABILITY MODELING AND META-MODELING FOR MODEL-DRIVEN SERVICE-ORIENTED ARCHITECTURES

Mohammad Ahmad Abu-Matar, PhD

George Mason University, 2011

Dissertation Director: Dr. Hassan Gomaa

Service Oriented Architecture (SOA) has emerged as an architectural style for distributed computing that promotes flexible application development and reuse. One of the major benefits claimed for SOA is the flexible building of IT solutions that can react to changing business requirements quickly and economically. Services could be consumed by many applications that have different requirements. In addition, applications usually change by adding new requirements, removing existing requirements, or updating existing requirements. Thus, applications that consume the same service usually exhibit *varying* requirements. Varying requirements usually necessitate varying software architectures that satisfy the varying requirements of software applications. Thus, both requirements and architectures have intrinsic variability characteristics.

SOA development practices currently lack a systematic approach for managing variability in service requirements and architectures. This dissertation addresses this gap

by applying software product line (SPL) concepts to model SOA systems as *service families*. The dissertation introduces an approach to model SOA variability with a multiple-view service variability model and a corresponding meta-model. The approach integrates 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). At the heart of this research is a multiple-view meta-model that defines the relationships among variable service views and maps features to variable service models along with a corresponding consistency checking rules that ensure the consistency of the multiple service views as they change. The dissertation describes how to derive family member applications from the SPL and presents a validation of the approach. This dissertation makes the case that the presented multiple-view service variability modeling and meta-modeling approach facilitates variability modeling of service families in a systematic and platform independent way. The key contributions of this research include: Multiple-View Service Variability Meta-Model, Multiple-View Service Variability Model, Consistency Checking and Mapping Rules, Model Driven Framework for Service Oriented SPL Engineering, Service Member Applications Derivation Rules, Explicit Modeling of Service Coordination Variability, and a Proof-of-Concept Tool Prototype.

1. Introduction

1.1. Background

Service Oriented Architecture (SOA) has emerged as an architectural style [1] for distributed computing that promotes flexible deployment and reuse. One of the major benefits claimed for SOA is the flexible building of IT solutions that can react to changing business requirements quickly and economically. The service-oriented architectural style consists of service providers that register their services, and of service requesters that search and discover these services based on their business needs.

Services could be consumed by many applications that have different requirements. In addition, applications usually change by adding new requirements, removing existing requirements, or updating existing requirements. Thus, applications that consume the same services usually exhibit *varying* requirements needs.

Varying requirements usually necessitate varying software architectures. In other words, when applications' requirements are changed, the software architectures of these applications are modified to satisfy the changed requirements. Thus, software architectures usually *vary* to implement the varying requirements of software applications. Therefore, both requirements and architectures have intrinsic *variability* characteristics.

Software architectures describe application designs from different perspectives [1]. In other words, the same application architecture consists of multiple depictions of different perspectives, also called *views* that address specific architectural concerns. For example, software architectures could describe conceptual, logical, and physical views of application design [2].

1.2. Motivation

In SOA, service providers are usually decoupled from service requesters, thus requesters and providers *vary* independently of each other. This variation manifests itself in several ways, i.e. in changing requirements, changing architectures, and changing execution environments. Requirements change because both clients and service providers will always add, improve, all remove features to/from their applications. Architectures change because in SOA, the architecture is not fixed, because the main elements of the architecture are services usually provided by external providers. Furthermore, execution environments vary because of the available variations in operating systems, middleware environments, and programming languages. Thus, variability modeling is necessary to manage the inherent complexity of service-oriented systems.

Variability also manifests itself in the execution environments, aka platforms, of service-oriented systems as well. Service providers can have different platforms, i.e. operating systems, middleware, and programming languages, than service consumers. In addition, service consumers and providers can switch to different platforms for reasons such as

better performance and high availability. Again, managing the change in platforms for multiple-view service-oriented systems can quickly become unwieldy.

It is hard to model complex and reasonably sized software systems from one perspective, e.g., structural. To manage this complexity, the software engineering community has used multiple-view modeling (Chapter 2) to model software systems from different perspectives. In essence, the same application can be modeled from different perspectives, where each perspective models a specific concern, e.g. requirements, architectures, and physical environments. Multiple-view modeling techniques can be applied to SOA modeling for the same reasons mentioned above. SOA systems can be segregated into multiple views such as contract/ business workflow requirements views, and service interface/ coordination architectural views. It should be noted, that these views depend on each other. In other words, a change in one view could necessitate a change in a different view.

For example, a service-oriented E-Commerce system could have a view that describes service contracts and service providers and consumers. Another view could be a business process view that describes the workflow of order fulfillment. Yet another view could be a service interface view that describes an ordering service's operations and parameters. If a task changes in the business process view, say a task is added to allow electronic check payments, a new service interface for electronic check payment has to be added to the service interface view. Likewise, if a credit check contract is added in the service contract view, a credit check service provider gets added to provide this capability. For reasonably

sized applications, changes in the interdependent views of service-oriented systems can quickly become unwieldy and difficult to manage.

It becomes evident from the aforementioned discussion, that requirements and architectural variability concerns are *dispersed* among the multiple views of SOA systems. To have a full picture of variability in SOA based systems, it would be necessary to have *one* view that only describes variability in the entire system. In addition, there is a need to model variability of SOA systems in all views in a consistent manner. Furthermore, consistency of all related elements in the multiple views should be ensured as these views are modeled.

Software Product Lines (SPL) and Commonality/Variability Modeling (Chapter 2) model the variability of application families. Application families share common features, but each differ in some unique way. The SPL's PLUS methodology [3] models variability in multiple views and has distinct modeling treatment for the different phases of the software development life cycle of application families. Since SOA systems vary in such a way that is similar to application families, i.e. they have common and variable features; the research in this dissertation proposes the use of SPL concepts to model the variability concerns of SOA systems.

It would be beneficent to have a framework that manages the aforementioned SOA variability concerns in a *unified* and *platform-independent* manner. However, current SOA variability management practices (Chapter 2) lack a systematic approach for managing variability and are typically platform-dependent. Furthermore, existing SOA

variability management approaches [4], [5], [6], [7], [8], do not address the multiple-view nature of variability in SOA in a *unified* and *platform-independent* manner.

1.3. Glossary of Relevant Terms

Some relevant terms that could have varying definitions in different disciplines are defined in this section. The goal is to establish a consistent level of understanding for terms used throughout this dissertation.

- **Consistency Checking Rules** – rules that are determined from the proposed multiple-view meta-model, which ensure the consistency of the interdependent views of SOA systems. (Chapter 5)
- **Feature Modeling** – a modeling practice that model common and variable requirements for an application family (Chapter 2)
- **Model Driven Architecture (MDA)** –an Object Management Group (OMG) initiative that promotes development practices where models are used as first class entities. Software development is driven by constructing models in all phases of the development life cycle.
- **Multiple-View Modeling** – a technique for describing the architecture of software-intensive systems via the use of multiple or perspectives, where each perspective, i.e. view, addresses specific set of concerns. Examples of multiple views are: logical, physical, and development.

- **Platform Independent Model (PIM)** – within MDA, these models capture business logic details independent of programming languages, operating systems, or middleware environments.
- **Platform Specific Model (PSM)** – within MDA, these models represent technology specific concerns such as software languages and middleware environments. PIMs are transformed into PSMs using predefined transformation rules. Eventually, PSMs are transformed into code.
- **Software Product Lines (SPL)** – are families of software systems that share common functionality, where each member has variable functionality. The main goal of SPL is the rapid development of member systems by using reusable assets from all phases of the development life cycle. (Chapter 2)
- **Service Oriented Software Product Line Engineering (SoaSPLE)** – the name of the framework and tool prototype proposed in this research. (Chapter 6).
- **Variability Modeling** – in SPL, this practice models changes in all SDLC phases and all views of application families (Chapter 2).
- **Unified View** – a view that captures a specific concern that is dispersed among the multiple views of a system. The Feature View, in this research, captures variability concerns which are dispersed among the multiple views of SOA based systems. (Chapters 4 and 5).

1.4. Problem Statement

Existing SOA variability management approaches do not provide a systematic way to address variability concerns in a multiple-view, unified, consistent, and platform-independent manner. Since SOA is multi-view in nature, it is necessary to have a variability management approach that addresses variability concerns in SOA systematically to ensure the consistency and correctness of service-oriented systems.

1.5. Research Statement

An approach and automated framework can be devised to model, develop, and execute variable service oriented systems in a multiple-view, consistent, and platform-independent manner by using Software Product Lines (SPL) and Feature Modeling principles.

Since services in SOA can be used by different clients with varying functionality, SOA variability modeling would benefit from SPL variability modeling techniques. Applying SPL concepts, service oriented systems can be modeled as service application families.

1.6. Research Approach

This research addresses the lack of systematic approaches to handling variability management concerns in SOA by developing a multiple-view variability modeling and meta-modeling approach. Furthermore, this research addresses SOA variability management concerns in a systematic and platform-independent way. The research approach is summarized as follows:

1. Develop a multiple-view service variability model, and define the relationships between the service views, since variability in SOA systems is too complex to be modeled in a single view. This multiple-view model uses Feature Modeling to construct a unifying view that describes variability in SOA which is dispersed in multiple views.
2. Develop a multiple-view service variability meta-model that formalizes the aforementioned multiple-view service variability model. A meta-model is needed to serve as the underlying representation of the automated framework that is proposed in this research.
3. Develop rules for consistency checking and mapping between the multiple views in the meta-model to ensure that these views are consistent with each other as SOA based systems change.
4. Develop service family member applications derivation rules; since the approach integrates SPL concepts of application families with multiple-view modeling to capture variability in SOA systems.
5. Automate the research approach by designing a model-driven framework, which is implemented in a proof-of-concept prototype.
6. Validate this research by applying the multiple-view modeling approach and proof-of-concept prototype to case studies. The case studies are E-Commerce service-oriented SPL, and Hotel Reservations service-oriented SPL.

The research approach is described in more detail in Chapter 3.

1.7. Importance and Rationale for this Research

The ability of distributed computing systems to respond to change in an effective manner is essential. SOA emphasizes an architectural style that enables software engineers to design and implement loosely-coupled solutions that are amenable to change. However, existing service-oriented variability design, development, and management approaches are largely ad-hoc and do not ensure the consistency of multiple views of service-oriented systems. In addition, existing approaches are largely platform-dependent, which impedes their adoption in multiple technology platforms.

The multiple-view service variability modeling and meta-modeling approach presented in this research allows the development of service-oriented application families, by using the concepts of SPL and Feature Modeling, in a consistent and platform-independent manner, thus enabling these application families to change in an effective and unified manner.

It should be noted that parts of this dissertation have been published in refereed conferences and workshops, in particular references [9], [10], [11], [12].

1.8. Contributions

This section summarizes the contribution of this research. A detailed contribution is described in Chapter 10.

- Multiple-View Service Variability Meta-Model – this meta-model governs the multiple-view service variability model and serves as the basis of the automated of the SoaSPL framework.
- Multiple-View Service Variability Model – this model provides notation for capturing the variable multiple views of SOA systems.
- Consistency Checking and Mapping Rules – these rules ensure that the variable multiple views of SOA systems are consistent with each other as these systems change.
- Model Driven Framework for Service Oriented SPLs – this is an automated framework that helps developers and modelers to systematically build variable service-oriented SPLs.
- Service Member Applications Derivation Rules – these rules act upon multiple-view service-oriented SPL models and derive single applications of the SPL.
- SoaML Variability Notation – a notation to use the Service Oriented Architecture Modeling Language for modeling service-oriented SPLs.
- Explicit Modeling of Service Coordination Variability – service coordination was modeled down to the architectures of service coordinators where individual coordination messages were tied to the variability of business processes.

- Tool Prototype – a proof-of-concept tool prototype was developed to realize and help in validating the proposed automated framework. The prototype was built by using current SOA open-source modeling and execution technologies.

1.9. Organization

This dissertation is organized as follows. Chapter 2 surveys related research. Chapter 3 details the research approach. Chapter 4 describes a multiple-view service variability modeling approach and views relationships. Chapter 5 describes a multiple-view service variability meta-modeling approach. Chapter 6 presents a model-driven framework for service-oriented SPL engineering along with a tool support environment. Chapter 7 presents the validation approach of this research, Chapter 8 details an E-Commerce case study that validated the research. Chapter 9 details a Hotel Reservation System case study that also validated the research. Finally, Chapter 10 concludes the dissertation, outlines the contributions of this research, and suggests future work.

2. Related Research

This chapter presents a literature review that spans areas related to this research: service oriented computing and architecture, service modeling, service oriented variability techniques, software product lines (SPL), feature modeling, SPL based SOA variability techniques, multiple-view modeling, meta-modeling, and model driven architecture (MDA).

2.1. Service Oriented Computing (SOC)

Service Oriented Computing (SOC) is an emerging computing paradigm that evolved from object orientation, client/server systems, and component-based computing. Unlike traditional OO development practices where developers receive requirements and then design and build applications, SOC divides development activities into three groups: application builders (or service requesters), service brokers (or mediators), and service developers (or providers) [13].

Application development is ideally accomplished via discovery of desired services and service assembly rather than coding. Service providers publish their services on registries where consumers can find and use them. Current Web services standards support the aforementioned architecture, but do not necessarily adhere to the basic principles of Service Oriented Computing (SOC). [14]

The following are some SOC principles [15]: loose coupling, service contract, autonomy, abstraction, reusability, composability, statelessness, and discoverability.

2.2. Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is an emerging standard-based architectural style for designing, building, and deploying flexible distributed software applications. SOA emphasizes extremely loosely coupled design approaches where disparate systems, with different computing platforms, can collaborate and evolve without major changes to their existing core architectures. Services are designed as self-contained modules that can be advertised, discovered, composed, and negotiated on demand.

Although SOA relies on existing software architecture practices like information hiding and separation of concerns, it adds new ones like service composition, service choreography, and service repositories. To realize a SOA, a process to identify and design services has to be established much like the traditional processes of OOAD [16]. Several authors argue that to model business related services, the level of abstraction has to be raised up closer to the business domain [16], [17]. Despite the wide academic and industrial activities related to SOA, there is no systematic end-to-end methodology for analyzing and designing service-oriented applications. There is a near unanimous agreement [16], [17], [7], [19] that such a methodology is needed to elevate service-oriented computing to a mainstream computing level. As a result, Service Oriented Analysis and Design (SOAD) is emerging as a new field which is concerned with

identifying and building services based on business requirements. SOAD aims at treating services as first class entities much like OOAD treated classes and objects.

Several UML meta-models [20] were developed to organize the SOAD activities. SOA is based on a multi-level architecture that blends business process models, Enterprise Architecture, Object Orientation, and Service Orientation [16]. The benefits of SOA [15] can be realized when applied across multiple solution environments where processing is highly distributed and each service has an explicit functional boundary and resource requirements.

It should be noted that Web Services technology is the most prominent implementation platform for SOA, however Web Services are only one manifestation of SOA and they do not necessarily fully adhere to SOC principles [15]. Because of the current reliance on Web Services, SOA requires the establishment of XML data representation architecture [15].

2.3. Service Composition

Service composition, i.e. assembling services to build applications, is a central theme in SOA because it emphasizes reuse at a higher level. Service composition is a fundamental activity in any meaningful SOC environment [21]. Reuse is realized by assembling other business partners' services rather than reusing individual components.

The main challenges to composition stem from the changing nature of users' requirements and the open environments that services reside in [22].

Any service composition approach must satisfy the following requirements [21]:

Connectivity, Nonfunctional QoS properties, Composition correctness, and Services scalability.

It should be noted that service composition is not standardized and several approaches exist to handle it.

2.4. SOA Variability Modeling

In this section, SOA variability modeling techniques in the requirements and architecture phases are investigated.

2.4.1. Service Requirements Variability Modeling

Business Process Models (BPM) are the most widely used requirements artifacts in SOA.

In this section, variability modeling in UML activity diagrams is investigated since activity diagrams are the most prominent graphical notations for representing BPM [23].

Activity diagrams are suitable for representing BPM for several reasons: the popularity of the UML, precise semantics, and the rich structure of class modeling in UML 2.0 [23].

There are many ways to represent variability in activity diagrams [24]. Variability could be modeled as an encapsulation of variant sub-processes [24]. This mechanism is implemented simply by invoking the UML's CallBehaviorAction meta-class on the desired varying activity. Thus, variability is achieved by encapsulating different sub-activities.

Variability can also be achieved by addition, replacement, and omission of encapsulated sub-processes [24]. These mechanisms can be implemented by UML's

CallBehaviorAction meta-classes as well. CallBehaviorAction meta-class can be added at any location in the activity diagrams; however, the number of output parameters of the new CallBehaviorAction meta-class must be equal to the number of ObjectFlowEdge meta-classes [25] interrupted by the addition. Replacements and Omissions can be implemented similarly with the careful consideration of the consistencies between input and output parameters.

Extension points can also be used as a variability mechanism by having the CallBehaviorAction meta-class call Activities that contain extending sub-processes at pre-designated locations [24]. In addition, Delegation can be achieved by calling external activities using the SendSignalAction [25] meta-class.

Parameterization is another popular variation mechanism which can be implemented in several ways in UML Activity diagrams. The main goal of parameterization in activity diagrams is the execution of variant Actions in the process flow. This can be achieved by the utilization of DecisionNode, JoinNode, and guard expressions attached to ActivityEdge meta-classes [25]. Actions themselves can be parameterized using ValuePins . The data flow between actions can be parameterized using ParameterSet meta-classes [25].

Data type variability can be implemented by using UML's Activity ObjectNode meta-class [25] that corresponds to variant types.

After detailing the different variability mechanisms in activity diagrams [24], several annotations were presented in the form of stereotypes that can be used in the variant activity diagrams.

An SPL testing approach is described [26] where variation points in use cases are converted into activity nodes in activity diagrams with variation point parameters.

Variant activity nodes are annotated with the <<adaptable>> stereotype. Further, feature conditions are expressed in guard expressions with the 'fc' prefix [fc: condition]. It should be noted that this approach is similar to the approach presented above [24] but on a smaller scale and without the use of the activity diagrams meta-model. The SPL testing approach [26] uses one stereotype, <<adaptable>>, and relies heavily on using guard expressions to model feature conditions.

The stereotypes <<optional>> and <<alternative>> [27] are used to denote variant activity nodes in Activity diagrams. The same stereotypes are also applied to partitions, or Activity Swim Lanes, to highlight variant sequences instead of individual nodes. Annotating entire sequences of nodes provides a richer and more explicit mechanism to model variability in activity diagrams.

2.4.2. Service Design Variability Modeling

Services can be used in several systems through customization [5]. The basic, or generic, service can be invoked by several systems with varying functionalities. To be used by multiple consumers, services need to be designed with variability in mind.

A variability analysis technique [28] is introduced into an existing service oriented analysis and design method (SOAD). A four layered SOA architecture is adopted that consists of: Business Process Layer, Unit Service Layer, Service Interface Layer, and Service Component Layer. In addition, each layer is associated with a variability type: Workflow (Business Process Layer), Composition (Unit Service Layer), Logic (Service Component Layer), and Interface Mismatch (Service Interface Layer). Decision tables are used to record variability types in each phase of the SOAD process. Decision tables are used to map business process requirements to unit services and service components. This approach emphasizes the process of creating adaptable services, but it does not provide modeling nor automation methods.

Architectural patterns [5] are used to model variation points in Web Services. It is argued that Web Services standards have inherent support for variability. For example, clients can communicate with services implemented in different technologies like Java or .NET and several protocols can be used to transfer messages between clients and services. In addition, Web Services Definition Language (WSDL), and the Universal Description, Discovery and Integration (UDDI) attributes and parameters can all be customized and modeled as variation points.

2.4.3. Variation by Parameterized Services

In this approach, services are adapted for variation by introducing parameterized variation points within the services. The following are some examples of service parameterization:

- Operations and Parameters Variability

Operations and their parameters can be parameterized at the Interface Definition level for services. (WSDL) documents can be exploited to implement variability when Web Services are used as an implementation technology. In fact, each element in a WSDL document could be considered as a candidate for a variation point [5]. In addition, elements addition and operations reordering in WSDL do not break backward compatibility [29]. Thus, evolving services' operations based on product line members' features is possible without introducing separate WSDL documents.

- Transport Variability

Variation points can be realized in the transport choice for a service. For example, a software product line member can select SOAP as a transport while another member selects FTP [5]. Again, this transport choice could be parameterized within one WSDL document if Web Services are used for implementation.

- Endpoint Variability

Variation points can be realized in the location choice for a service. For example, some product line members may specify URIs, while others only specify one location. Multiple endpoints may be needed for fault tolerance or data replication. Endpoints specification could be parameterized within WSDL as well.

- Discoverability and Binding Variability

Variation points can be realized in the way the product line members are discovered and bound to. Services are advertised in public registries by exposing certain characteristics. Software product line members can parameterize these

advertised characteristics to control how clients discover and bind to them. UDDI discovery technology can be used for this purpose as shown in [5]. On the other hand, some members may choose not to be discovered at all for security and other concerns. In these situations, the UDDI discovery mechanism could be parameterized to offline a product line member based on some features' conditions. Off-lining a product line member in the registry does not render it unusable, since its endpoint can be fed to selected clients statically.

- Error Handling Variability

Variation points can be realized in the way product line members process their error handling [5]. Error handling can be parameterized based on feature conditions that dictate the actions taken in certain events. For example, some product line members may chose to simply log error messages while others may attempt some predetermined corrective actions.

Several design patterns like Strategy and Decorator can be used to realize the identified variation points within services [5]. The Strategy pattern can be used to realize operations' variability, whereas the Decorator pattern can be used to realize parameters' variability.

2.4.4. Variation by Service Composition

When grouping related features into services, variations can be realized by composing feature-based services. A lightweight product line engineering approach [6] is proposed with a specific phase for supporting service composition in the product line architecture.

Several variation points are introduced that can be used to customize the product line during service selection in the application engineering phase:

- ***OrchestrationType*** – specifies the activities that constitute service invocations.
- ***DataModel*** – defines data exchanged between services
- ***ServiceSelection*** – specifies the types of bindings (static and dynamic) between services in a composition.
- ***TypeOfException*** – is used to specify how a specific exception is handled.
- ***QualityFactor*** – specifies quality attributes, like cost and performance, needed by different services.

However, service selection is not tied to feature selection of the product line and composition verification is not explained.

Several design patterns like Composite, Iterator, and Chain of Responsibility can be used to realize composition variation between services [5]. The Composite pattern can be used to implement the composite service that encompasses other sub-services, whereas the Iterator pattern can be used to implement the traversal of sub-services. In addition, the Broker architectural pattern [3] could be used in designing the architecture for such a composition.

A variability modeling language (OVM) is proposed [30] to model service composition choices in business process definitions. A meta-model is introduced that explains the relations among business process activities, variation points, and services. By modeling variation points and their variants, service composition is determined based on users'

selection of available services that meet their goals. Thus, a user centric approach is proposed where the selection of variants dictates service composition.

A Web Services based software product line engineering approach [4] is proposed where components in the architecture were modeled using the <<web service>> stereotype.

Feature selection from feature models determined the selection of Web Services. UML activity diagrams modeled customization choices based on feature/Web Service interactions. However, the selection of Web Services that support feature selection was predetermined and no discovery or composition of services was modeled in the architecture.

2.5. Self-Architecting Service Oriented Systems

Some systems have intrinsic variability in both requirements and architectures, e.g. emergency response systems. These types of systems can benefit from SOA principles to facilitate their efficient construction and evolution.

A self-architecting SOA framework is proposed that allows domain experts to model business processes and an SOA based architecture to be automatically generated [31].

Business processes are modeled by using an activity based modeling language [32] similar to BPMN. Changing requirements trigger the regeneration of software architecture. Qualities of service (QoS) requirements are modeled by using a scenario-based modeling language to which QoS objectives are associated.

To satisfy QoS requirements, an SOA is generated that optimizes utility function for the entire system [33].

2.6. Software Product Lines

Software Product Lines (SPL) are families of software systems that share common functionality, where each member has variable functionality [34], [35], [3]. The main goal of SPL is the rapid development of member systems by using reusable assets from all phases of the development life cycle.

In the SPL domain engineering phase, requirements, analysis, and architecture of the application family is developed upfront. Then, in the application engineering phase, the derivation of member applications is carried out by tailoring the architecture based on the unique member application's features.

Several SPL engineering methods exist in the literature as described in the following subsections:

2.6.1.Product Line UML-based Software Engineering (PLUS)

PLUS [3] is a UML based iterative software engineering method that extends the COMET [36] method for software product lines. PLUS models commonality and variability of product line families throughout the development life cycle. UML 2.0 is extended using stereotypes, constraints, and tagged values. PLUS has two main phases:

- Domain Engineering – The commonality and variability of the domain are analyzed through the following steps:

- Requirements modeling – user requirements are categorized as kernel, optional, and alternative use case models. Variation points and use case extensions are used to model variability. Feature models are constructed based on the use case models. Dependencies between use cases and features are explicitly outlined.
- Analysis modeling – the product line context and entity classes are modeled using static class diagrams. Classes are categorized as kernel, optional, and variants. Objects interactions in use cases are modeled by UML dynamic communication diagrams. State based systems are modeled by UML state chart diagrams.
- Architectural design modeling – a component based architecture is designed for the software product line. Connections and communication patterns among concurrent objects are explicitly designed. Software architectural patterns are used throughout the architecture.
- Application Engineering – member applications are instantiated by tailoring the product line architecture based on selected users' features.

2.6.2.Feature Oriented Domain Analysis (FODA)

FODA is a domain analysis method based on identifying the common and distinguishing features of family of systems [37]. The FODA method consists of three main phases:

- Context Analysis – The scope of the domain is analyzed by examining its external environment. The relation between the domain and the external environment is represented in a context model.
- Domain Modeling – The commonality and variability of the domain are analyzed thru the following steps:
 - Feature analysis – end users' functional requirement are analyzed and categorized into mandatory, optional, and alternative features. The relations among the feature of the domain are represented in a tree-like feature model.
 - Information analysis – domain data requirements are analyzed by focusing on domain knowledge entities and their relationship. The outcome of this phase is represented in entity relationship (ER) or object oriented (OO) models.
 - Operational analyses – behavioral aspects of the domain are analyzed with a focus on data and control flow. Domain functions are identified and sequenced to satisfy previously identified features. The outcome is represented in operational models.
- Architecture modeling – a high level software design of the overall structure of the domain is developed. Concurrent processes, domain modules, and their relationships are represented in the architectural model.

2.6.3.Reuse-driven Software Engineering Method (RSEB)

RSEB [38] is a use case driven domain modeling method that consists of several steps:

- Object-oriented business engineering – use cases are analyzed to derive automatable business processes.
- Application family engineering – a layered architecture is developed for the domain model.
- Component system engineering – reusable components and their connections are developed.
- Application system engineering – selected applications from the domain are developed based on the architecture.

RSEB uses variation points to express variability throughout the development process.

Variation points describe the location of variable behavior in use cases and architecture.

2.6.4. Family-Oriented Abstraction, Specification, and Translation (FAST)

FAST [39] is a method for developing families of systems and environments for deriving family members. The main processes of FAST are:

- A process for defining commonalities and variability for the family – parameters are used to express variations.
- A process for producing family members – parameters are mapped into templates which define family members.

- A language for specifying family members – a configuration language that captures family variations thru parameterization.
- Generating software from specifications – a code generating environment that generates executable family members.

2.6.5.KobrA

KobrA [40] is a component based incremental method for developing product lines architectures. The method can be used for both single and family of systems. The most important steps of KobrA are:

- Framework engineering – a generic reusable framework that represents all products variations of the software family is developed. This step consists of: Context realization, Component specification, and Component realization.
- Application engineering – specific products are instantiated from the generic framework to meet unique customers' requirements.

2.7. Feature Modeling

Feature modeling is rooted in the seminal work of Kang et al. [37] in the Feature Oriented Domain Analysis (FODA) method. A feature is a requirement that is present in one or more members of the product family [3]. Feature models are widely used in the SPL requirements commonality and variability analysis phase. Specifically, they are used to model the possible requirements configurations of SPL member applications.

Feature models are normally represented in tree-like graphical notations [37]. Although several notations exist, only UML based notations are investigated in this literature review.

Commonality and variability analysis is directly applied on class diagrams in a traditional OO analysis phase [41]. The stereotype <<V>> is used to denote optional features and the lack of this stereotype to denote common features. In addition, they apply this convention to the operations and attributes of classes. However, this approach is not expressive enough as it doesn't address alternative features and their dependencies. Also, this approach does not directly address variability analysis in the requirements phase.

A <<Variant>> stereotype is introduced [42] to denote optional features in class diagrams. Furthermore, a tagged value with the keyword 'feature' is attached to each <<Variant>> class. The purpose of this tagged value is to provide traceability between features in the requirements models to classes in the analysis model. However, no notation is provided to denote feature dependencies or feature groups.

A simple feature meta-model [43], using UML meta-class diagrams, is introduced. The meta-model consists of a *Feature* meta-class which is specialized by *CommonFeature*, *VariableFeature*, *FunctionalFeature*, and *NonFunctionalFeature* meta-classes.

VariableFeature aggregates two meta-classes: *VariationPoint* and *Variant*. Meta-associations and dependencies are used to depict feature relations. In addition, predicate logic expressions are provided to describe feature relationships. These expressions are valuable contribution, since they can be translated to Object Constraint Language (OCL) to constraint feature models' dependencies.

An approach for mapping the traditional tree-like feature models to UML [44] is presented. Model templates, expressed in the desired UML notation, are created to describe all possible combination of features from the feature model. These templates are annotated with “*presence conditions*” and “*meta-expressions*”. Presence conditions determine the inclusion or exclusion of features in the target template based on features information from the source model. Meta-expressions describe feature attributes from the source model as well. To carry out the transformation, “*template instances*” are created by evaluating the presence conditions along with the meta-expressions. This approach is interesting, because it provides a method to transform feature models to virtually any model that is MOF-based (Meta Object Facility) [45]. However, the mapping approach assumes that the starting source feature model is expressed as a tree and not as a generic model.

In the PLUS method [3], feature models are derived from variant use case models and represented by UML class diagrams. PLUS uses stereotypes to categorize reusable requirements, i.e. Features as shown in Table 2.1.

Table 2.1 PLUS Feature Stereotypes

Feature/Feature Group	Description
<<feature>>	Represents the top-level feature element
<<common feature>> extends <<feature>>	These features are present in all members of the product line
<<optional feature>> extends <<feature>>	These features are present in some members of the product line
<<alternative feature>> extends <<feature>>	Some members of the product line need to choose from alternative features
<<default feature>> extends <<feature>>	Alternative features may specify a default feature.
<<parameterized feature>> extends <<feature>>	Parameterized features define a product line parameter whose value needs to be defined at system configuration time
<<feature group>>	The top-level feature group element
<<zero-or-one-of feature group>> extends <<feature group>>	Zero or one feature can be present in the product line member
<<exactly-one-of feature group>> extends <<feature group>>	Exactly one feature can be present in the product line member
<<at-least-one-of feature group>> extends <<feature group>>	At least one feature must be present in the product line member
<<zero-or-more-of feature group>> extends <<feature group>>	Zero or more features can be present in the product line member.

In addition, feature dependencies are depicted as associations in the class diagram. Associations, such as ‘requires’ and ‘mutually includes’, are used to illustrate feature relationships. It should be noted this feature modeling notation is adopted by this dissertation as explained in Chapter 4.

2.8. Multiple-View Modeling

In his seminal work on multiple-view architectures, Kruchten [46] argues that it is unwieldy to capture the architecture of software-intensive systems on one diagram. Further, he notes that such diagrams usually do not address the concerns of the diverse stakeholders of such systems. To that end, Kruchten proposes a model for describing the architecture of software-intensive systems via the use multiple views or perspectives (4+1 View), where each view addresses specific set of concerns. The 4 views are: logical, process, physical, and development. The 1 view is the use case view which ties the 4 views together.

A multiple-view requirements engineering framework is described where each view is defined based on viewpoints held by actors or agents [47]. Viewpoints template are described which they can be instantiated for specific requirements engineering techniques. Inter-Viewpoint communication relationships are described to help in multiple viewpoint integration.

A dynamic navigation approach, of modeling information, in multiple-view models is proposed [48]. The approach allows developers to automatically locate information in

multiple views regardless of the notation used. An underlying representation is defined to describe the multiple views and mappings among the views.

2.9. Meta-Modeling

In language modeling, a meta-model is a model that describes a model. In other words, a meta-model is a model used to specify a language [49]. Meta-models are used to capture the essential features and properties of languages [50]. In a natural language analogy, all languages have grammars that describe their structure. Natural languages' grammars are the *meta-models* that describe the proper structure of languages. In computer science, programming languages have meta-models called Backus–Naur Form (BNF) [49] that describes their valid syntax.

Another term used for language meta-models is *abstract syntax* [50], which is the underlying and unifying structure of a language. The language itself is referred to as the *concrete syntax* [50]. For example, in UML, the Meta-Object Facility (MOF) [45] meta-model is the abstract syntax and the actual UML language diagrams are the Concrete Syntax. Most modelers use the concrete syntax of languages, while the abstract syntax is used by language engineers who normally create or modify languages.

Meta-models describe modeling languages at a higher level of abstraction than the language itself [50]. Hence, concrete model, like a UML class diagram, is an instance of its corresponding meta-model, which is MOF. Meta-models are especially useful to tools developers since they can be used to describe several languages in a uniform way.

It should be noted that meta-modeling has existed for a long time and has been used in data modeling for example.

2.10. Model Driven Architecture (MDA)

MDA [51] is an Object Management Group (OMG) initiative that promotes development practices where models are used as first class entities. Software development is driven by constructing models in all phases of the development life cycle. Platform Independent Models (PIM) [52] capture business logic details independent of any technological platform. Platform Specific Models (PSM) represent technology specific concerns like software languages and environments. PIMs are transformed into PSMs using predefined transformation definitions. Eventually, PSMs are transformed into code.

Evolving business requirements are handled by changing PIMs and generating PSMs using the transformation definitions. Transformation definitions [45] consist of precise transformation rules and mappings that transform elements in the PIM into elements in the PSM automatically. Transformation rules are written using formal language that can be understood by machines.

The most widely used MDA modeling language is the Unified Modeling Language (UML). However, other languages can be used if they are based on Meta Object Facility (MOF) [45] meta-model.

MDA aspires to improve productivity by letting developers focus on the PIM and having them generate the PSM and code. Portability is gained by developing transformation rules for new platforms while keeping the PIM intact. In addition, documentation

becomes always current and in sync with operational systems since all work is done on the models.

The MDA vision depends heavily on tool support for building the PIM and PSM, the formulation of transformation rules, and code generation.

3. Research Approach

3.1. Introduction

The purpose of this research is to develop a multiple-view variability modeling and meta-modeling approach to address the modeling of variable service-oriented systems in a systematic, unified, and platform-independent way.

When building variable systems, i.e. application families, software engineers typically have the following design alternatives:

Traditional development approach, where requirements are gathered for current capabilities at hand without considering functional requirements that can change in the future. Consequently, analysis, design, testing, and implementation commence to build a single application. When future requirements arise, the aforementioned development activities are repeated to build another single application that satisfies the new requirements. This process is repeated every time new requirements arise even if the new desired application is only slightly different from the original application.

Software Product Line (SPL), aka application family, development approach. Instead of developing new single systems from scratch for new requirements, current and anticipated future requirements are gathered and analyzed to create a family of similar but slightly different applications. Hence, analysis, architecture, design, and testing

artifacts are created for the entire SPL upfront. Consequently, components are built for the entire application family and stored in repositories for future reuse. When anticipated requirements arise, new member applications are derived from the variable component architecture. Details of SPL engineering are described in Chapter 2.

Service-Oriented Software Product Lines. This alternative is similar to the SPL approach, however instead of developing software components for the entire SPL upfront, this approach employs services that might be developed by external providers. The analysis and design activities are based on service interfaces and business process workflows. Services could be discovered and bound at member application derivation time based on their exposed interfaces. Obviously, considerable time and efforts are saved since the developing organization does not have to create a complete variable component architecture upfront as described in the previous design alternative.

This research develops an approach to facilitate and realize the service-oriented SPL design alternative mentioned above. To achieve this goal, this research develops a multiple-view model that specifies the relationships between variable service views. Additionally, this research develops a multiple-view service variability meta-model that formalizes the relationships and mappings of the multiple-view service variability model. Consistency checking rules between the multiple service views are developed based on the meta-model. Furthermore, this research presents an automated model-driven framework that realizes the aforementioned meta-modeling and modeling approach. A proof-of-concept tool prototype is developed to realize the automated framework. The

tool prototype is used to model and execute service oriented product lines (SPLs) and to ensure the consistency of the multiple views of SPLs.

3.2. Research Approach

Since services in SOA could be used by different clients with varying functionality, SOA variability modeling can benefit from software product lines (SPL) variability modeling techniques. Applying SPL concepts, service oriented systems can be modeled as service families. In particular, the research approach integrates SPL concepts of feature modeling and commonality/variability analysis, meta-modeling, multiple-view modeling, and service modeling to model SOA variability. The main goal of SPL is the reuse-driven development of SPL member applications by utilizing reusable assets from all phases of the software development life cycle. This goal is similar to the goal of SOA where reusable service development is a common theme. The approach in this research combines SPL variability modeling concepts [3] with SOA concepts, as represented in the Unified Modeling Language (UML) [53] and the newly released standard SoaML[54]. Such an approach facilitates variability modeling of service family architectures in a systematic and platform independent way.

At the heart of the approach in this research is a meta-model that models requirements and architectural views of variable service oriented systems. The meta-model captures variability in the service views and adds a feature view that addresses the variability in the SOA system in a unified manner. The meta-model also captures relationships among the service views, and among the feature and service views. The meta-model is used to

help developers specify requirements, relate requirements to architectural artifacts, and create new applications based on changing requirements.

A proof-of-concept service-oriented engineering environment (SoaSPLE) is developed to validate the approach. This tool prototype is used in carrying out two case studies to further validate the approach.

3.3. Relation to Existing Research Approaches

The research approach builds on existing research in the following ways:

Feature Modeling [37] – it is difficult to get a complete picture of the variability in the service architecture, because it is dispersed among multiple views. To get a full understanding of the variability in the service architecture, it is necessary to have one view that focuses entirely on variability and defines dependencies in this variability. That is the purpose of feature modeling.

SPL commonality/variability management [3] – since SOA development approaches lack a systematic way to handle variability management concerns, SPL principles are exploited to create a systematic methodology to handle variability management concerns in variable SOA systems.

Multiple view modeling [2], [47], [48] – multiple-view modeling techniques are used to model various views of SOA and to define the relationships among these views.

Meta-modeling of SPL phases [55], [50] – meta-modeling principles are used to formalize the multiple-view service variability model and to help in defining consistency checking rules among service views.

Model Driven Architecture [51], [56], [52], [45] – MDA is exploited to create a platform independent service variability approach and in designing an automated framework to realize this approach.

The following sections detail the research approach components:

3.4. Feature Modeling and Meta-Modeling

Traditionally, feature modeling [3], [37] is used to model the reusable requirements of SPLs. In this research, in addition to modeling reusable requirements, feature modeling is used to model the variability of the service architecture that is dispersed among the multiple service views. In essence, feature modeling is exploited to produce a Feature View that focuses entirely on variability and defines dependencies in the variability of the service architecture. The Feature View is described in Chapters 4 and 5.

3.5. Multiple View Service Variability Modeling

A multiple-view model is developed to model the variability of requirements and architectural views of SOA systems. The multiple views are integrated via explicit relationships that describe their dependences.

The multiple-view variability model is intended to provide a modeling notation for modelers to help in modeling service-oriented software product lines. The multiple-view service variability model is described in Chapter 4.

3.6. Multiple View Service Variability Meta-Modeling

A multiple-view service variability meta-model is created to provide a formal description of the aforementioned multiple-view service variability model. The meta-model contains Meta-Views that model each View in the multiple-view model. In addition, the meta-model captures the relationships between the service views and the relationships between the feature view and service views.

The multiple-view service variability meta-model is embedded within the proof-of-concept prototype to serve as Abstract Syntax [49] for the multiple-view model's Concrete Syntax [49]. The meta-model is described in Chapter 5.

3.7. Consistency Checking and Mapping Rules

To ensure consistency among the multiple service views, consistency checking rules are derived from the semantic relationships between the multiple views in the meta-model. In addition, mapping rules are derived to provide mapping between the feature view and service views.

The consistency checking and mapping rules are annotated with the multiple-view service variability meta-model in the proof-of-concept prototype to provide an automatic consistency mechanism for modelers. Consistency and mapping rules are described in Chapter 5.

3.8. Model-Driven Service-Oriented Product Line Engineering Framework

This research exploits the model driven architecture (MDA) principles to create an automated framework that realizes the multiple-view service variability approach for building service-oriented SPLs.

The framework provides model-driven techniques to design service-oriented SPLs, automation for service-oriented product line engineering, and model-driven techniques to handle the variability of SOA middleware environments. This approach is described in Chapter 6.

3.9. Proof-of-Concept Tool Prototype

To realize the aforementioned automated framework, this research built a model-driven Service-Oriented SPL Engineering proof-of-concept tool prototype (SoaSPLE). The goals of the prototype are: to 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, and to deploy, execute, and test member applications of the SPL. The prototype is described in Chapter 6.

3.10. Approach Validation

The objective of the validation is to evaluate the approach in this research with regard to the following properties:

The multiple views of the service-oriented software product line are consistent with each other.

The multiple-view service variability model is compliant with the underlying multiple-view service variability meta-model.

Derived software product line member applications are valid service-oriented SPL members.

To achieve the aforementioned validation objectives, the validation procedure is divided into two main testing tasks:

Unit Testing – this type of testing tests each element and relationship in the multiple-view service variability meta-model. Unit testing is needed, because the case studies may not exercise every part of the meta-model.

System Testing – this is a system-wide testing that tests the running service-oriented applications of the SPL end-to-end.

Chapter 7 describes the validation approach and unit testing, while Chapters 8 and 9 detail system testing through two case studies.

4. Multiple-View Service Variability Model

4.1. Introduction

In this chapter, the multiple-view variability modeling approach that addresses SOA variability concerns is introduced. The approach integrates software product lines (SPL) feature modeling techniques with service modeling to model variability in multiple views pertinent to SOA. Although earlier research (Chapter 2) provides ways to model variability in service views, variability of each view has been addressed only individually without relation to other views. By integrating feature modeling with multiple-view service modeling, consistency between the multiple views can be checked and then enforced. The approach is intended to be platform-independent in which SPL variability modeling [3] concepts are combined with SOA concepts, as represented in UML and SoaML [54].

Erl [15] describes service-oriented systems as having multiple perspectives where these perspectives depend on each other. In essence, each perspective describes a distinct view of the whole SOA system. In this research, the different SOA perspectives are formalized into multiple Requirements and Architectural views.

Kruchten [46] introduced the 4+1 view model of software architecture, in which he advocated a multiple view modeling approach for software architectures that addresses

the needs of distinct stakeholders. In the 4+1 multiple-view model, the use case view is the unifying view (the 1 view of the 4+1 views). The approach in this research describes a multiple view modeling approach for service-oriented software product lines in which the unifying view is the feature model. In particular, feature modeling provides the added dimension of modeling variability in service-oriented software product line architectures. The other 4 views in the approach consist of 2 Requirements views (Service Contract and Business Process) and 2 Architectural views (Service Interface and Service Coordination).

Each view of the multiple-view model is depicted by a UML/SoaML diagram that consists of new modeling elements that were created by directly extending the UML meta-model. In addition, the relationships between the multiple service views are described. Finally, the relationships that relate features to service views are developed

4.2. Using SPL Concepts to Model SOA Variability

Although, there are many differences between typical software product lines and service-oriented architectures, this section analyzes how SPL concepts can be used to model SOA variability. Since services in SOA could be used by different clients with varying functionality, SOA variability modeling can benefit from SPL variability modeling techniques. Service-oriented systems can be modeled as *service families*, similar to the concept of SPL. The main goal of SPL is the reuse-driven development of SPL member applications by using reusable assets from all phases of the development life cycle. This

goal is similar to the goal of SOA where reusable application development is a common theme.

SOA and SPL differ in the following ways:

- In SPL, components (core assets and variants) are designed and implemented a priori and usually owned by the same developing organization, whereas in SOA, services are usually developed by external providers who are unaware of their clients.
- SOA development practices focus on automating business workflows by assembling services, whereas the focus of SPL is on developing application families.
- Reuse in SPL is utilized in all phases of the development life cycle using all types of assets, however in SOA, only services are reused [57].
- SPL approaches have explicit techniques to model variability. However, SOA approaches rely on industry best practices and ad-hoc techniques [57].

Existing approaches to handling variability in SOA [6], [4], [7], [58], [59], [60], [61] have used SPL concepts to model variability in service families (this is discussed in more detail in Chapter 2). However, these approaches do not provide a treatment of multiple variability concerns in SOA in a *unified framework*. Further, existing research mainly treats SOA variability issues in a platform specific way by focusing on Web Services and orchestration languages such as the Business Process Execution Language (BPEL).

4.3. Service Contract Variability View

This *Requirements* view models service contracts and service participants. Service contracts are prescribed by collaborating organizations to govern and regulate their interactions. Service contracts may include service interfaces, policies, and service level agreements.

To model the service contract view, SoaML's **ServiceContract** element, which specifies the agreement between providers and consumers, is used. It should be noted that SoaML's **ServiceContract** element is based on UML's **Collaboration** element, which is represented by dashed bubbles (Fig. 4.1). To model variability, a **ServiceContract** is specialized into kernel, optional, and alternative **ServiceContracts**. Kernel contracts are required by all members of an SPL, whereas optional contracts are required by only some members. An alternative contract is a variant of a kernel or optional contract to meet a specific requirement of some SPL members. Fig. 4.1 depicts service contracts of an E-Commerce SPL.

It should be noted that UML 2.0 specification allows the use of multiple stereotypes per

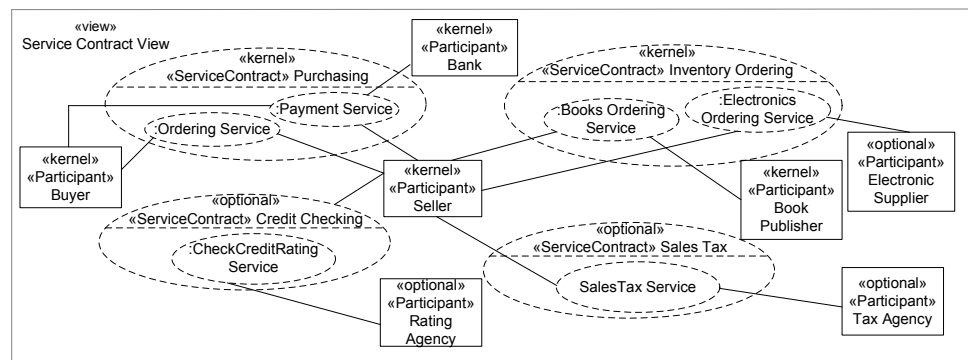


Fig. 4.1 Service Contract View of an E-Commerce SPL

modeling element. Therefore, this research uses this feature by attaching a stereotype that conveys an SOA concept and another stereotype that conveys a reusability concept for each element in the multiple-view service variability model. However, this usage of stereotypes is for display purposes only, since the multiple-view service variability model consists of meta-classes that directly extend the UML meta-model.

Each service contract prescribes roles for the organizations participating in it. Hence, the Service Contract View also models contract participants. **Participants** are entities that abide by service contracts and provide or require service interfaces. Service interfaces are discussed in the Service Interface View below. To model contract participants, SoaML's **Participant** element is used, which specifies providers or consumers of services. This element extends the UML **Class** element. A **Participant** is specialized into kernel, optional, or alternative **Participants**. An example of the Service Contract View is given in Fig. 4.1.

4.4. Business Process Variability View

This *Requirements* view models the workflow of business processes. **Participants** can define internal business processes to conduct their business. In this way, each organization can define its own business processes while satisfying inter-organization business contracts. This view consists of one or more business process models.

Neither SoaML nor UML explicitly model business process workflow. Since a business process workflow is composed of a sequence of activities, UML **Activity** diagrams are used to model business processes. From an SPL perspective, each activity in the **Activity**

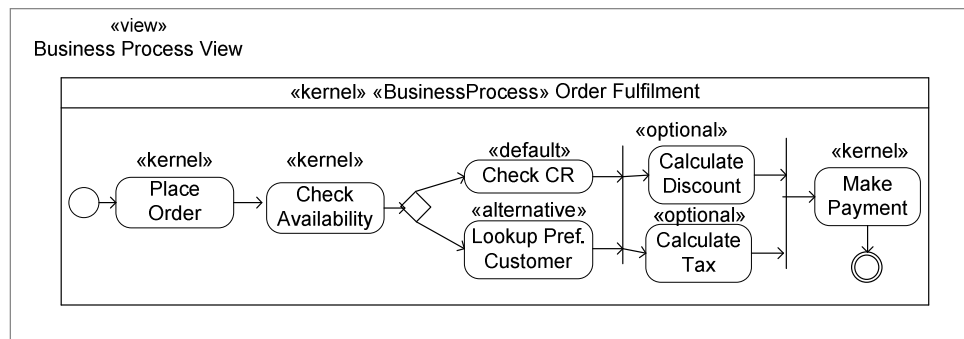


Fig. 4.2 Business Process View of an E-Commerce SPL

model is specialized into kernel, optional, or alternative **Activity**. An example of the Business Process View is given in Fig. 4.2, where the Seller **Participant** has its own internal business process modeled as an Order Fulfillment business process. Note that this business process is modeled as an SPL service activity diagram consisting of a sequence of service activities (kernel, optional, default, and alternative), which can be tailored into a service application business process in the SPL application derivation phase (Chapter 6).

4.5. Service Interface Variability View

Services expose their capabilities through service interfaces only. This *Architectural* view models service interfaces that specify the operations *provided* or *required* by **Participants**.

Service interfaces are modeled by using SoaML's **ServiceInterface** class. Service Interfaces are specialized into kernel, optional, or variant **ServiceInterfaces**.

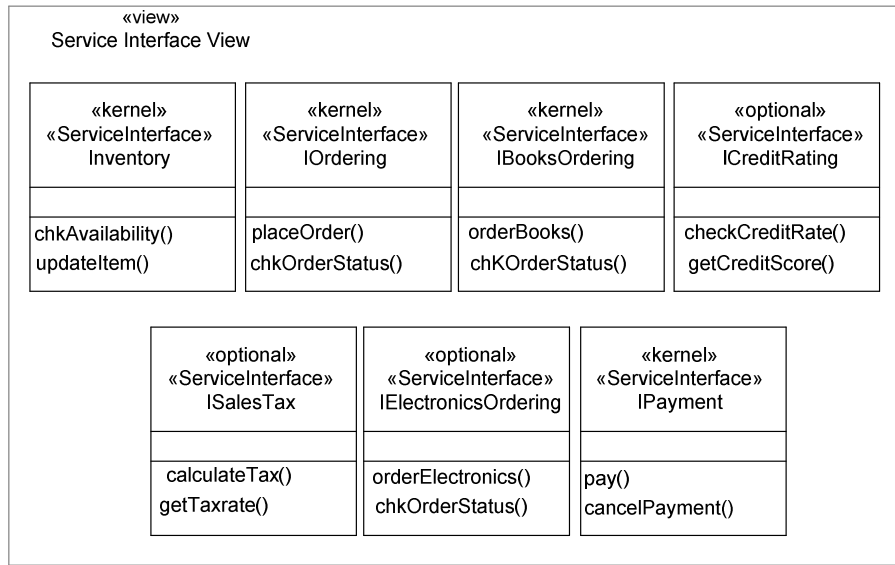


Fig. 4.3 Service Interface View of an E-Commerce SPL

An example of the Service Interface View is given in Fig. 4.3. An organization that plays the Seller role in the Purchasing **ServiceContract** must be able to implement and advertize an Ordering Service exposed through the Ordering **ServiceInterfaces**, which enables Buyer **Participants** to order goods from Seller **Participants**.

4.6. Service Coordination Variability View

The Service Coordination View models the sequencing of service invocations. It should be noted that this is an *Architectural* view that is related to the business workflow described in the Business Process View. In other words, each business process, i.e., **Activity** Diagram, in the Business Process View is associated with a **ServiceCoordinator** in the Service Coordination View. It should be noted that this

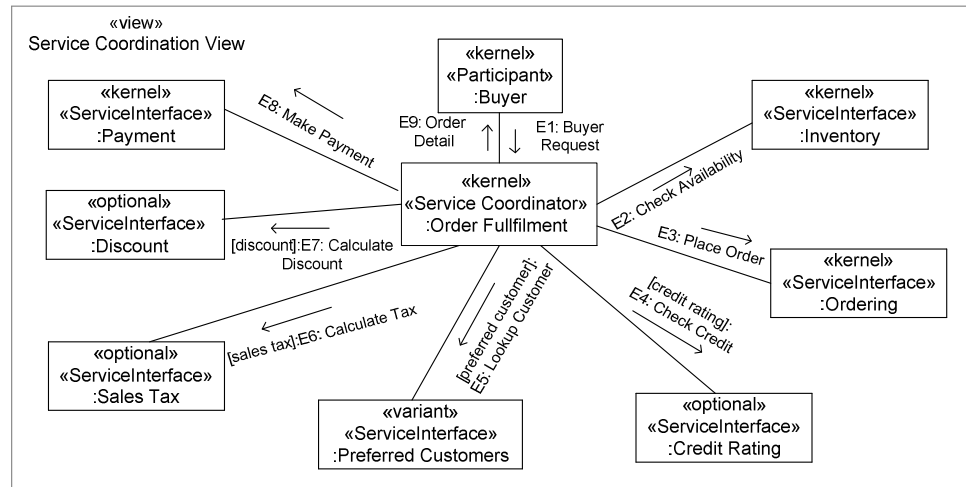


Fig. 4.4 Service Coordination View of an E-Commerce SPL

coordination is *centralized* where one **ServiceCoordinator** solely coordinates one business process.

Services should be self-contained and loosely coupled. In order to have a high degree of reuse, dependencies between services should be kept to a minimum [62]. Hence, service coordination is used in situations where multiple services need to be accessed and access to them needs to be coordinated and/or sequenced [63].

The Service Coordination View consists of **ServiceCoordinator** elements which are direct extensions of UML Class meta-class. **Service Coordinators**, depicted on UML communication diagrams, interact with clients and services. Services in the Service Coordination View are modeled via the **ServiceInterface** elements. In addition, from a reuse perspective, a **ServiceCoordinator** is specialized into kernel, optional, or variant **ServiceCoordinators**. **ServiceCoordinators** receive messages from clients and/or services, and send messages to clients and/or services. The sequencing of multiple service invocations is encapsulated within the Coordinator [62]. Service Coordination can be

categorized by type of coordination (*independent, distributed, or hierarchical*), and the degree of concurrency (*sequential or concurrent*) [63]. An example of the Service Coordination View is shown in Fig. 4.4, where the Order Fulfillment **ServiceCoordinator** coordinates service invocations for the Order Fulfillment Business Process (Fig. 4.2). Notice how the Order Fulfillment **ServiceCoordinator** encapsulates all sequencing logic and only sends and receives messages to and from services thus minimizing coupling among services. In addition, notice how messages are annotated by feature conditions between square brackets. These conditions act as guards that allow messages to be fired only if the feature conditions are true.

4.7. Feature View

With multiple-view service variability modeling, it is possible to define the variability in each view. However, it is difficult to get a complete picture of the variability in the service architecture because it is dispersed among the multiple views. To get a full understanding of the variability in the service architecture, it is necessary to have one view that focuses entirely on variability and defines dependencies in this variability. That is the purpose of the Feature View described in this section.

Feature modeling is the process of identifying reusable requirements or characteristics of members of an SPL in terms of features and organizing them into a feature model. The Feature View consists of **Feature** elements which are direct extensions of the UML Class meta-class. Feature models are used to express and manage similarities and differences among different family members in an SPL. **Features** are specialized into kernel,

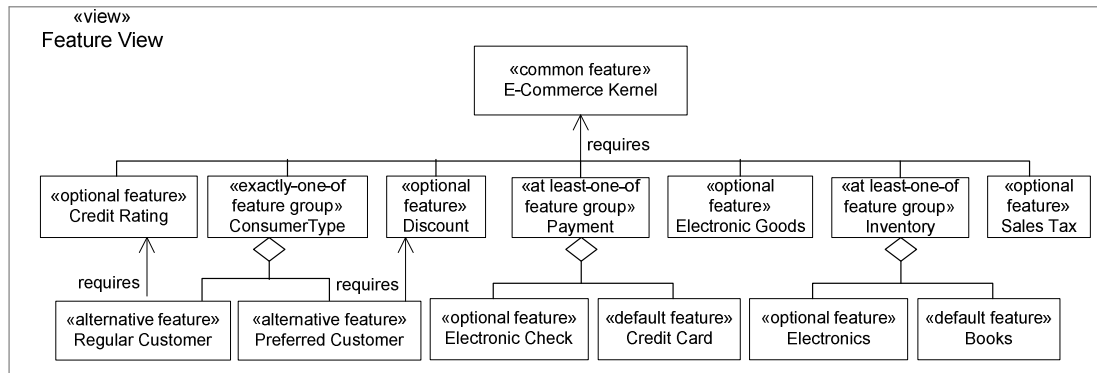


Fig. 4.5 Feature View of an E-Commerce SPL

optional, or alternative **Features**. Kernel features among products in an SPL are mandatory, while different features among them may be optional or alternative features. Related features can be grouped into feature groups, which constrain how features are used by a product of an SPL. Feature modeling is discussed in more details in Chapter 2. A feature model of an E-Commerce SPL is shown in Fig. 4.5. The feature model consists of UML classes that denote reusable requirements, i.e. features. In addition, the feature model contains feature group classes which are aggregations of optional or alternative features. In addition, feature dependencies are depicted as associations in the class diagram.

It is necessary to conduct commonality and variability analysis, represented in feature models, in order to understand how service-oriented systems can change in reaction to changing requirements. In addition, as will be shown in Section 4.9, the Feature View serves as a *unifying* view since features can be mapped to multiple service views.

4.8. Multiple-View Service Variability Model Relationships

In this section, the relationships between views of the service variability multiple-view model are defined. It is important to develop these relationships to gain an understanding of what happens in one view if a change happens in another view. This understanding serves as a basis for the consistency checking rules described in Chapter 5. In addition, these relationships illustrate how the Requirements views can be mapped to Architectural views and how the Feature View serves as the unifying view for all views.

4.8.1. Intra-View Relationships

The associations and dependencies inside each view are defined. A **ServiceContract**, in the Service Contract View, is associated with two or more **Participants**, because a **ServiceContract** defines the relationships for participating entities in the SOA system. For example, in Fig. 4.6b, the Purchasing **ServiceContract** is associated with Buyer and Seller **Participants**. **ServiceContracts** may contain other **ServiceContracts** to form composite contracts [54].

Participants provide or require services in the Service Contract View (Fig. 4.6b). Services are exposed through **ServiceInterfaces** in the Service Interface View (Fig 4.6d). **ServiceCoordinators**, in the Service Coordination View (Fig. 4.6e), coordinate one or more services as they send and receive messages to/from services. It should be noted that services are exposed through **ServiceInterfaces** in the Service Interface view.

It should also be noted that the intra-view relationships of the Feature View are adopted based on previous research [55] and explained in Chapter 2.

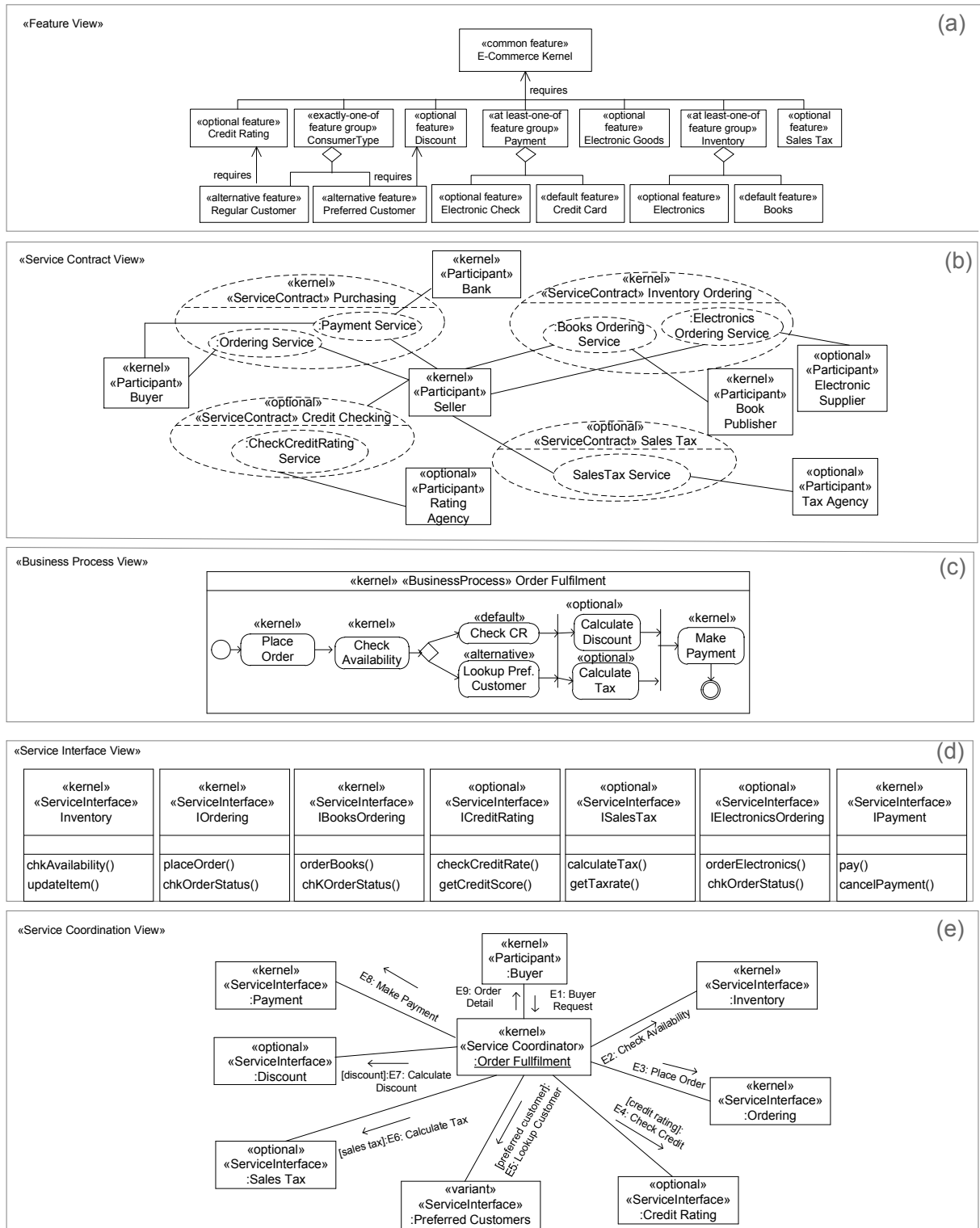


Fig. 4.6 E-Commerce SPL Multiple-View Service Variability Model

4.8.2. Inter-View Relationships

The associations and dependencies between the multiple views of the multiple-view service variability model are defined.

A service-oriented SPL has one or more **ServiceContract** elements. For example, in the Service Contract View of Fig. 4.6b, an E-Commerce SPL has *Purchasing*, *Credit Checking*, *Inventory Ordering*, and *Sales Tax* **ServiceContract** elements.

Participant elements provide or require **ServiceInterface** elements, because participating entities only interact through interfaces to minimize coupling among services. For example, in Fig. 4.6b, the *Tax Agency* **Participant** provides the *ISalesTax* **ServiceInterface** (Fig. 4.6d), while the *Seller* **Participant** (Fig. 4.6b) requires the *ISalesTax* **ServiceInterface**.

Participant elements can define their own internal business processes. Business process details are not known to other **Participants** and can change without notice to other **Participants** in the service-oriented system. This minimizes coupling and allows **Participants** to change their business processes without affecting the operation of other participants in the system. For example, in Fig. 4.6c, The *Seller* **Participant** has an *Order Fulfillment* business process, which is comprised of **Activity** elements.

Activity elements can be either *local* or *service* elements. Local **Activities** are executed within the **Participant** execution environment, e.g. ‘*Calculate Discount*’, whereas Service **Activities** are executed by calling external services exposed through **ServiceInterfaces**. For example, in Fig. 4.6c, the *Calculate Tax* **Activity** is executed by calling a *SalesTax* service exposed through the *ISalesTax* **ServiceInterfaces** (Fig. 4.6d).

Activities in the Business Process View are realized by calling operations of services exposed by **ServiceInterfaces**. For example, the ‘Check CR’ **Activity** in the Order Fulfillment business process (Fig. 4.6c) is realized by calling the `checkCreditRate` operation of the Credit Rating service exposed through the *ICreditRating* **ServiceInterface** (Fig. 4.6d).

ServiceCoordinator elements, in the Service Coordination View, coordinate the sequencing of service invocations exposed by **ServiceInterfaces** in the Service Interface View. For example, the Order Fulfillment **ServiceCoordinator** (Fig. 4.6e) invokes operations on the Credit Rating service exposed through the *ICreditRating* **ServiceInterface** (Fig. 4.6d).

Messages in the Service Coordination View (Fig. 4.6e) trigger operation invocations on **ServiceInterface** in the Service Interface View (Fig. 4.6d). For example, the ‘Calculate Tax’ **Message** in Fig. 4.6e is triggered by an invocation of an operation on the *ISalesTax* **ServiceInterface** (Fig. 4.6d).

Elements in one view of the multiple-view service variability model affect elements in other views. For example, in Fig. 4.6c, when the *Calculate Tax* **Activity** is added to the *Order Fulfillment* Business Process View, a *Sales Tax* **ServiceContract** is introduced into the E-Commerce SPL in the Service Contract View (Fig. 4.6b). Consequently, a *Tax Agency* **Participant** is also added, which provides an *ISalesTax* **ServiceInterface** in the Service Interface View (Fig. 4.6d).

It should be noted that these relationships are enforced by an underlying meta-model described in Chapter 5.

4.8.3.Feature to Service Contract View Relationships

A **Feature** is supported by one or more **ServiceContract** elements in the Service Contract View. For example, when feature *Credit Rating* (Fig. 4.6a) is selected, the *Credit Checking ServiceContract* is configured into the SPL member application (Fig. 4.6b). The variability stereotype on a **ServiceContract** dictates the type of feature it may map to. For instance, an optional feature (e.g., *Credit Rating*) can only be supported by optional service contracts (e.g., *Credit Checking* service contract). Similarly, an alternative feature may be supported by alternative service contracts only.

A **Feature** is supported by one or more **Participants**. For example, when the *Electronic Goods* optional feature (Fig. 4.6a) is selected, which means the *Seller* will start selling electronic items in addition to books, the *ElectronicSupplier Participant* participates in the *InventoryOrdering ServiceContract* (Fig. 4.6b). Consequently, the *ElectronicsOrdering ServiceInterface* will be introduced into the *InventoryOrdering ServiceContract* (Fig. 4.6b). Hence, the selection of one feature in the Feature View is supported by two service model elements (contract and interface) in the service Contract and Interface views respectively.

4.8.4.Feature to Business Process View Relationships

A **Feature** is supported by one or more **Activities** in the Business Process View. For example, when the *Discount* optional feature is selected (Fig. 4.6a), which means that the system changes to provide the ‘Discount’ capability, the *Calculate Discount Activity* is

added to the *Order Fulfillment* Business Process (Fig. 4.6c). Thus, the *Discount* optional feature is supported by an optional *Calculate Discount Activity* in the Business Process View. **Features** may be supported by one or more nodes in the Activity diagram of the business process, or by an entire business process.

4.8.5.Feature to Service Interface View Relationships

Service Interfaces can support Features in three ways:

- A **Feature** is supported by one or more **ServiceInterfaces**. For example, if the *Credit Rating* optional feature is selected (Fig. 4.6a), the *Seller Participant* has to provide a new **ServiceInterface** that can interact with a credit rating agency. Thus, the *Credit Rating* optional feature is supported by an optional *Credit Rating ServiceInterface* in the Service Interface View (Fig. 4.6d).
- Abstract **ServiceInterfaces** – here, an abstract **ServiceInterface** is specialized differently for each member of the product line [3]. The main advantage of this approach is the isolation of each variation in a separate subclass. However, this approach suffers from the problem of combinatorial explosion when the number of features is large, and a subclass is needed for each feature and feature combination. In other word, there would be many subclasses of the parent **ServiceInterface**, which can become quickly unwieldy.
- Parameterized **ServiceInterfaces**– here, *one* parameterized service class is created with feature related configuration parameters which are assigned different values for different members of the product line [3]. The main advantage of this

approach is that there would be only one parameterized **ServiceInterface** instead of several variant classes. The main disadvantage is that the parameterized **ServiceInterface** is affected by more than one feature.

4.8.6.Feature to Service Coordination View Relationships

ServiceCoordinators can support Features in two ways:

- Specialization – **ServiceCoordinators** can be specialized, using inheritance, to model variability [3]. In this case, a parent coordinator class will be extended to realize selected features or feature combinations. The main advantage of this approach is the isolation of each variation in a separate subclass. However, this approach suffers from the combinatorial explosion problem explained above.
- Parameterization – **ServiceCoordinators** can be pre-designed with variation points based on **Features** in the feature model. For example, the Service Coordination View of the Order Fulfillment business process (Fig. 4.6e) has a pre-designed **ServiceCoordinator** with kernel and variant message invocations. Notice, that optional and variant messages on the service coordination communication diagram are preceded with the corresponding feature name in the feature model (Fig. 4.6a) between square brackets to identify a conditional message. These feature names act as guards that prevent a message sequence from being invoked if the corresponding feature has not been selected. Optional and variant message paths will be invoked if the corresponding features are selected in the target member application. The advantages and disadvantages of this approach

are the same as for the parameterization of the Service Interface View explained above.

4.8.7.Feature Dependency to Service Views Relationships

In a feature model, relationships between features are represented by feature dependencies, where features may require other features or where features are mutually exclusive of each other. In other words, a feature may depend on another feature, or a feature must not be included along with another feature [3].

If there is a dependency between two features that are supported by two different service model elements (e.g., service contract, or service interface), the dependency between the two features must map to a dependency between the two service model elements. For example, in the feature model (Fig. 4.6a), the Regular Customer feature requires the Credit Rating feature. Therefore, the Check CR Activity (Fig. 4.6c), which supports the Regular Customer feature, must relate to the ICreditRating ServiceInterface (Fig. 4.6d), which supports the Credit Rating feature . These relationships are enforced by the underlying meta-model as well (Chapter 5).

5. Multiple-View Service Variability Meta-Modeling

5.1. Introduction

The multiple-view variability modeling approach (Chapter 4) is based on a meta-model that precisely describes all views and view relationships. Each view in the multiple-view model is described by a corresponding meta-view in the meta-model. There are two Requirements meta-views, Contract and Business Process, and two Architecture meta-views, Service Interface and Service Coordination. In addition, there is a Feature meta-view that describes the Feature view in the multiple-view variability model.

The meta-modeling approach in this research builds on previous research [55], in which a multiple-view modeling and meta-modeling approach for SPL was described. The approach has a multiple-view meta-model that defines the different perspectives of SPL. The meta-model depicts life cycle phases, views within each phase, and meta-classes within each view. Consistency checking rules are specified based on the relationships among meta-classes in the meta-model. More details on this approach are covered in Chapter 2.

The meta-modeling approach in this research exploits meta-modeling to achieve the following goals:

- Model intra and inter views relationships of the multiple views of service-oriented systems.
- Describe variability of interrelated views of SOA to produce an overall description of service-oriented systems. In other words, the approach detects changes in a specific view if a change happens in different, but related, views.
- Describe variability of SOA in a unified way. Since the multiple views of SOA use different modeling notations, meta-modeling provides one notation to describe all views in one language at a higher level of abstraction.
- Specify variability of service-oriented systems in a platform-independent manner by using meta-classes that could be mapped to several technology platforms.
- Formulate consistency checking rules that ensure the consistency of the multiple-views of service-oriented systems as they change. OCL-based rules are determined from the meta-model to ensure multiple-views consistency.
- Use the multiple-view variability meta-model to produce a model-driven automated framework. MDA tools use meta-models to represent models at a high level and to derive model-to-model and model-to-code transformation definitions [52].

The following sections describe the multiple meta-views of the service variability meta-model, their relationships, and consistency checking rules.

5.2. Service Contract Variability Meta-View

SoaML's ServiceContract meta-class is used to model elements in this meta-view. This meta-class extends the UML Collaboration meta-class. To model variability, ServiceContract meta-classes are categorized as kernel, optional, or alternative (Fig. 5.1). The Service Contract View prescribes roles for the entities participating in it. Hence, this meta-view also models contracts' participants. Participants are entities that abide by

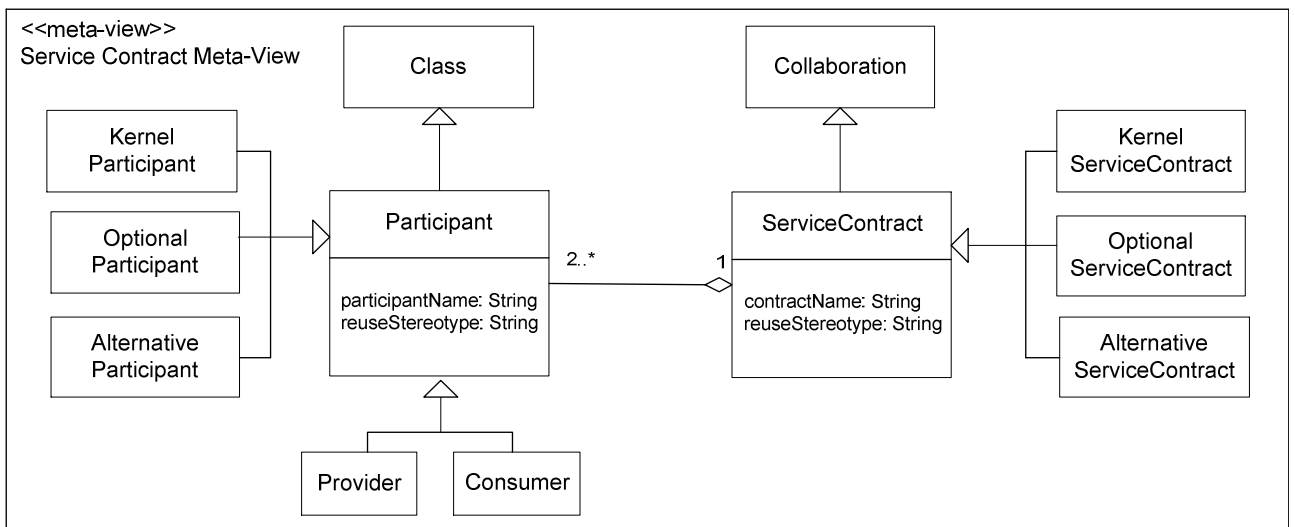


Fig. 5.1 Service Contract Variability Meta-View

service contracts and provide or require service interfaces. SoaML's Participant meta-class is used (Fig. 5.1) which specifies providers or consumers of services. This meta-class extends the UML Class meta-class. The Participant meta-class is specialized into Provider and Consumer Participant. Variability is modeled the same way as ServiceContract meta-classes, i.e. kernel, optional, and alternative.

5.3. Business Process Variability Meta-View

Neither SoaML nor UML explicitly model business process workflow. Since a business process is composed of a sequence of activities, UML **Activity** meta-classes are used to model meta-classes in this meta-view (Fig. 5.2). **Activity** meta-classes are categorized as

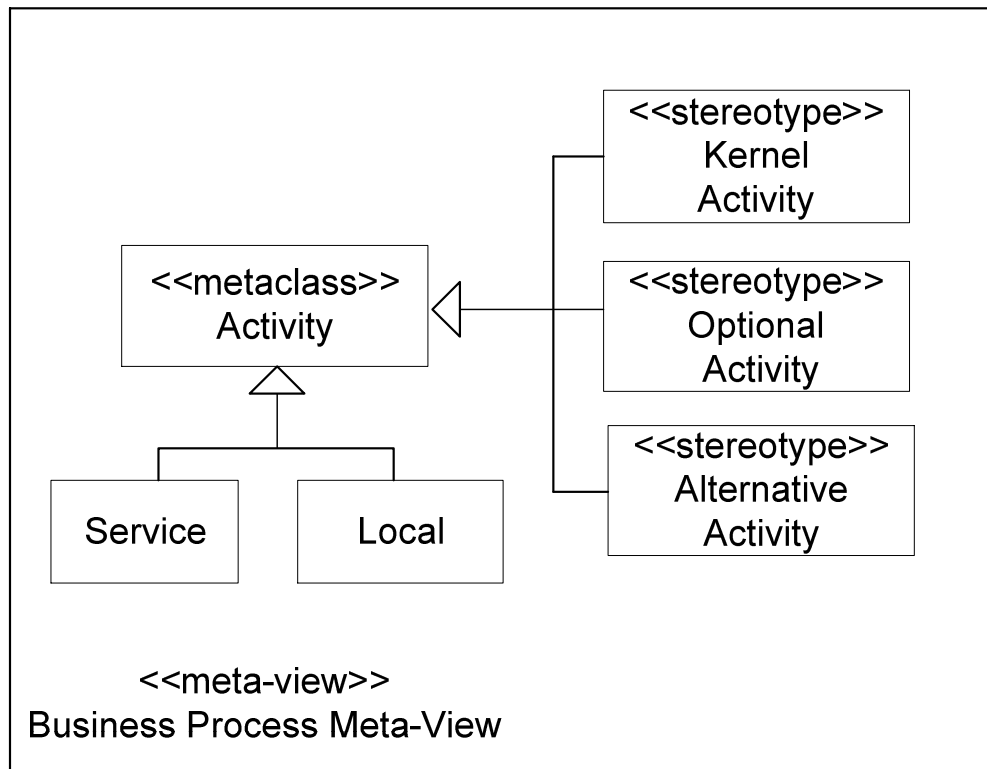


Fig. 5.2 Business Process Meta-View

kernel, optional, or alternative.

Activity meta-classes are specialized into Local and Service meta-classes (Fig. 5.2).

Local **Activities** are executed within the **Participant** execution environment, whereas Service **Activities** are executed by invoking external services exposed through **ServiceInterfaces**.

5.4. Service Interface Variability Meta-View

Service interfaces (Fig. 5.3) are modeled by SoaML's **ServiceInterface** meta-classes.

This meta-class extends the UML Interface meta-class. **ServiceInterface** meta-classes specify provided and required interfaces by **Participants**. A **ServiceInterface** is categorized as kernel, optional, or variant. It should be noted that the 'variant' categorization is used for the Architectural Views in contrast to 'alternative', which is used for the Requirements Views.

To manage complexity, variability meta-modeling for service **ServiceInterface** is restricted to the whole interface and the whole operation. In other words, selected features in the SPL could be supported by a new service **ServiceInterface** or by a new operation in an existing **ServiceInterface**.

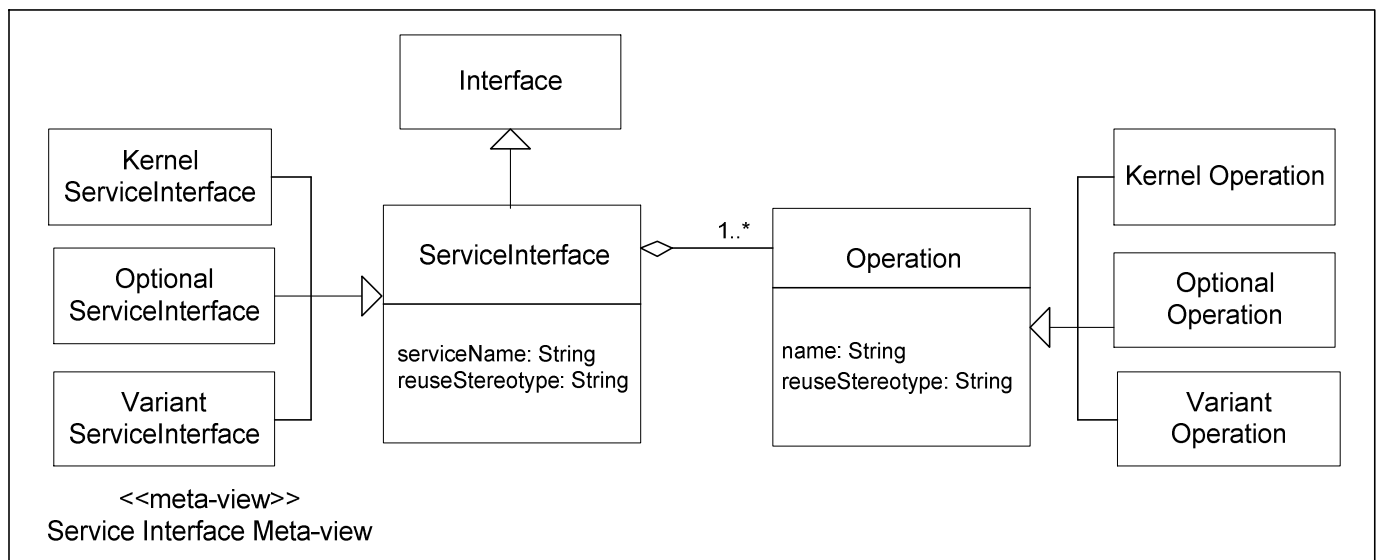


Fig. 5.3 Service Interface Variability Meta-View

5.5. Service Coordination Variability Meta-View

The Service Coordination Meta-View (Fig. 5.4) 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. MessageType extends UML DataType meta-class. ServiceCoordinator and MessageType are categorized as kernel, optional, or variant.

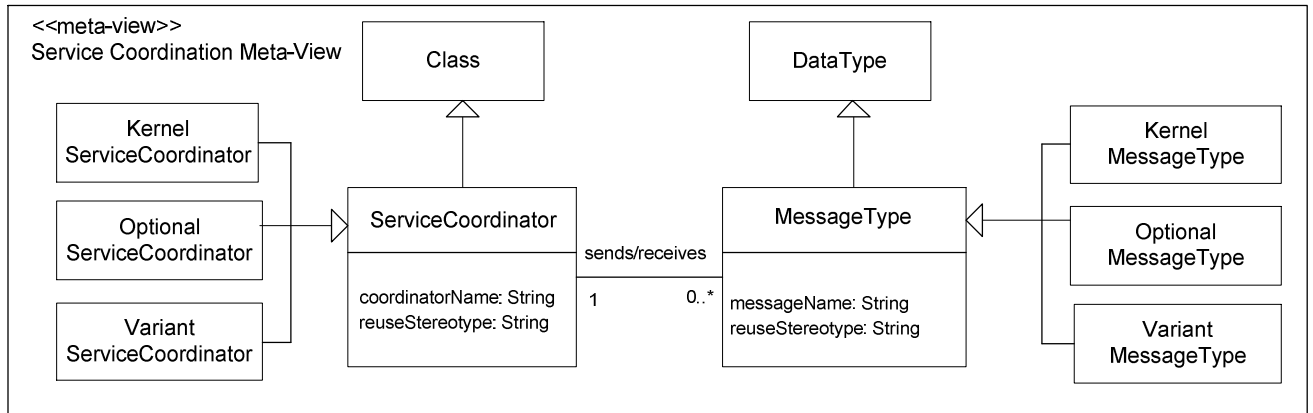


Fig. 5.4 Service Coordination Variability Meta-View

5.6. Feature Meta-View

Features (Fig. 5.5) are modeled by UML meta-classes that extend UML Class meta-class. Features are categorized into kernel, optional, alternative, and default depending on the characteristic of the reusable requirements as described in section 2. Feature groups (Fig. 5.5) are modeled by UML meta-classes as well. Feature groups refer to constraints on the selection of a group of features (e.g., preventing selection of mutually exclusive features).

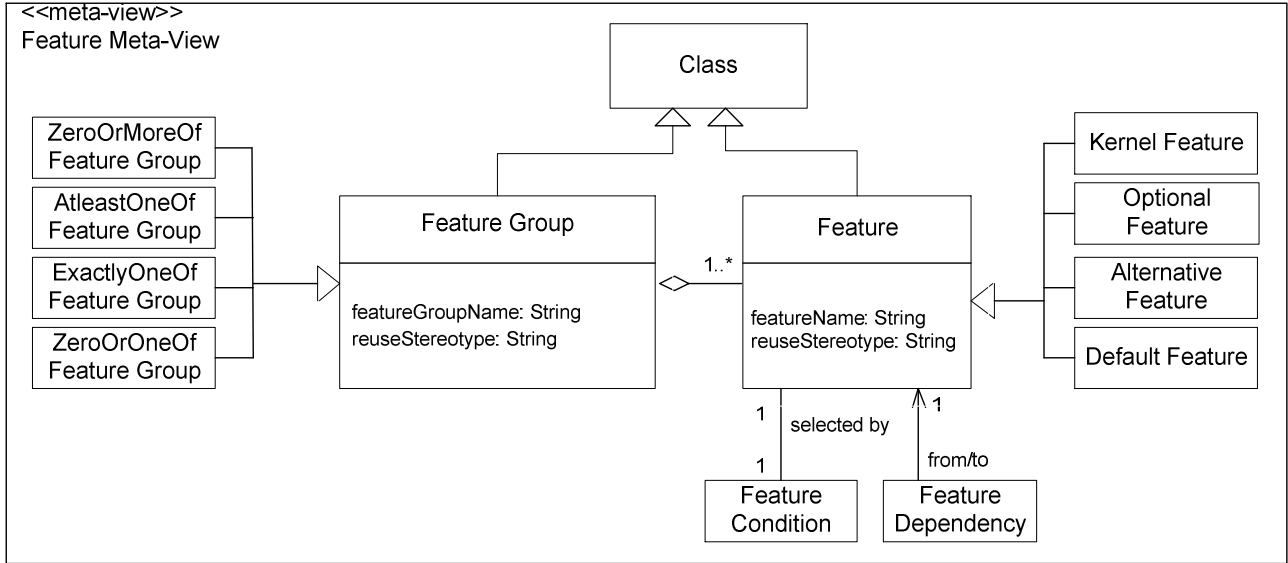


Fig. 5.5 Feature Meta-View

Feature groups are categorized into *ZeroOrMoreOf*, *AtleastOneOf*, *ExactlyOneOf*, and *ZeroOrOneOf*. Feature dependencies represent relationships between features and are modeled as UML meta-classes. Finally, Feature Conditions are Boolean constraints that determine the selection or de-selection of features in the SPL.

5.7. Service Variability Meta-Model Relationships

This section describes the relationships of the unified service variability meta-model (Fig. 5.6) that ties all the aforementioned views together. The meta-model consists of 5 meta-views (4+1 feature view) that correspond to each view in the multiple-view model (Chapter 4). The Feature Meta-View (Fig. 5.5) unifies the service views as explained in Chapter 4.

It should be noted that these relationships are enforced by the associations and dependencies of the multiple-view service variability meta-model. OCL rules are

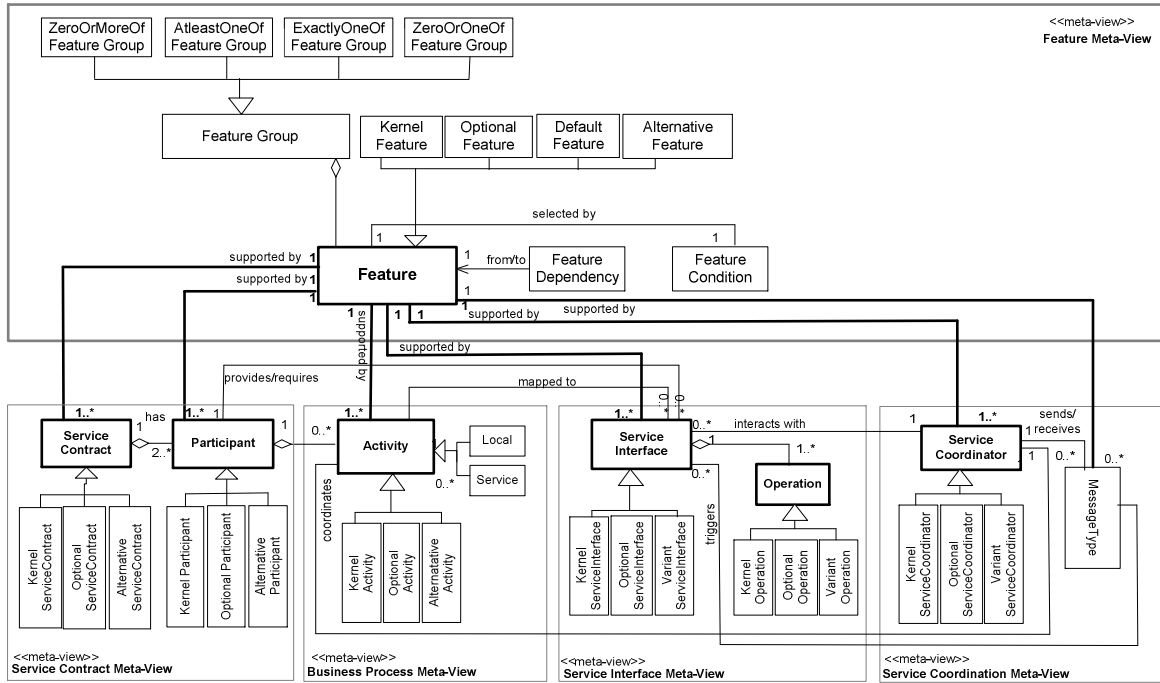


Fig. 5.6 Service Variability Meta-Model

provided as shown below for the relationships that cannot be explicitly described in the meta-model.

5.8. Intra Meta-View Relationships

The associations and dependencies inside each view are described (Fig. 5.6) along with OCL, which are used when additional constraints are needed for the relationships.

- A ServiceContract meta-class, in the Service Contract Meta-View, is associated with two or more Participants.

A kernel ServiceContract must relate to at least 2 kernel Participants

```
context servicecontract inv: reuseStereotype = 'kernel' implies
(select participant.reuseStereotype = 'kernel')->size() >= 2
```

- ServiceContract meta-classes may contain other ServiceContracts meta-classes to form composite contracts [54].
- The ServiceCoordinator meta-class in the Service Coordination view is associated with a MessageType meta-class as it sends/receives messages to/from services.

5.9. Inter Meta-View Relationships

The associations and dependencies between the multiple views of the multiple-view service variability meta-model are defined (Fig. 5.6).

- A service-oriented SPL has one or more ServiceContract meta-classes. In essence, ServiceContract meta-classes form the basis of any service-oriented SPL.
- **Participant** meta-classes, in the Service Contract Meta-View, provide or require **ServiceInterface** meta-classes in the Service Interface Meta-View.

A Participant must provide or require at least one ServiceInterface

```
context participant inv: reuseStereotype = 'kernel' implies
serviceinterface->exists(si | si.reuseStereotype = 'kernel')
```

- **Participant** meta-classes can be associated with **Activity** meta-classes in the Business Process Meta-View. This relationship defines a meta-business process, comprised of **Activity** meta-classes, for the **Participant** meta-classes.

- **Activity** meta-classes, in the Business Process Meta-View, are mapped to **ServiceInterface** meta-classes in the Service Interface Meta-View. In essence, Activities are realized by service operations exposed through **ServiceInterfaces**.
- Each Business Process (comprised of **Activity** meta-classes), in the Business Process View, is associated with one unique **ServiceCoordinator** meta-class in the Service Coordination View. It should be noted that centralized coordination is assumed.
- **ServiceCoordinator** meta-classes, in the Service Coordination Meta-View, interact with **ServiceInterface** meta-classes in the Service Interface Meta-View.
- **MessageType** meta-classes, in the Service Coordination Meta-View, trigger **Operation** meta-classes invocations on the **ServiceInterface** meta-classes, in the Service Interface Meta-View.

The following sections describe constraints on relationships between the Feature Meta-View and the Service views. Consistency checking rules that add explicit constraints on relationships between meta-classes are determined. Consistency checking rules are specified in English and OCL.

5.10. Feature to Service Contract Meta-View Relationships

- A **Feature** meta-class, in the Feature Meta-View, is related to one or more **ServiceContract** meta-classes in the Service Contract View. The variability stereotype on a **ServiceContract** dictates the type of **Feature** it may support. For instance, an optional feature can only be supported by optional service contracts.

Similarly, an alternative feature can only be supported by alternative service contracts.

A Kernel ServiceContract can only support a kernel Feature

```
context Feature inv: reuseStereotype = 'kernel' implies
servicecontract->size() >= 1 and servicecontract.reuseStereotype
= 'kernel'
```

- A **Feature** meta-class is related to one or more **Participant** meta-classes. The variability stereotype on a **Participant** dictates the type of **Feature** it may support.

An optional Participant can only support an optional Feature

```
context Feature inv: reuseStereotype = 'optional' implies
participant->size() >=1 and participant.reuseStereotype
= 'optional'
```

5.11. Feature to Business Process Meta-View

Relationships

- A **Feature** meta-class is related to one or more **Activity** meta-classes in the Business Process View. The variability stereotype on an **Activity** dictates the type of **Feature** it may support. A **Feature** could be supported by an entire business process, comprised of **Activity** meta-classes, in the Business Process Meta-View.

An optional Activity can only support an optional Feature

```
context Feature inv: reuseStereotype = 'optional' implies
activity->size() >=1 and activity.reuseStereotype = 'optional'
```

5.12. Feature to Service Interface Meta-View Relationships

- A **Feature** meta-class is related to one or more **ServiceInterface** meta-classes in the Service Interface Meta-View. The variability stereotype on a **ServiceInterface** dictates the type of **Feature** it may support.

A variant ServiceInterface can only support an alternative Feature
context Feature inv: reuseStereotype = 'alternative' implies
serviceinterface->size() >= 1 and
serviceinterface.reuseStereotype = 'variant'

- A **Feature** meta-class is related to one or more **Operation** meta-classes in the Service Interface Meta-View. The variability stereotype on an **Operation** dictates the type of **Feature** it may support.

An optional Operation can only support an optional Feature
context Feature inv: reuseStereotype = 'optional' implies
operation->size() >= 1 and operation.reuseStereotype = 'optional'

It should be noted that the stereotype 'variant' is used for architectural meta-classes, while the stereotype 'alternative' is used for requirements meta-classes [3].

5.13. Feature to Service Coordination Meta-View Relationships

- A **Feature** meta-class is related to one or more **ServiceCoordinator** meta-classes in the Service Coordination View. It should be noted that each business process, comprised of **Activity** meta-classes, is associated with a

unique **ServiceCoordinator**. The variability stereotype on a

ServiceCoordinator dictates the type of **Feature** it may support.

A kernel ServiceCoordinator can only support a kernel Feature

```
context Feature inv: reuseStereotype = 'kernel' implies  
servicecoordinator->size() >= 1 and  
servicecoordinator.reuseStereotype = 'kernel'
```

- A **Feature** meta-class is related to one or more **MessageType** meta-classes in

the Service Coordination Meta-View. The variability stereotype on a

MessageType dictates the type of **Feature** it may support.

An alternative MessageType can only support an alternative Feature

```
context Feature inv: reuseStereotype = 'alternative' implies  
messagetype->size() >= 1 and messagetype.reuseStereotype =  
'alternative'
```

6. Model-Driven Service-Oriented Product Line Engineering Framework

6.1. Introduction

This research created an automated framework that realizes the multiple-view service variability approach. To that end, this research exploits the model driven architecture (MDA) principles to achieve the following goals:

- Development of model-driven techniques to design service-oriented SPLs, since MDA research has focused on the design of *single* object-oriented applications.
- Automation of service-oriented product line engineering – this includes SPL Domain and Application Engineering, i.e. Requirements, Analysis, Design, Member Application Derivation, and Deployment (Chapter 2).
- Development of model-driven techniques to handle the variability of SOA middleware platforms which are usually hosted by the Enterprise Service Bus (ESB).
- Development of a model-driven service-oriented tool prototype that realizes the aforementioned framework.

As explained in Chapter 2, MDA treats software models as first-class entities in the software development process. In other words, programmers develop requirements and design models first without worrying about coding and platform concerns. These models are referred to as Platform-Independent Models (PIM). Once the PIMs are verified to match business requirements, they get *transformed* into more refined models geared towards specific technology platforms. These models are termed Platform-Specific Models (PSM). Finally, code is *generated* from the PSMs for a targeted technology platform such as Java, or .NET. Future software maintenance is performed on the PIMs and code is re-generated from them. This way, models and code are always in sync and developers can concentrate on business logic instead of platform and technology concerns. In addition, the PSMs can be reused if the target platform is selected again. The rest of the chapter describes the details of a model-driven service-oriented SPL framework that was created by this research. The chapter illustrates how MDA concepts are applied to service-oriented SPL engineering. More importantly, the chapter details how traditional MDA concepts are adapted to cater for service-oriented SPL engineering. Finally, a proof-of-concept tool prototype (SoaSPLE) that realizes the automated framework is described.

6.2. Platform Independent Model (PIM)

In typical MDA approaches [52], [51],[45] only one PIM is constructed for the entire application. Since SPLs are family of applications, this research proposes the construction of two types of PIMs:

1. A software product line PIM – this PIM models the entire service-oriented software family (SPL) 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 Chapter 4. Fig. 6.1 depicts a multiple-view PIM for an E-Commerce SPL. In this research, this model is termed splPIM.
2. Software member application PIMs – these PIMs model the derived member applications of the service-oriented SPL. In this research, each model is termed memberPIM. Fig. 6.2 depicts an example of a derived member application of the aforementioned multiple-view splPIM (Fig. 6.1). Fig. 6.2c shows a feature selection, a.k.a. feature configuration, of a memberSPL. Fig. 6.2a,b,d,e show the derived service variability views that supports the feature selection. Derivation of the memberPIM from the splPIM is described later in this chapter.

Fig. 6.3 depicts the process of creating two types of PIMs in SoaSPLE.

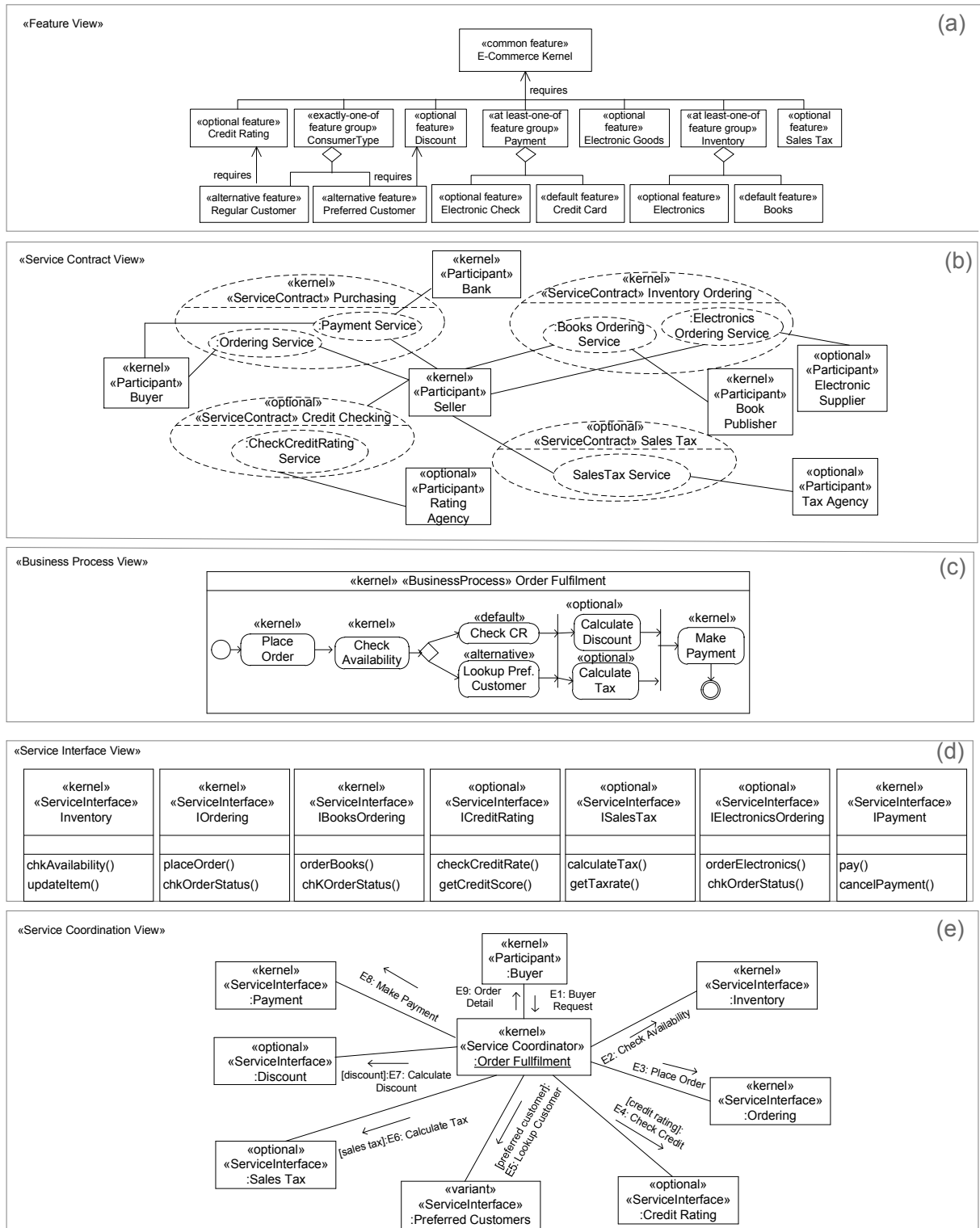


Fig. 6.1 E-Commerce SPL Service Oriented Multiple-View Platform Independent Model

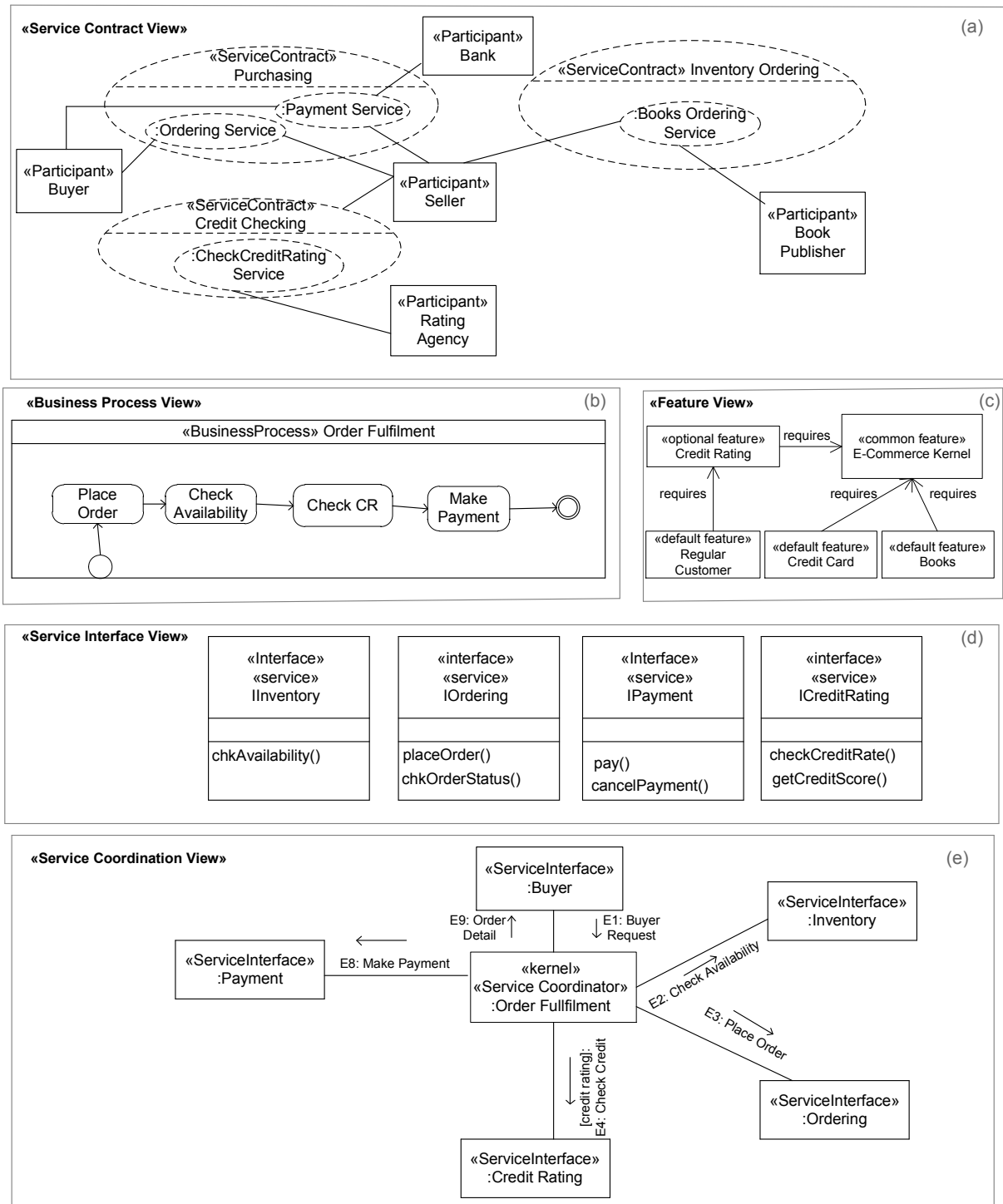


Fig. 6.2 Basic E-Commerce SPL Member Application

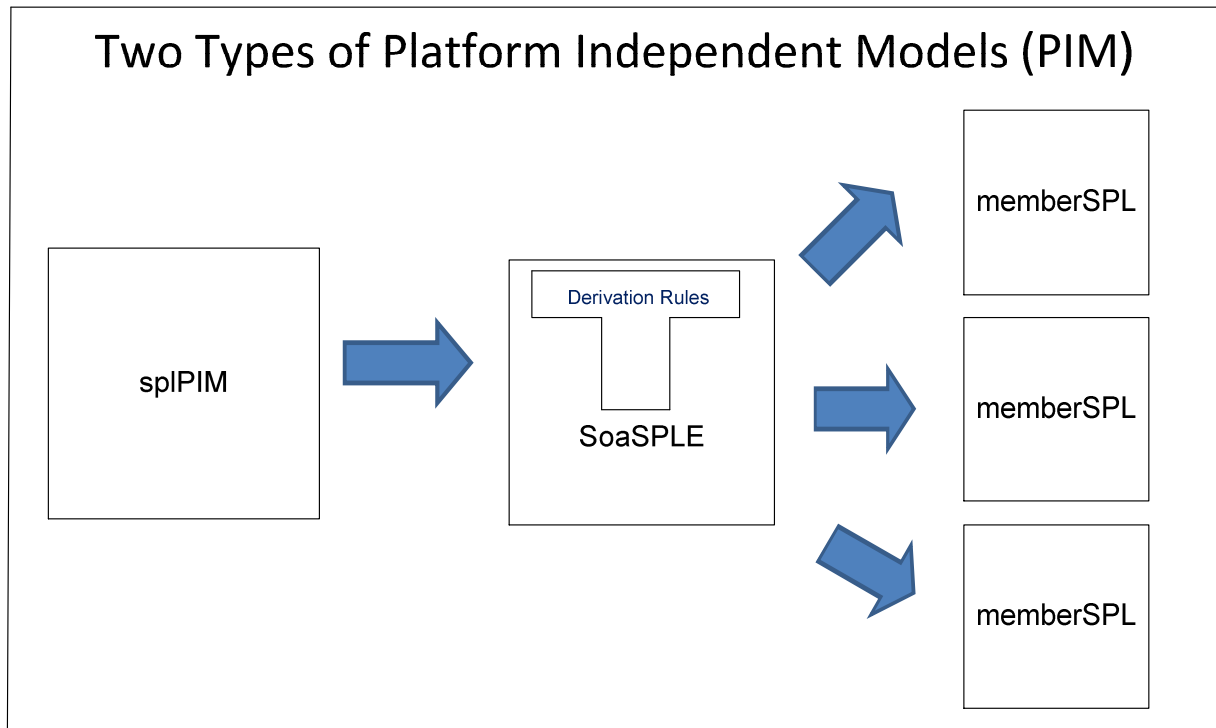


Fig. 6.3 Two Types of PIMs in SoaSPLE

6.3. Platform Specific Model (PSM)

This research does not provide transformation rules for transforming memberPIMs to PSMs, since the MDA literature already includes these rules. For example, transformation rules that transform PIMs, constructed in UML, to different platforms such as Java, .NET, and SQL, already exist [52], [51], [45].

However, this research can utilize any PIM to PSM transformation rules based on the desired target platform. In this research, this capability is realized by designing a transformation facility in the automated service-oriented SPL framework. Based on the desired target platform, a transformation rules definition can be imported into the

transformation facility. Consequently, this definition is used to transform memberPIMs to the target platform's PSM.

6.4. Feature Based Service Application Derivation

Member application derivation is based on feature selection from the feature model. 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 in Chapters 4 and 5.

To automatically derive member applications, this research defines a member application derivation algorithm that derives memberPIMs from the splPIM. The derivation algorithm traverses the splPIM, based on selected features, and constructs the memberPIMs from the SOA elements that are mapped to the selected features.

The derivation algorithm is implemented in JAVA code that takes splPIM and feature selection as inputs and produces memberPIMs. Fig. 6.4 depicts the major building blocks of the derivation process in the automated framework (SoaSPLE).

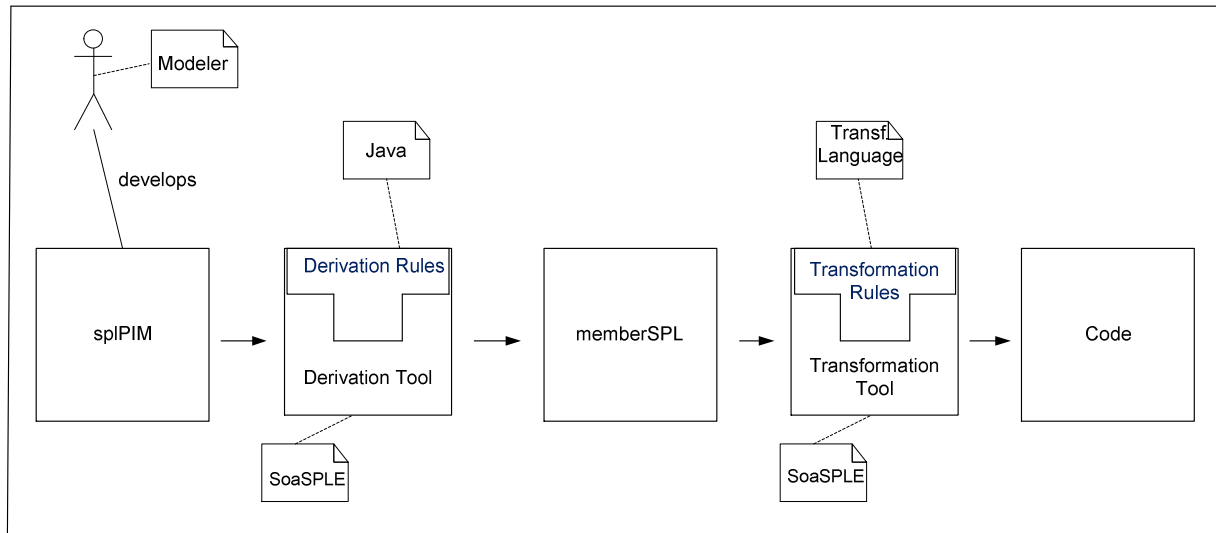


Fig. 6.4 splPIM to memberPIM Derivation Process in SoaSPLE

6.5. splPIM to memberPIM Derivation Algorithm

The member application derivation algorithm traverses the Feature-to-SOA mapping model and constructs multiple-view service-oriented member applications based on selected features of member applications.

The algorithm inputs are the Feature-to-SOA mapping model and the selected features for the specific member application. 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 ‘supBy’ 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.

6.6. Service-Oriented SPL Engineering Tool Prototype

To realize the automated framework described in this chapter, this research built a model-driven Service-Oriented SPL Engineering proof-of-concept tool prototype (SoaSPLE).

The goals of the prototype are as follows:

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

This research designed and implemented a tool prototype by utilizing the Eclipse Modeling Framework (EMF) [64]. EMF is an open-source meta-modeling framework and code generation facility for building tools and languages based on a structured meta-model called Ecore. Ecore is the meta model, which is the basis of the meta-modeling language provided by EMF. Ecore is equivalent to the Meta-Object Facility (MOF) provided by the OMG for UML.

EMF allows modelers to create meta-models in three different ways: direct diagramming with Ecore modeling elements, writing Java classes that define the meta-model, and writing XMI schema that defines the meta-model. Once a meta-model is constructed in

one format, EMF automatically creates the definition of the meta-model in the other formats. Further, when the meta-model is modified in any format, EMF automatically updates the other formats. In this research, meta-models were created with direct construction of Ecore meta-modeling elements. It should be noted that EMF provides runtime support where instances of the meta-models, i.e. models, can be manipulated by programs.

This research utilizes EMF as a design environment. In other words, the Ecore meta-modeling language is used to create all model elements in this research as first-class modeling elements. By representing the modeling elements of the meta-views as first-class elements, high flexibility is gained, which facilitates precise modeling and consequently helps achieve the goals mentioned above. The following subsections detail the design steps of SoaSPLE.

6.7. Feature Meta-View

This research modeled the Feature Meta-View (Chapter 5) as an Ecore meta-model. Fig. 6.5 depicts a snapshot of the Feature View meta-model in SoaSPLE.

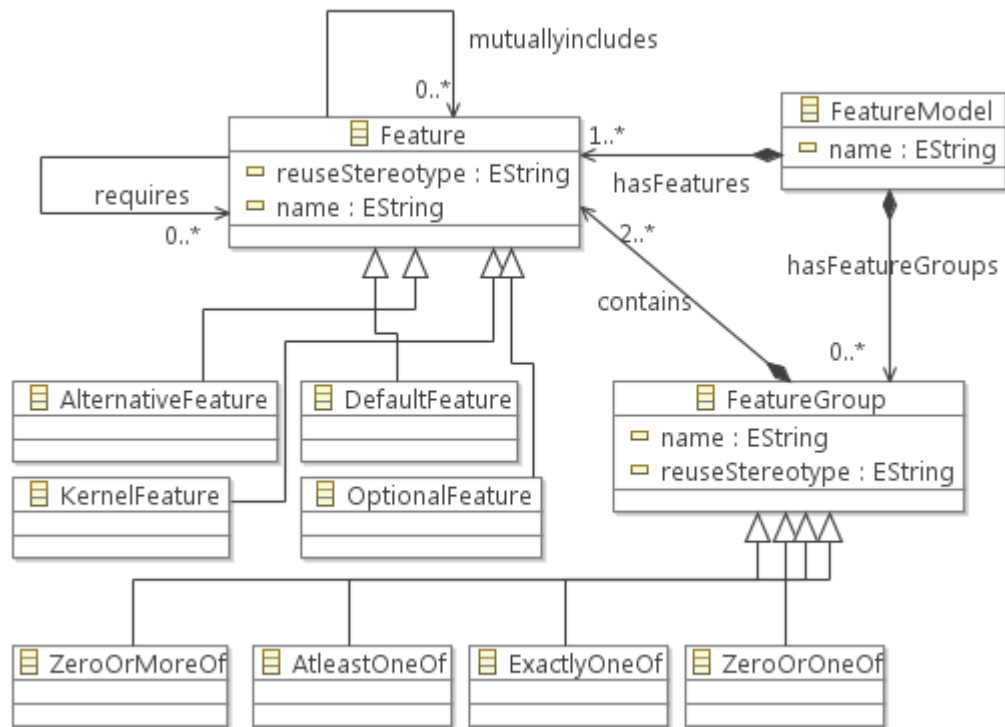


Fig. 6.5 Feature View Meta-Model in SoaSPLE

6.8. Service Meta-Views

This research models the Service Variability Meta-Views (Chapter 5) by using the Ecore meta-model as well. Fig. 6.6 depicts the Service Contract Meta-View in SoaSPLE.

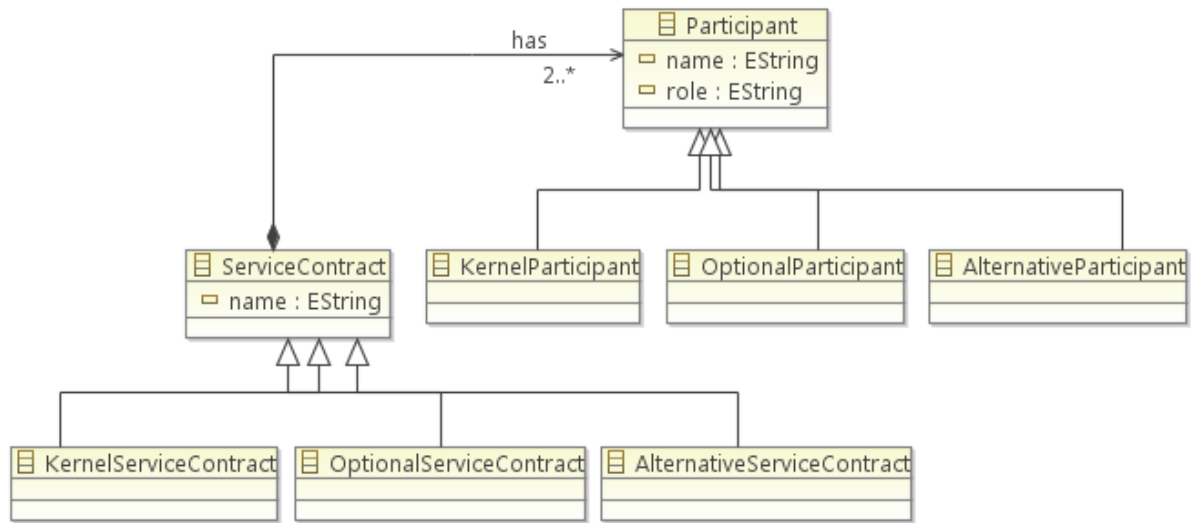


Fig. 6.6 Service Contract View Meta-Model in SoaSPLE

6.9. Feature Meta-View to Variable Service Meta-Views Mapping

This research constructed another Ecore meta-model that establishes the mapping relationships among the Service Meta-Views and the Feature Meta-View. This meta-model is a representation of the Service Variability Meta-Model described in Chapter 5. 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 represent the multiple-view service variability meta-model described in Chapter 5. Fig. 6.7 depicts this meta-model in SoaSPLE.

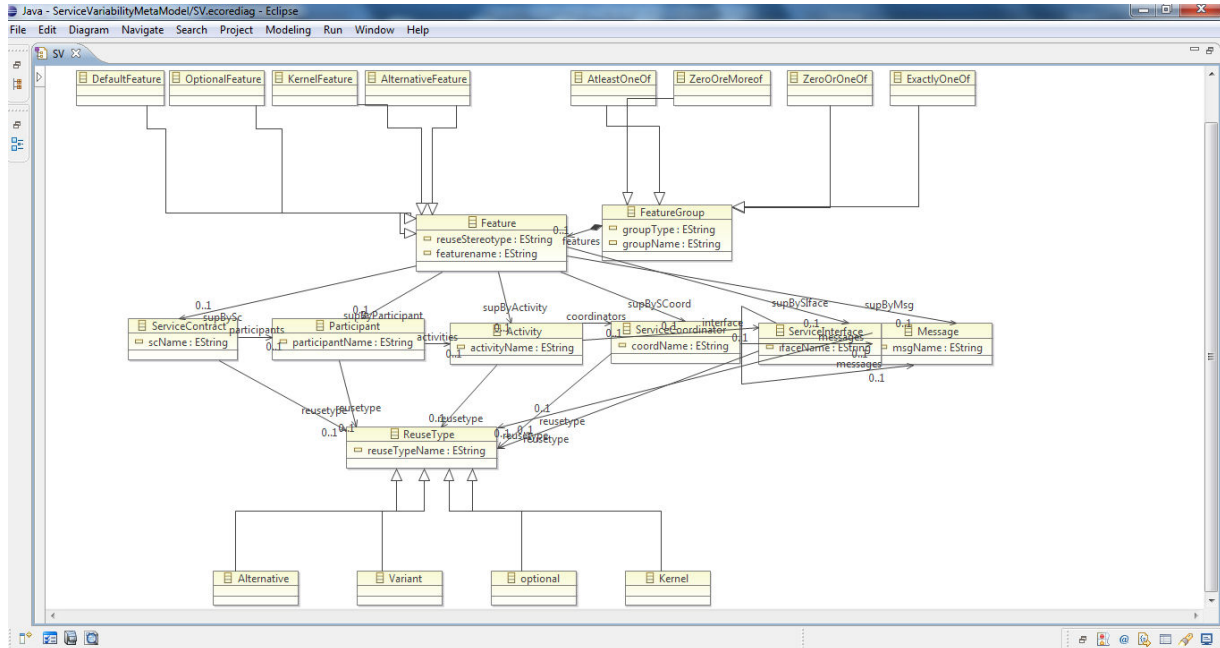


Fig. 6.7 Multiple-View Service Variability View Meta-Model in EMF

This research constructed a Mapping Facility within the tool prototype that enables modelers to map features to service views. This Mapping Facility is governed by the multiple-view service variability meta-model described in Chapter 5 and shown in Fig. 6.7. The use of the Mapping Facility is demonstrated in the case studies of Chapters 8 and 9.

6.10. Consistency Checking Rules

This research applied the consistency checking rules (Chapter 5) to the multiple-view service variability meta-model. The OCL consistency checking rules were added to the multiple-view service variability meta-model as Ecore annotations [64]. SoaSPLE

executes the consistency rules when service models, based on the multiple-view service variability meta-model (Chapter 5), are created. If the service models violate these consistency checking rules, SoaSPLE emits popup messages indicating the violation. Validation of the consistency checking rules is described in Chapter 7, 8, and 9.

For example, to implement the following OCL rule:

A Kernel ServiceContract can only support a kernel Feature
context Feature inv: reuseStereotype = 'kernel' implies
servicecontract->size() >= 1 and servicecontract.reuseStereotype =
'kernel'

The Feature meta-class in the Feature Meta-View model in the tool prototype was annotated with the OCL rule as shown in Fig. 6.8. When the meta-model is compiled in the tool, SoaSPLE generates Java code that represents the aforementioned OCL rule.

When models are created based on the meta-model, SoaSPLE automatically evaluates the

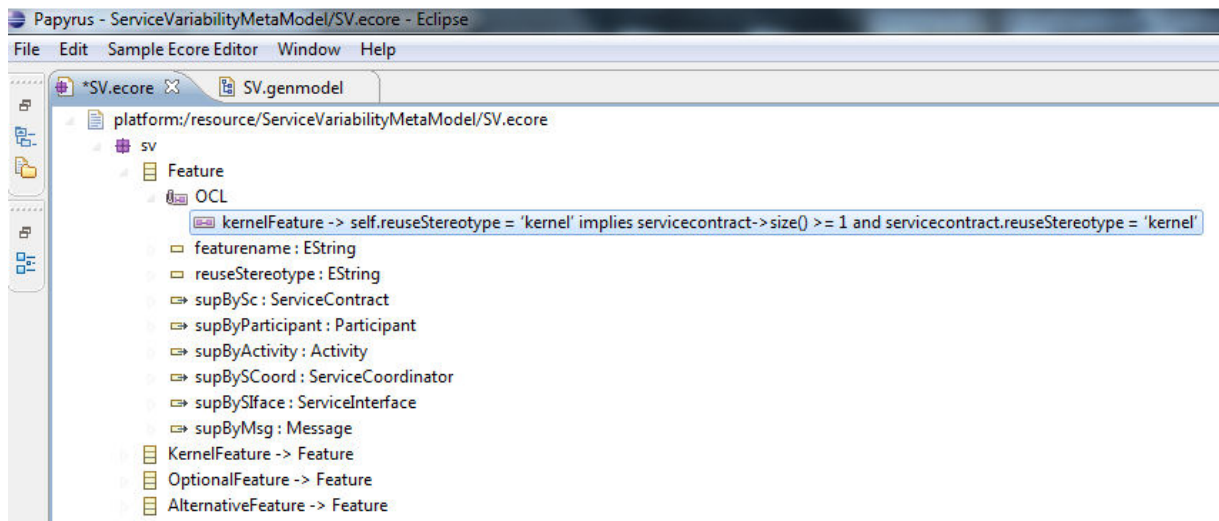


Fig. 6.8 Meta-Class OCL Annotation Example

embedded OCL rules using the underlying Java code.

6.11. Service Member Application Derivation

This research derives service member applications by applying a derivation algorithm as described in section 6.6. The derivation algorithm is implemented in Java and defined in SoaSPLE. SoaSPLE automatically generates Java classes and interfaces for all meta-classes in the meta-model. In other words, each meta-class in the multiple-view service variability meta-model is represented internally in SoaSPLE as a Java class. SoaSPLE's derivation algorithm gets executed against the generated meta-classes' Java code. The Derivation component within SoaSPLE takes the splPIM and feature selection as inputs and produces the specific memberPIM as an output as explained in section 6.5.

6.12. Code Generation

As mentioned in section 6.3 , this research does not provide rules for transforming memberPIMs into memberPSMs, since the MDA literature already includes these rules. This research designed SoaSPLE in such a way to use any PIM-to-PSM transformation rules based on the desired target platform.

SoaSPLE uses a Java/Web Services PIM-to-PSM transformation rules to derive the memberPSMs. If a different target language is desired, say .NET instead of Java, users of SoaSPLE can use a .NET PIM-to-PSM transformation rules and regenerate the

memberPSMs. It should be noted that the alternate transformation rules are applied to the original memberPIMs.

6.13. Deployment and Execution

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

- Eclipse runtime support environment [64].
- Apache ODE [65] – 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 [66] – 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 [67] – Swordfish is an open-source extensible Enterprise Service Bus (ESB).

The case studies in Chapter 8 and 9 demonstrate the implementation, deployment, and execution of member applications by using SoaSPLE.

7. Research Validation

7.1. Validation Approach

The objective of the validation is to evaluate the approach in this research with regard to the following properties:

1. The multiple views of the service-oriented software product line are consistent with each other. More specifically, the evaluation validates that the following multiple-view modeling relationships, described in Chapter 4 and 5, are satisfied:
 - a) Intra-View Relationships, i.e. the relationships between meta-classes within each meta-view.
 - b) Inter-View Relationships, i.e. the relationships between meta-classes across the meta-views.
 - c) Feature to Service Views Relationships (Mapping Features to SOA), i.e., feature meta-classes are supported by valid meta-classes in the multiple views according to the mapping rules described in Chapter 5.

This property is referred to as *Consistency* throughout the chapter.

2. The multiple-view service variability model is compliant with the underlying multiple-view service variability meta-model. In other words, each element in the

multiple-view model is a valid instantiation of its corresponding meta-class in the meta-model. In addition, each association in the multiple-view model is a valid instantiation of its corresponding meta-association in the meta-model. This property is referred to as *Compliance* throughout the chapter.

3. Derived software product line member applications are valid service-oriented SPL members. Hence, the evaluation validates the following two characteristics of service member derivation:

- a) The derived member applications are valid members of the SPL family.
This implies that member applications are *only* derived if they adhere to a *valid* feature selection from the feature model which describes the requirements of the entire SPL.
- b) The derived SPL member applications are valid *service-oriented* applications, i.e. they execute within a service-oriented environment and their constituent components can be invoked as the business workflow of the application is exercised.

This property is referred to as *Legitimacy* throughout the chapter.

To achieve the aforementioned validation objectives, the validation procedure is divided into two main testing tasks:

1. Unit Testing – this type of testing tests each element and relationship in the multiple-view service variability meta-model. Unit testing is needed, because the case studies may not exercise every part of the meta-model.

2. System Testing – this is a system-wide testing that tests the running service-oriented applications of the SPL end-to-end.

This chapter details unit testing, while Chapters 8 and 9 details system testing through the case studies.

7.2. System Testing Approach

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 the member application through testing the sequence of service coordinators and service invocations. The testing results are demonstrated by having an execution trace that validates that the activity model sequence, i.e. the business process, is executed by the service coordinator and service operation invocations.

System testing validates the *Legitimacy* property. System testing is accomplished by conducting case studies using the proof-of-concept tool prototype, SoaSPLE. The case studies involve 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. Two case studies are conducted: an electronic commerce service-oriented SPL and a hotel reservation service-oriented SPL. The case studies are detailed in Chapters 8 and 9.

7.3. Unit Testing

Unit testing validates the *Consistency* and *Compliance* properties by demonstrating that each element in the multiple-view service variability meta-model can be instantiated as a modeling element. In addition, unit testing 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. The following types of unit tests are carried out:

- a. Test cases to validate the consistency checking rules of each type of meta-model relationships, i.e. intra-view, inter-view, and feature-to-SOA mapping. *This type of unit tests validates the Consistency property.*
- b. Test cases to create a multiple-view service model, which results in the creation of model instances of each meta-class and model associations of each meta-association. *This type of unit tests validates Compliance property.*

7.4. Unit Test Cases

Many unit tests are expressed via the JUnit Testing Framework. JUnit is an open-source unit testing framework that allows Java developers to write repeatable unit tests for their code. In this research, JUnit tests were designed to operate on Java code that represents the meta-modeling elements of the multiple-view service variability meta-model. This is

possible, because Eclipse Modeling Framework (Chapter 6) creates Java interfaces for all meta-modeling elements. Hence, the JUnit tests in this research programmatically create modeling elements and associations of the multiple-view meta-model and then run pertinent tests.

Other unit tests are performed by directly engaging SoaSPLE to test for specific scenarios. Unit tests are detailed in the remainder of this chapter per meta-view of the multiple-view service variability meta-mode

7.4.1.Feature Meta-View Unit Tests

Table 7.1 Feature Meta-View Unit Tests

Unit Test	Meta-Model Element	Model Element	Expected Result	Pass/Fail	JUnit/Manual
Create kernel Feature	KernelFeature	Purchasing Kernel Feature	Pass	Pass	JUnit
Create Optional Feature	OptionalFeature	Credit Rating Optional Feature	Pass	Pass	JUnit
Create Alternative Feature	AlternativeFeature	Preferred Customer Alternative Feature	Pass	Pass	JUnit
Create ZeroOrMoreOf Feature Group	ZeroOrMoreOf	Consumer Type ExactlyOneOf Feature Group	Pass	Pass	JUnit
Create AtleastOneOf Feature Group	AtleastOneOf	Payment AtleastOneOf Feature Group	Pass	Pass	JUnit
Create ExactlyOneOf Feature Group	ExactlyOneOf	Consumer Type ExactlyOneOf Feature Group	Pass	Pass	JUnit
Create ZeroOrOneOf Feature Group	ZeroOrOneOf	Dummy ZeroOrOneOf Feature Group	Pass	Pass	JUnit
Select two mutually exclusive features	Feature FeatureGroup	Regular Customer Feature Preferred Customer Features	Fail	Fail	Manual
Select an optional feature without a feature it depends on	Feature OptionalFeature	Electronic Goods Optional Feature E-Commerce Kernel Feature	Fail	Fail	Manual
Select each feature in ExactlyOneOf Feature Group on its own	Feature FeatureGroup	Regular Customer Feature Preferred Customer Features Consumer Type ExactlyOneOf Feature Group	Pass	Pass	Manual
Select each feature in ZeroOrOneOf Feature Group on its own	Feature FeatureGroup	Dummy Feature Dummy ZeroOrOneOf Feature Group	Pass	Pass	Manual
Do Not select a feature from ExactlyOneOf Feature Group	Feature FeatureGroup	Regular Customer Feature Preferred Customer Features Consumer Type ExactlyOneOf Feature Group	Fail	Fail	Manual
Do Not select a feature from a AtleastOneOf Feature Group	Feature FeatureGroup	Books Feature Electronics Feature Inventory AtleastOneOf Feature Group	Fail	Fail	Manual
Select more than one feature from ExactlyOneOf Feature Group	Feature FeatureGroup	Regular Customer Feature Preferred Customer Features Consumer Type ExactlyOneOf	Fail	Fail	Manual
Select more than one feature from ZeroOrOneOf Feature Group	Feature FeatureGroup	Dummy Feature Dummy ZeroOrOneOf Feature Group	Fail	Fail	Manual
Select more than one feature from ZeroOrMoreOf Feature Group	Feature FeatureGroup	Dummy Feature Dummy ZeroOrMoreOf Feature Group	Pass	Pass	Manual

7.4.2. Service Contract Meta-View Unit Tests

Table 7.2 Service Contract Meta-View Unit Tests					
Unit Test	Meta-Model Element	Model Element	Expected Result	Pass/Fail	JUnit/Manual
Create Kernel Service Contract	KernelServiceContract	Purchasing Service Contract	Pass	Pass	JUnit
Create Optional Service Contract	OptionalServiceContract	Sales Tax Service Contract	Pass	Pass	JUnit
Create Alternative Service Contract	AlternativeServiceContract	Dummy Service Contract	Pass	Pass	JUnit
Create Kernel Participant	KernelParticipant	Buyer Kernel Participant	Pass	Pass	JUnit
Create Optional Participant	OptionalParticipant	Rating Agency Optional Participant	Pass	Pass	JUnit
Create Alternative Participant	AlternativeParticipant	Dummy Alternative Participant	Pass	Pass	JUnit
Create a ServiceContract with Zero Participants	KernelServiceContract	Purchasing Service Contract	Fail	Fail	Manual
Create a ServiceContract with One Participant	OptionalServiceContract	Credit Checking Service Contract	Fail	Fail	Manual

7.4.3. Business Process Meta-View Unit Tests

Table 7.3 Business Process Meta-View Unit Tests					
Unit Test	Meta-Model Element	Model Element	Expected Result	Pass/Fail	JUnit/Manual
Create a Kernel Activity	KernelActivity	Place Order	Pass	Pass	JUnit
Create an Optional Activity	OptionalActivity	Calculate Discount	Pass	Pass	JUnit
Create an Alternative Activity	AlternativeActivity	Lookup Preferred Customer	Pass	Pass	JUnit
Create a Start Node with no Next Activity	StartNode	Dummy Start Node	Fail	Fail	Manual

7.4.4. Service Interface Meta-View Unit Tests

Table 7.4 Service Interface Meta-View Unit Tests					
Unit Test	Meta-Model Element	Model Element	Expected Result	Pass/Fail	JUnit/Manual
Create a Kernel ServiceInterface	KernelServiceInterface	IOrdering	Pass	Pass	JUnit
Create an Optional ServiceInterface	OptionalServiceInterface	ICreditRating	Pass	Pass	JUnit
Create a Variant ServiceInterface	VariantServiceInterface	Dummy Iface	Pass	Pass	JUnit
Create a Kernel Operation	KernelOperation	placeOrder()	Pass	Pass	JUnit
Create an Optional Operation	OptionalOperation	orderElectronics()	Pass	Pass	JUnit
Create a Variant	VariantOperation	Dummy Operation	Pass	Pass	JUnit

7.4.5. Service Coordination Meta-View Unit Tests

Table 7.5 Service Coordination Meta-View Unit Tests					
Unit Test	Meta-Model Element	Model Element	Expected Result	Pass/Fail	JUnit/Manual
Create a Kernel ServiceCoordinator	KernelServiceCoordinator	Order Fulfillment	Pass	Pass	JUnit
Create an Optional ServiceCoordinator	OptionalServiceInterface	Residential Booking	Pass	Pass	JUnit
Create a Variant ServiceCoordinator	VariantServiceInterface	Dummy	Pass	Pass	JUnit
Create a ServiceCoordinator that sends Zero Messages	ServiceCoordinator	Dummy	Pass	Pass	Manual
Create a ServiceCoordinator that sends One or more	ServiceCoordinator	Dummy	Pass	Pass	Manual

7.4.6. Inter-View Relationships Unit Tests

Table 7.6 Inter-View Relationships Unit Tests

Unit Test	Meta-Model Element	Model Element	Expected Result	Pass/ Fail	JUnit/ Manual
Create a ServiceContract with No Participants	OptionalServiceContract	Timing Optional ServiceContract	Fail	Fail	Manual
Create a ServiceContract with One Participants	OptionalServiceContract OptionalParticipant	Timing Optional ServiceContract	Fail	Fail	Manual
Create a Participant that does not provide or consume a ServiceInterface	Participant ServiceInterface	Dummy	Fail	Fail	Manual
Create a Participant that does provide or consume a ServiceInterface	Participant ServiceInterface	Dummy	Pass	Pass	Manual
Create Business Process that is not associated with a Participant	Activity Participant	Dummy	Fail	Fail	Manual
Create an Activity that is not mapped to ServiceInterface	Activity ServiceInterface	Dummy	Fail	Fail	Manual
Create a Business Process that is not associated with a ServiceCoordinator	Activity ServiceCoordinator	Dummy	Fail	Fail	Manual

7.4.7.Consistency Checking Rules Unit Tests

Table 7.7 Consistency Checking Rules Unit Tests

Unit Test	Meta-Model Element	Model Element	Expected Result	Pass/ Fail	JUnit/ Manual
Map a Kernel Feature to a Kernel ServiceContract	KernelFeature KernelServiceContract	Pass	Pass		Manual
Map a Kernel Feature to Optional ServiceContract	KernelFeature OptionalServiceContract	Fail	Fail		Manual
Map a Kernel Feature to a Variant ServiceContract	KernelFeature VariantServiceContract	Fail	Fail		Manual
Map a Kernel Feature to Kernel Participant	KernelFeature KernelParticipant	Pass	Pass		Manual
Map a Kernel Feature to Optional Participant	KernelFeature OptionalParticipant	Fail	Fail		Manual
Map a Kernel Feature to an Alternative Participant	KernelFeature AlternativeParticipant	Fail	Fail		Manual
Map a Kernel Feature to a Kernel Activity	KernelFeature KernelActivity	Pass	Pass		Manual
Map a Kernel Feature to an Optional Activity	KernelFeature OptionalActivity	Fail	Fail		Manual
Map a Kernel Feature to a Alternative Activity	KernelFeature AlternativeActivity	Fail	Fail		Manual
Map a Kernel Feature to a Kernel ServiceInterface	KernelFeature KernelServiceInterface	Pass	Pass		Manual
Map a Kernel Feature to Optional ServiceInterface	KernelFeature OptionalServiceInterface	Fail	Fail		Manual
Map a Kernel Feature to Variant ServiceInterface	KernelFeature VariantServiceInterface	Fail	Fail		Manual
Map a Kernel Feature to Kernel ServiceCoordinator	KernelFeature KernelServiceCoordinator	Pass	Pass		Manual
Map a Kernel Feature to Optional ServiceCoordinator	KernelFeature OptionalServiceCoordinator	Fail	Fail		Manual
Map a Kernel Feature to Variant ServiceCoordinator	KernelFeature VariantServiceCoordinator	Fail	Fail		Manual

8. Electronic Commerce Service-Oriented Software Product Line Case Study

8.1. Case Study Objectives

The purpose of conducting this case study is to evaluate the approach in this research with regard to the validation properties detailed in Chapter 7. These properties are listed here briefly:

1. The multiple views of the service-oriented software product line are consistent with each other. This property is referred to as *Consistency*.
2. The multiple-view service variability model is compliant with the underlying multiple-view service variability meta-model. This property is referred to as *Compliance*
3. Derived software product line member applications are valid service-oriented SPL members. Hence, the evaluation validates the following two characteristics of service member derivation:
 - a. The derived member applications are valid members of the SPL family.
 - b. The derived SPL member applications are valid service-oriented applications.

This property is referred to as *Legitimacy*.

8.2. Case Study Validation Approach

The validation approach of the case study is a system testing approach that 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 tool 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.

8.3. Electronic-Commerce Case Study Problem

Description

This case study demonstrates the modeling of an E-Commerce web based SPL. The E-Commerce SPL consists of sellers, buyers, suppliers, banks, and tax and credit check agencies. Sellers get their goods from suppliers and sell them to interested buyers. Sellers define their own business workflow to check buyers' credit history, replenish goods' inventory, assess sales tax and fulfill buyers' orders among other things.

The E-Commerce domain has inherent variable requirements – different types of sellers, different types of sold goods, different payment methods, multiple suppliers, several sales tax jurisdictions, multiple pricing and discount schemes, and the like.

The following sections illustrate the steps taken to conduct the case study.

8.4. 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 represent the Feature View of the multiple-view service variability model (Chapter 4). After commonality and variability analysis, it was determined that the E-Commerce SPL can vary in the following ways:

- a) The E-Commerce SPL will always have Purchasing and Inventory capabilities. In addition, it will always have Sellers, Buyers, Book Publisher, and Bank roles. The Seller role will always have an Order Fulfillment business process. Finally, the E-Commerce SPL will always have Ordering, Payment, and Books Ordering capabilities. All this common functionality is grouped into the ‘E-Commerce Kernel’ <<common feature>> in the feature model, which includes the following features: ‘Purchasing’, ‘Order Fulfillment’, ‘Ordering’, ‘Payment’, ‘Inventory’, and ‘Books’.
- b) The E-Commerce product line can either have a credit checking option for regular customers and partners, which would then require a credit rating capability; however, if the business always deals with trusted buyers and partners, this

capability is not required. This requirement is modeled as a 'Credit Rating' <<optional feature>>.

- c) A Seller can decide to offer electronic goods for its customers. In turn, an electronic goods supplier would be needed for the Inventory feature. This requirement is modeled as 'Electronics' <<optional feature>>.
- d) The Order Fulfillment business process of the Seller role can offer two types of payments: credit card, which is default, and electronic check. This requirement is modeled as a 'Payment' <<at least-one-of feature group>> with a 'Credit Card', which is a default feature stereotyped with <<default feature>>, and an 'Electronic Check', which is an <<optional feature>> in the feature model.
- e) The Seller can offer a discount capability that can be selected seasonally. This requirement is modeled as a 'Discount' feature, which has the <<optional feature>> stereotype in the feature model.
- f) The Order Fulfillment business process of the Seller can offer a 'Preferred Customer' capability for customers with existing credit records to speed up order processing. This requirement is modeled as a 'Consumer Type' <<exactly-one-of feature group>> with a 'Preferred Customer' <<alternative feature>> and a 'Regular Customer' << alternative feature >> feature in the feature model. The 'Regular Customer' feature *requires* the 'Credit Rating' feature mentioned above. This dependency is modeled as a 'Requires' association in the feature model.
- g) Some states require their citizens to pay sales tax on internet-based purchases. Thus, Sellers need to include tax calculation capabilities in their Order Fulfillment

business process based on the Buyers' state of residence. This requirement is modeled as a 'Sales Tax' <<optional feature>> in the feature model.

Based on the aforementioned analysis, a feature model was created in SoaSPLE as shown in Fig. 8.1.

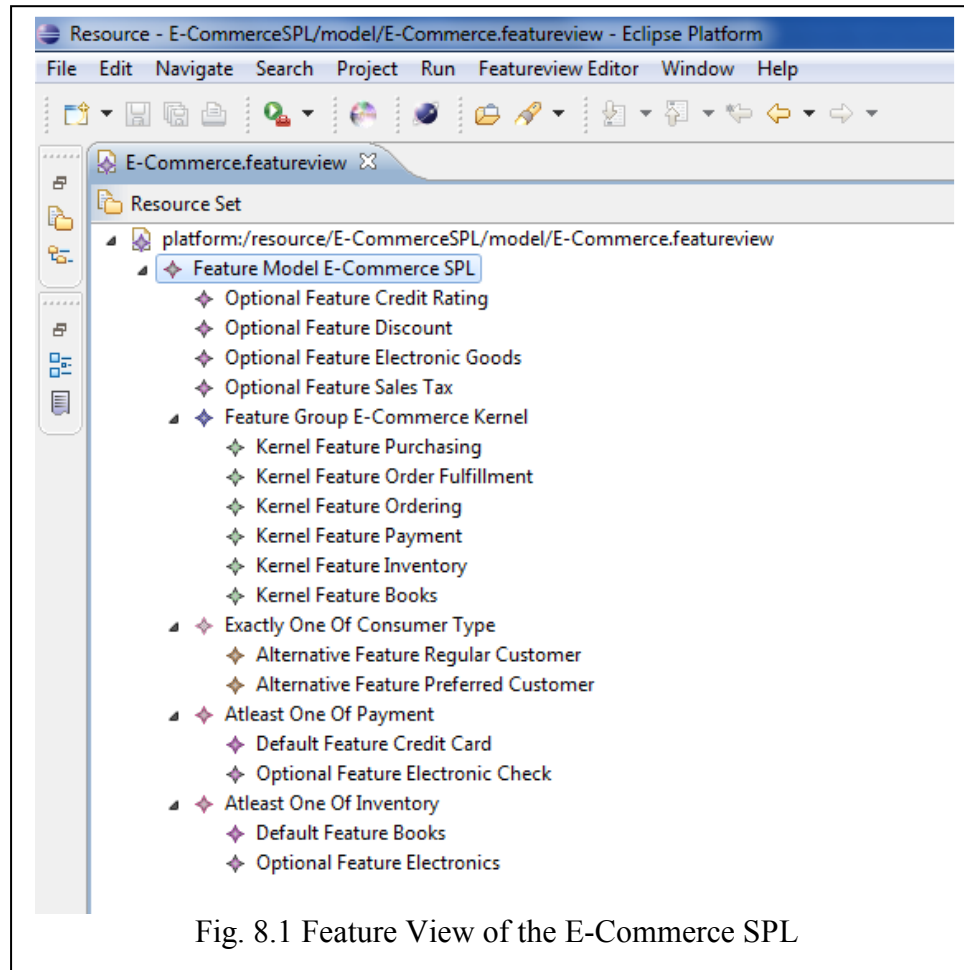


Fig. 8.1 Feature View of the E-Commerce SPL

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 (Chapter 5) 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.

8.5. Service Contract Variability View Modeling

This is a Requirements view that describes service contracts (Chapter 4). Based on the feature model described in the previous section and knowledge of the E-Commerce domain, it was decided to model ServiceContracts and Participants as shown in the Service Contract View in Fig. 8.2. The model in Fig. 8.2 consists of Purchasing, Inventory Ordering, Credit Checking, and Sales Tax ServiceContract classes. In addition, the model contains Buyer, Seller, Bank, Rating Agency, Book Publisher, Electronic Supplier, and Tax Agency service Participants. ServiceContracts 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 ServiceContracts to makeup an E-Commerce family of applications. It should be noted that this model is based on the Service Contract Variability Meta-Model described in Chapter 5.

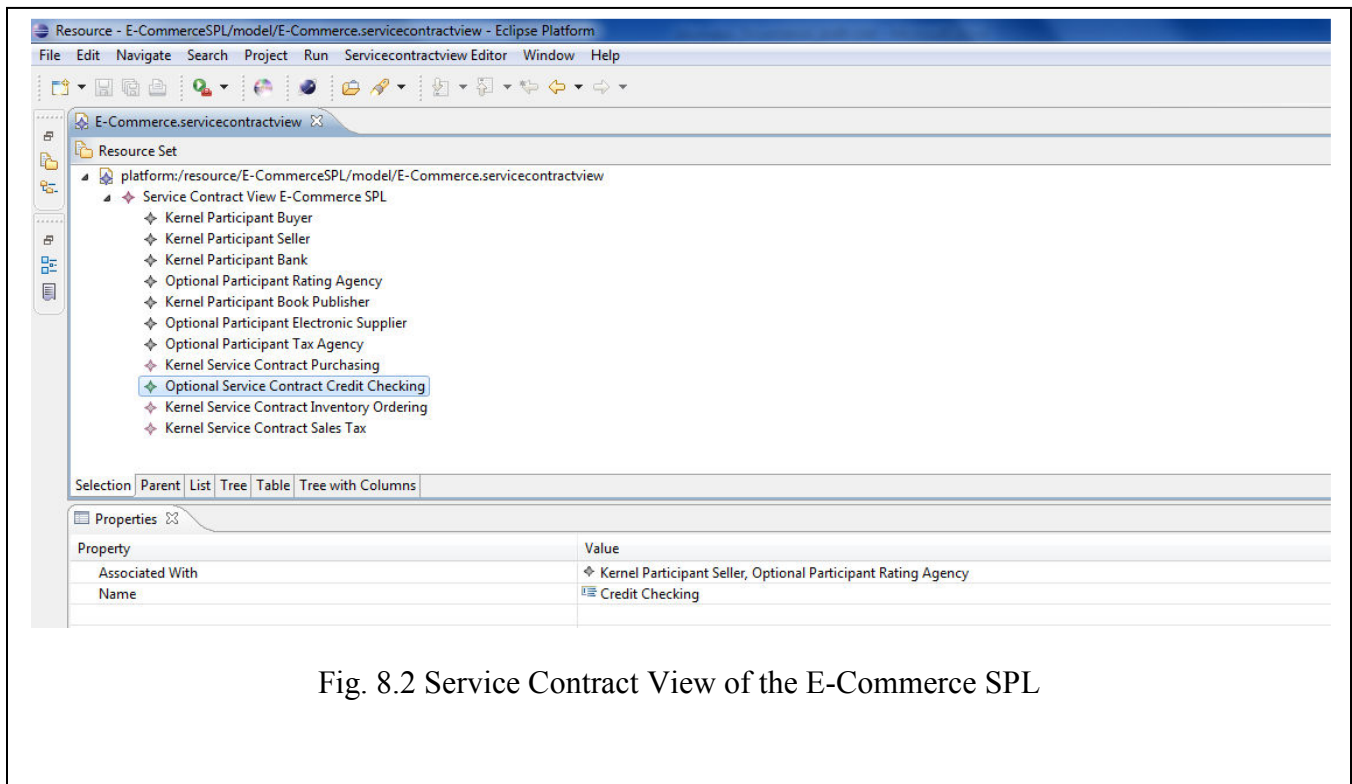
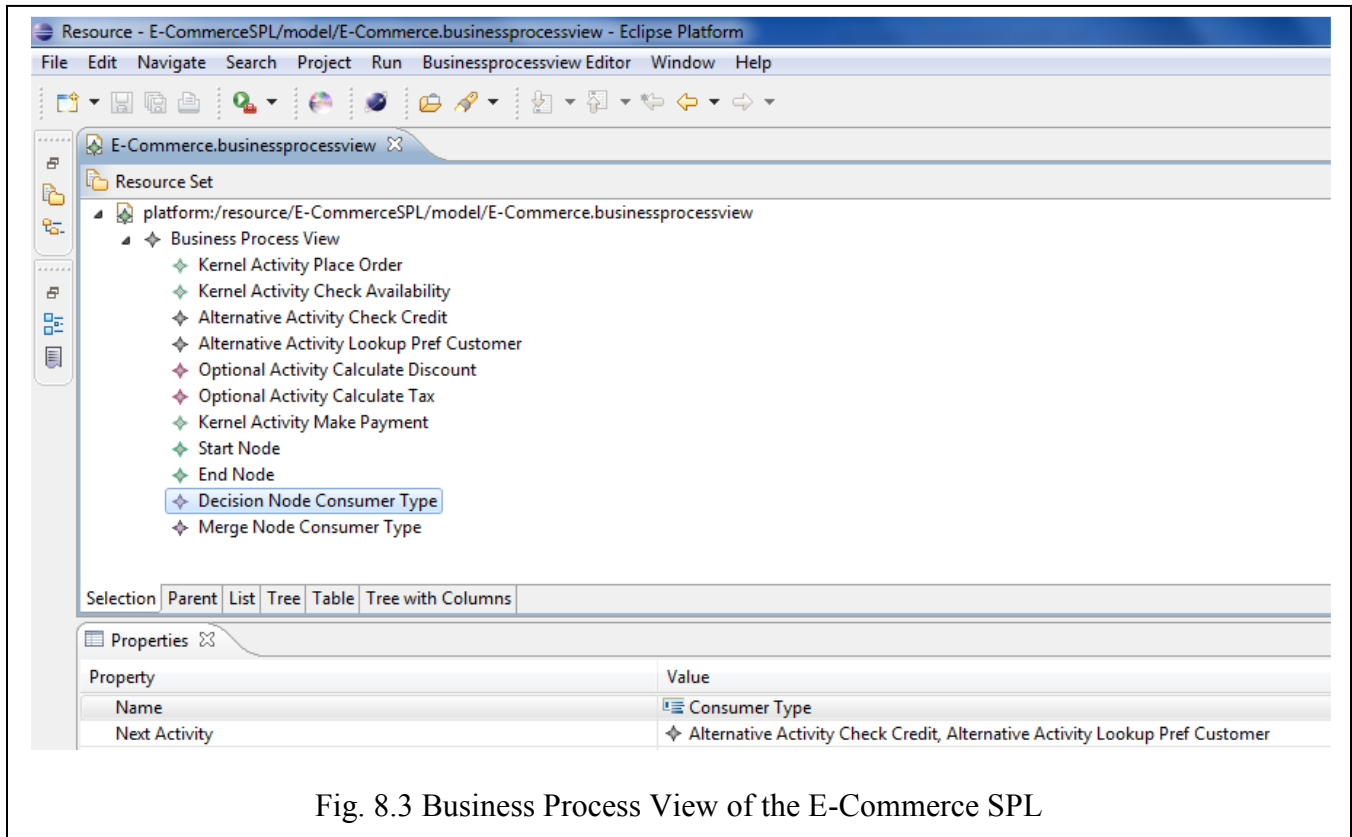


Fig. 8.2 Service Contract View of the E-Commerce SPL

8.6. Business Process Variability View Modeling

This Requirements 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 Meta-Model described in Chapter 5.

A UML Activity diagram is constructed with kernel, optional, and alternative activities as shown in Fig. 8.3.



8.7. Service Interface Variability View Modeling

This Architectural view models service interfaces, which specify the operations provided or required by Participants. Service interfaces are modeled by using the `ServiceInterface` element as described in Chapter 5. ServiceInterfaces are categorized as kernel, optional, or variant.

Upon examining the activities of the Order Fulfillment business process (section 8.6) and based on the E-Commerce domain knowledge, it was decided to have the following service interfaces (Fig. 8.4): `IOrdering`, `IInventory`, `IBooksOrdering`, `ICreditRating`, `ISalesTax`, `IElectronicsOrdering`, and `IPayment`. It should be noted that these service

interfaces are provided or required by service participants that are modeled in the Service Contract View (section 8.5).

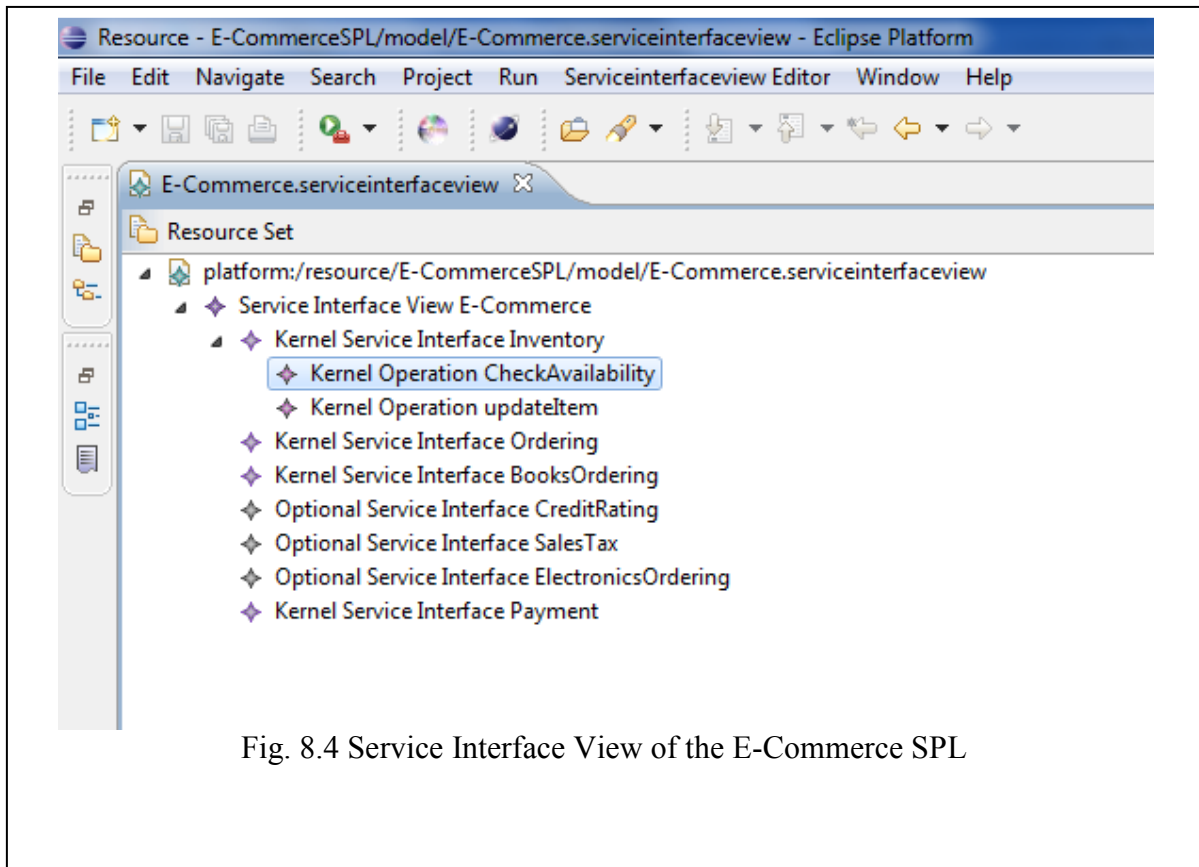


Fig. 8.4 Service Interface View of the E-Commerce SPL

8.8. Service Coordination Variability View Modeling

This Architectural view models service coordinators that coordinate business workflow (Chapter 4). The Order Fulfillment ServiceCoordinator coordinates the Order Fulfillment business process described in section 8.6. Upon examining the Activities of the Order Fulfillment business process (section 8.6) and their corresponding service interfaces

(Section 8.7), a ServiceCoordinator was designed as shown in figure 8.5. The syntax of service coordination was described in Chapter 4.

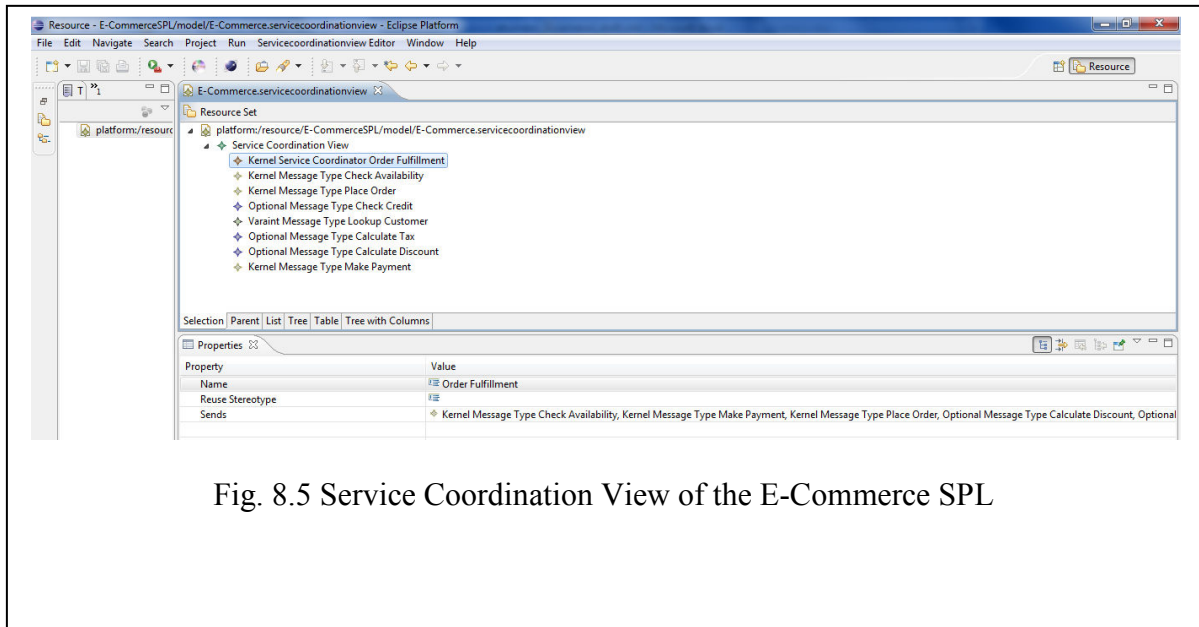


Fig. 8.5 Service Coordination View of the E-Commerce SPL

8.9. Feature View To Service Views Mapping

Once the 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 SoaSPL. Here, the modeler constructed a mapping by associating 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 described in Chapters 5.

Based on the feature view (section 8.4) and the E-Commerce domain knowledge, a mapping was constructed as shown in Fig. 8.6.

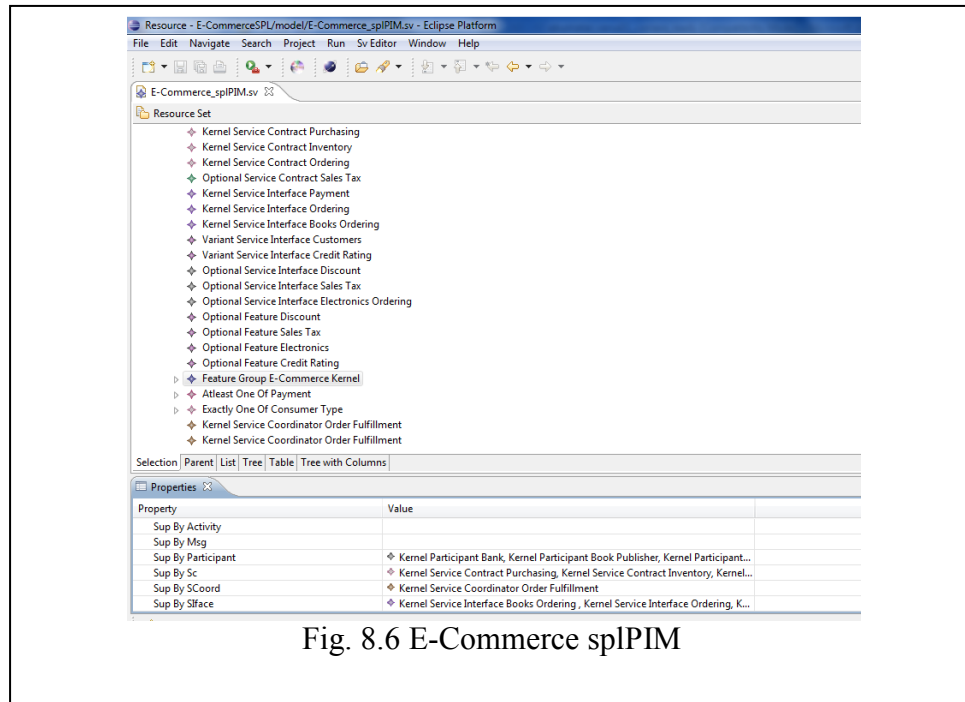


Table 8.1 shows how each feature, in the feature model, is supported by SOA elements from the different views. Also, the table shows that one feature in the feature model could be supported by one or more SOA elements from different views of the multiple view model.

It should be noted that this mapping represents the entire SPL platform-independent model, which was termed splPIM in Chapter 6. The splPIM will be used in the service member derivation phase below to derive service member applications.

Table 8.1 Feature to SOA Mapping

Feature View	Feature Category	Service Contract View	Business Process View	Service Interface View	Service Coordination View
E-Commerce Kernel	Common (this feature includes all the kernel features in the SPL, namely Purchasing, Order Fulfillment, Ordering, Payments, and Books)	Purchasing ServiceContract Inventory ServiceContract Ordering ServiceContract Buyer Participant Seller Participant Bank Participant Book Publisher Participant	Order Fulfillment Business Process	Payment ServiceInterface Ordering ServiceInterface Books Ordering ServiceInterface	Order Fulfillment ServiceCoordinator
Credit Card Payment	Default			Payment ServiceInterface	
Electronic Check Payment	Optional			Payment ServiceInterface	
Preferred Customer	Alternative		'Lookup Preferred Customer' Activity	Customers ServiceInterface	
Regular Customer	Default		'Check CR' Activity	Credit Rating ServiceInterface	
Discount	Optional		'Calculate Discount' Activity	Discount ServiceInterface	
Sales Tax	Optional	Sales Tax ServiceContract	'Calculate Tax' Activity	Sale Tax ServiceInterface	
Electronics	Optional	Electronic Supplier Participant		Electronics Ordering ServiceInterface	
Credit Rating	Optional		'Check CR' Activity	Credit Rating ServiceInterface	

8.10. Member Applications Derivation

At this phase of the case study, a complete multiple-view service-oriented E-Commerce splPIM 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:

8.10.1. Basic E-Commerce Application

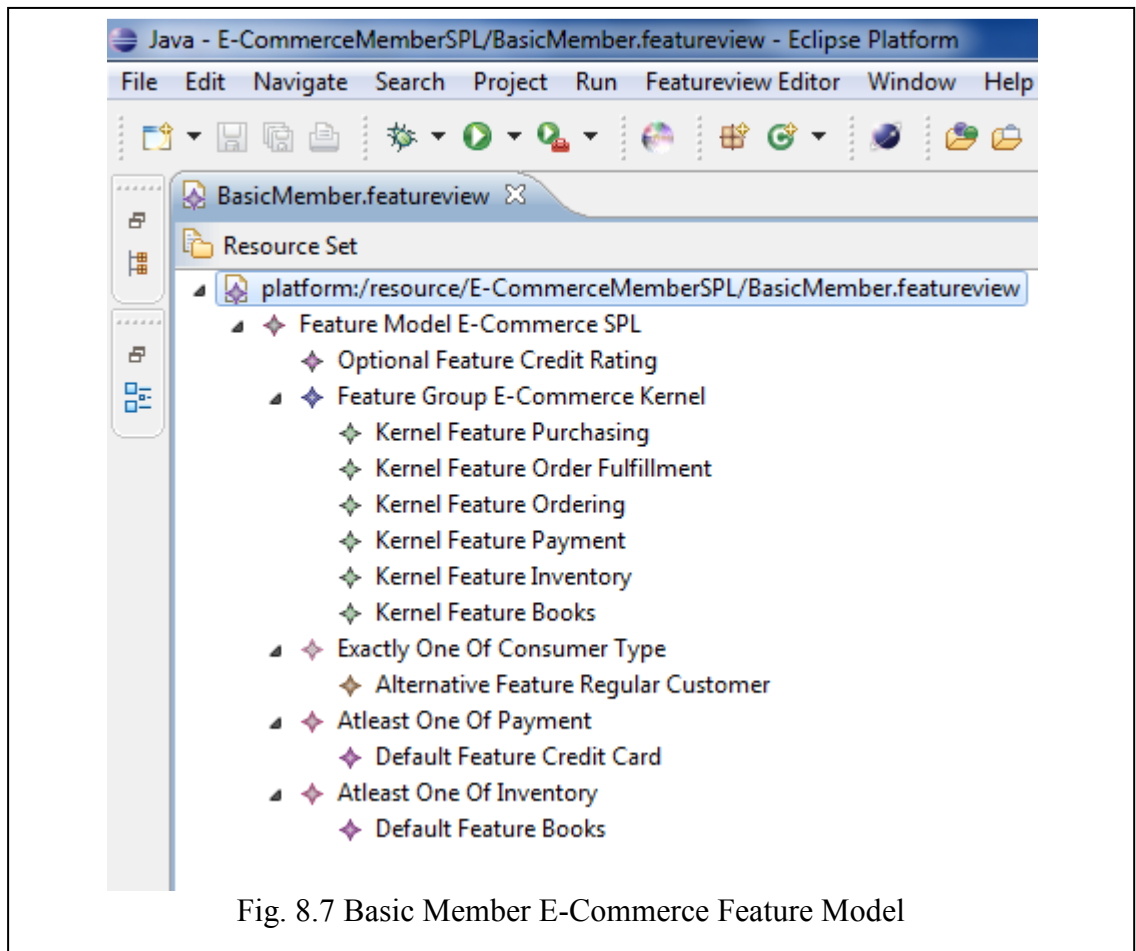
To specify the first member application, feature selection from the feature model (section 8.4) is performed as follows:

- a. 'E-Commerce Kernel' feature is always selected since it is a common feature and contains all kernel features in the SPL. This common feature is supported by: Purchasing Service Contract which includes the Ordering and the Payment Service Interfaces, Seller, Buyer, and Bank Participants which are associated with the Purchasing Service Contract, Inventory Ordering Service Contract which includes the Books Ordering Service Interface, and Book Publisher Participant which is associated with the Inventory Ordering Service Contract, and Order Fulfillment Business Process which is associated with the Seller Participant and includes the Place Order, Make Payment, and Check Availability Activities.
- b. 'Regular Customer' <<alternative feature>> is selected, which is a part of the 'Consumer Type' <<exactly-one-of feature group>>. Consequently, the

‘Credit Rating’ <<optional feature>> is selected since the ‘Regular Customer’ feature requires ‘Credit Rating’. These features are supported by Credit Checking Service Contract, which includes the Credit Rating Service Interface which is associated with the Rating Agency Participant and the Check CR Activity.

- c. ‘Credit Card’ <<default feature>> is selected, which is a part of the ‘Payment’ <<at least-one-of feature group>>. This feature is supported by the Make Payment Activity, and the Payment Service Interface.

To derive the Basic E-Commerce application, the Member Application Derivation facility of SoaSPLE (Chapter 6) was invoked. The derived multiple-view service-oriented Basic E-Commerce application features are depicted in Fig. 8.7.



Next, the Code Generation Facility of SoaSPLE (Chapter 6) 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. The execution logs show how each feature of the feature model is exercised by

tracing the sequence of activities of the Order Fulfillment business process through testing the sequence of the ServiceCoordinator and service invocations.

8.10.2. Enhanced E-Commerce Application

To specify the second member application, feature selection from the feature model is performed as follows:

- a) 'E-Commerce Kernel' feature is again selected since it contains all kernel features in the SPL, and therefore it will be supported by the same SOA elements as in the Basic Application above.
- b) 'Preferred Customer' optional feature, which is supported by the 'Lookup Preferred Customer' Activity in the Order Fulfillment Business Process. Consequently, the 'Discount' feature is automatically selected since it is required by the 'Preferred Customer' feature. The Discount feature is supported by the 'Calculate Discount' Activity in the Order Fulfillment Business Process.
- c) 'Sales Tax' optional feature is selected, which is supported by the 'Calculate Tax' Activity, SalesTax ServiceContract, SalesTax ServiceInterface, and Tax Agency Participant. Again, notice how the selection of one feature drives the selection of several SOA elements that span multiple views.
- d) 'Electronic Check' optional feature is selected along with the 'Credit Card' default feature; these 2 features are supported by the Make Payment Activity and the Payment ServiceInterface.

- e) 'Electronic Goods' optional feature is selected, which is supported by Electronic Supplier Participan and Electronics Ordering ServiceInterface.

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. 8.8.

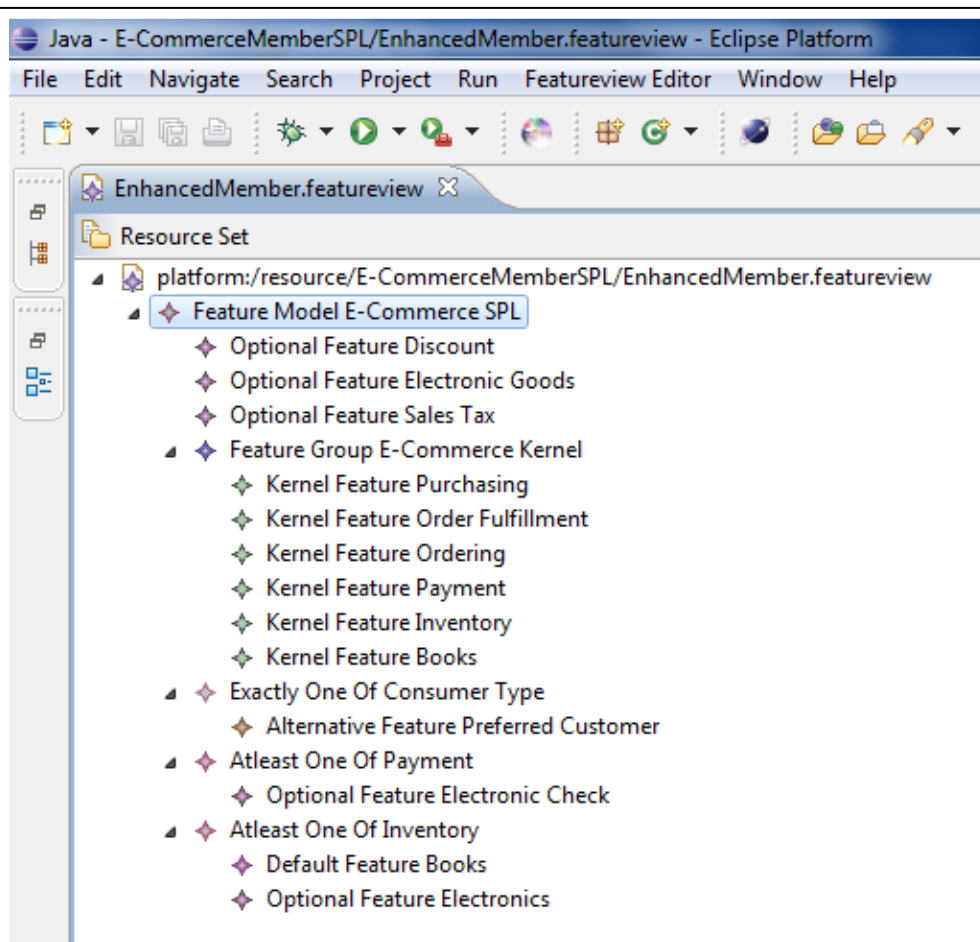


Fig. 8.8 Enhanced Member E-Commerce Feature Model

Next, the Code Generation Facility was invoked to transform the derived Enhanced 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 Enhanced E-Commerce member application was deployed and executed.

8.11. Case Study Conclusion

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 the research approach of this dissertation by satisfying the three properties set forth in Chapter 7, namely: *Consistency*, *Compliance*, and *Legitimacy*. The tool prototype (SoaSPLE) was used to carry out the case study. 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 ServiceCoordinator and service invocations.

9. Hotel Reservation Service-Oriented Software Product Line Case Study

9.1. Case Study Objectives

The purpose of conducting this case study is to evaluate the approach in this research with regard to the validation properties detailed in Chapter 7. These properties are listed here briefly:

- 1 The multiple views of the service-oriented software product line are consistent with each other. This property is referred to as *Consistency*.
- 2 The multiple-view service variability model is compliant with the underlying multiple-view service variability meta-model. This property is referred to as *Compliance*
- 3 Derived software product line member applications are valid service-oriented SPL members. Hence, the evaluation validates the following two characteristics of service member derivation:
 - a. The derived member applications are valid members of the SPL family.
 - b. The derived SPL member applications are valid service-oriented applications.

This property is referred to as *Legitimacy*.

9.2. Case Study Validation Approach

The validation approach of the case study is a system testing approach that 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 tool 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.

9.3. Hotel Reservations Case Study Problem Description

A Hotel Reservation Software Product Line (SPL) can be tailored to the needs of an individual hotel chain or hotel. The system manages information about rooms, reservations, customers, and customer billing. The system provides functionality for making reservations, check in, and check out, in addition to generating reports and displays. In addition, several optional and variant capabilities are provided.

A customer may make reservations, change, or cancel reservations. When making a reservation through a reservation clerk, a customer gives personal details, states the room type, number of occupants, and dates of arrival and departure. A reservation is either guaranteed by credit card or not guaranteed. Reservations that are not guaranteed are automatically cancelled at a pre-specified time, e.g., 6 PM. A no-show customer has to pay for a guaranteed reservation. A desk clerk can check in a customer (with or without a prior reservation), change the checkout date, and check out the customer. A specific room is assigned to the customer at check-in time and a customer record is created. A customer may pay by cash, check, or credit card. A customer billing record is created and the customer receives a check out statement. A customer who does not check out by the checkout time is charged for an additional night.

Optional capabilities of the Hotel Reservation System are:

- a) Management reports. The manager may view the hotel occupancy figure for the present or past dates, view projected occupancy figures for future dates, and view financial information, including room revenue information.
- b) Automatic cancellation for non-guaranteed reservations. At a pre-specified time, e.g., 6 PM, all rooms that are not guaranteed are cancelled and guaranteed reservations are marked as “must-pay”
- c) Automatic billing of no-show guaranteed reservations. At a pre-specified time, e.g., 7 AM, a report is generated of no-shows with guaranteed reservations. A billing record is created for each no-show reservation. At the

same time, a report is also produced giving the total occupancy and revenue (computed from rooms allocated) for the previous night.

- d) Block bookings. A travel company can book a block of rooms at a discounted rate for one or more nights. Bills are charged directly to the travel company.
- e) Variant functionality includes the reservation of residential suites instead of hotel rooms, where a guest can occupy a suite for a week or month at a time, paying a weekly or monthly rate.

The following sections illustrate the steps taken to conduct the case study.

9.4. Feature View Modeling

Commonality and variability analysis of the Hotel Reservations SPL is performed in this step. The result of this step is a feature model (Chapter 4). Based on commonality and variability analysis, a feature model was created in SoaSPLE as shown in Fig. 9.1. 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 (Chapter 5) 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.

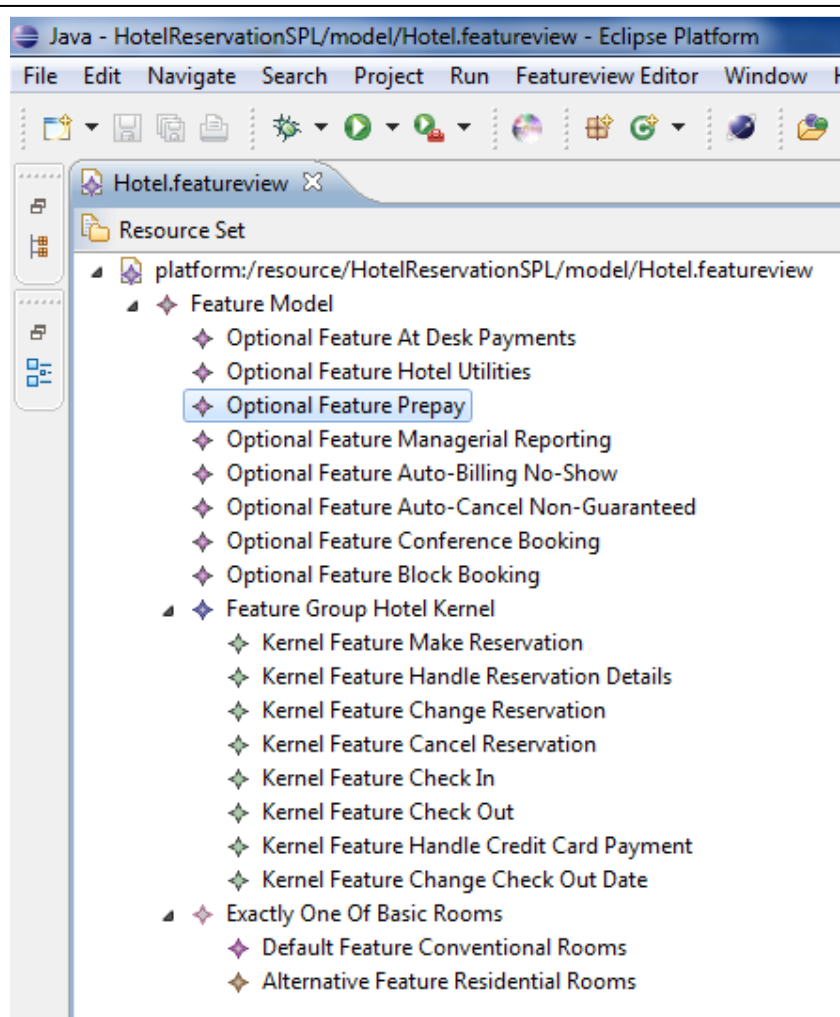


Fig. 9.1 Feature View of the Hotel Reservation SPL

9.5. Service Contract Variability View Modeling

This is a Requirements view that describes service contracts (Chapter 4). Based on the feature model described in the previous section and knowledge of the Hotel Reservation domain, it was decided to model ServiceContracts and Participants as shown in the Service Contract View in Fig. 9.2.

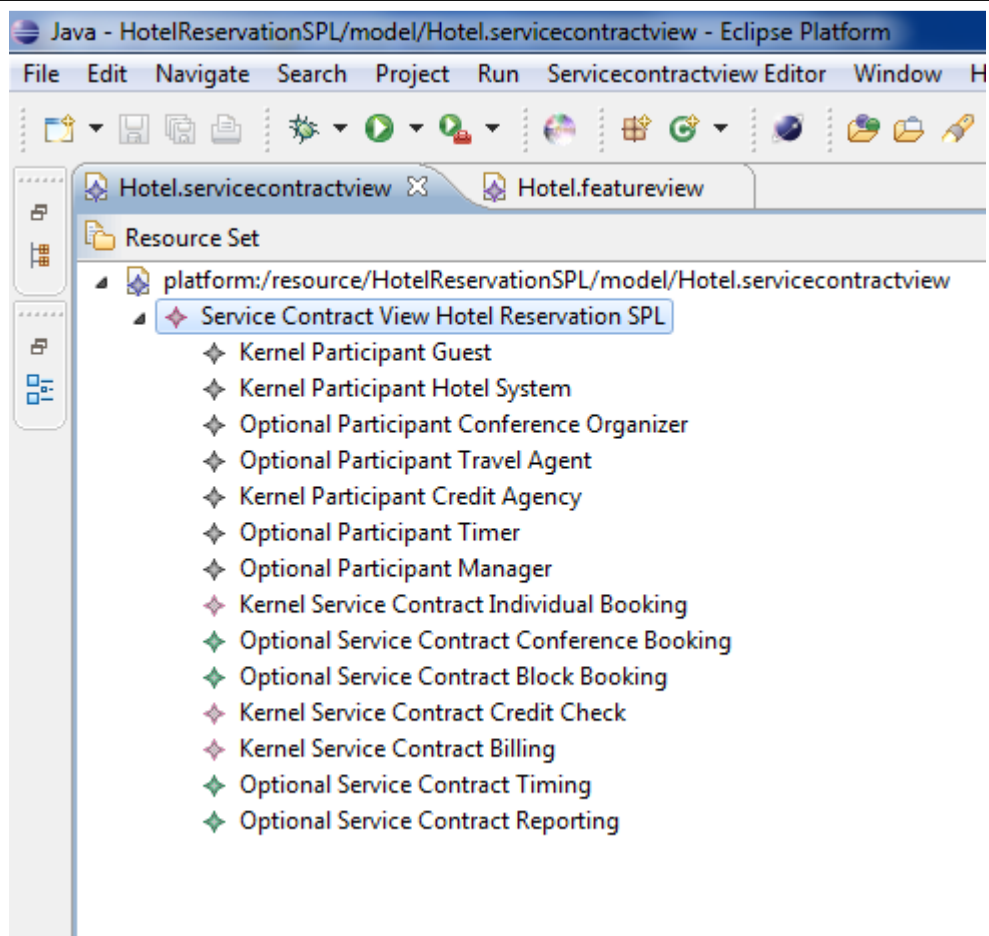


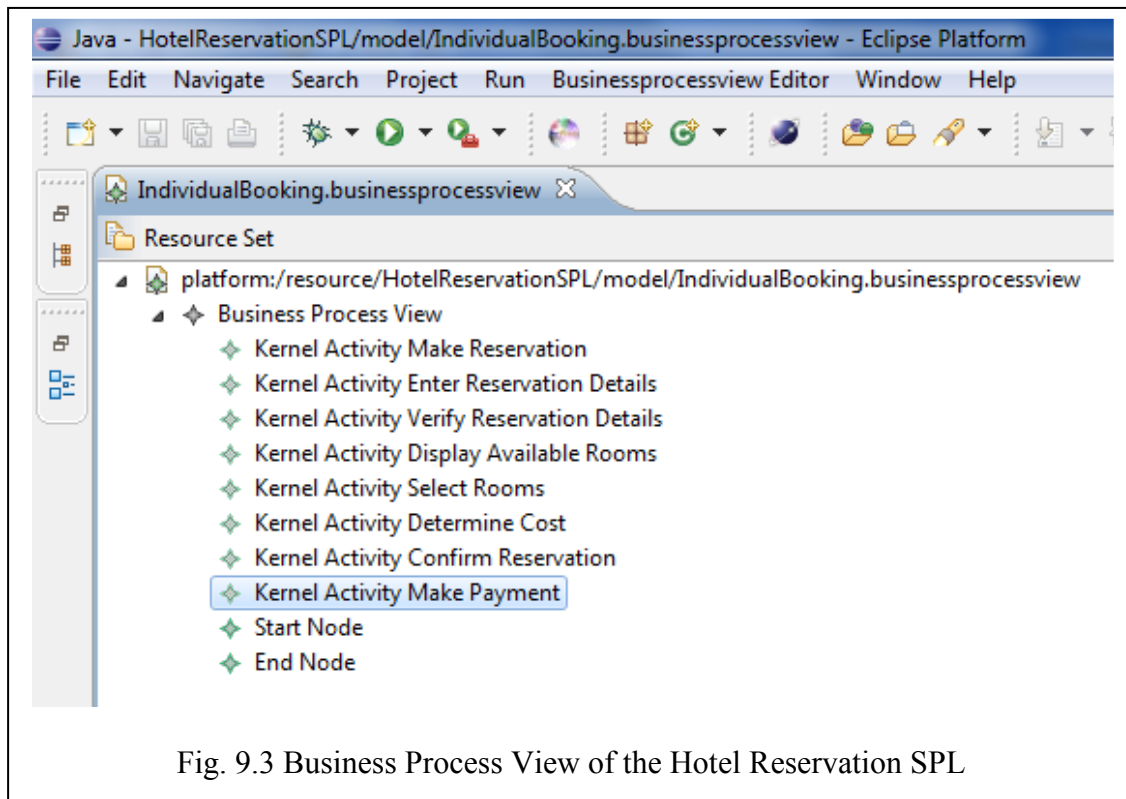
Fig. 9.2 Service Contract View of the Hotel Reservation SPL

As the modeler was building the service contract view model, SoaSPLE automatically checked the model against the Service Contract Meta-View of the multiple-view service variability meta-model (Chapter 5) 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 service contract model.

9.6. Business Process Variability View Modeling

This Requirements view models the workflow of the business processes in the Hotel Reservation SPL. This view is based on the Business Process Meta-Model described in Chapter 5. There are three business processes in the Hotel Reservation SPL: Individual Reservation (kernel), Conference Reservation (optional), and Block Reservation (optional). The Individual Reservation business process view is depicted in Fig. 9.3.

As the modeler was building the business process view model, which consists of three business processes, SoaSPLE automatically checked the model against the Business Process Meta-View of the multiple-view service variability meta-model (Chapter 5) 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 business process model.

9.7. Service Interface Variability View Modeling

This Architectural view models service interfaces, which specify the operations provided or required by Participants. Service interfaces are modeled by using the ServiceInterface element as described in Chapter 5. Upon examining the activities of the three business processes and based on the Hotel Reservation domain knowledge, it was decided to have the service interfaces depicted in Fig. 9.4.

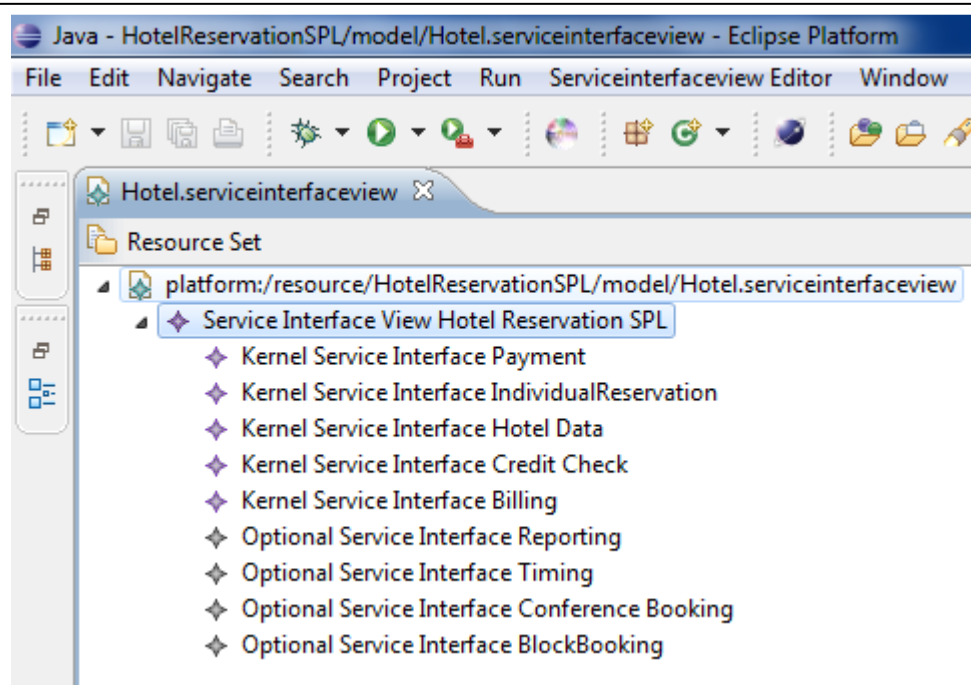


Fig. 9.4 Service Interface View of the Hotel Reservation SPL

As the modeler was building the service interface view model, SoaSPLE automatically checked the model against the Service Interface Meta-View of the multiple-view service variability meta-model (Chapter 5) 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 service interface model.

9.8. Service Coordination Variability View Modeling

This Architectural view models service coordinators that coordinate business workflow (Chapter 4). This view consists of three service coordinators: Individual Booking (kernel), Conference Booking (optional), and Block Booking (optional). Each service coordinator coordinates one of the business processes described in section 9.6. Upon examining the activities of the three business processes (section 9.6) and their corresponding service interfaces (section 9.7), three Service Coordination Views were designed. The Individual Service Coordination View is depicted in Fig. 9.5. SoaSPLE was used to create these views in a similar fashion to the other views.

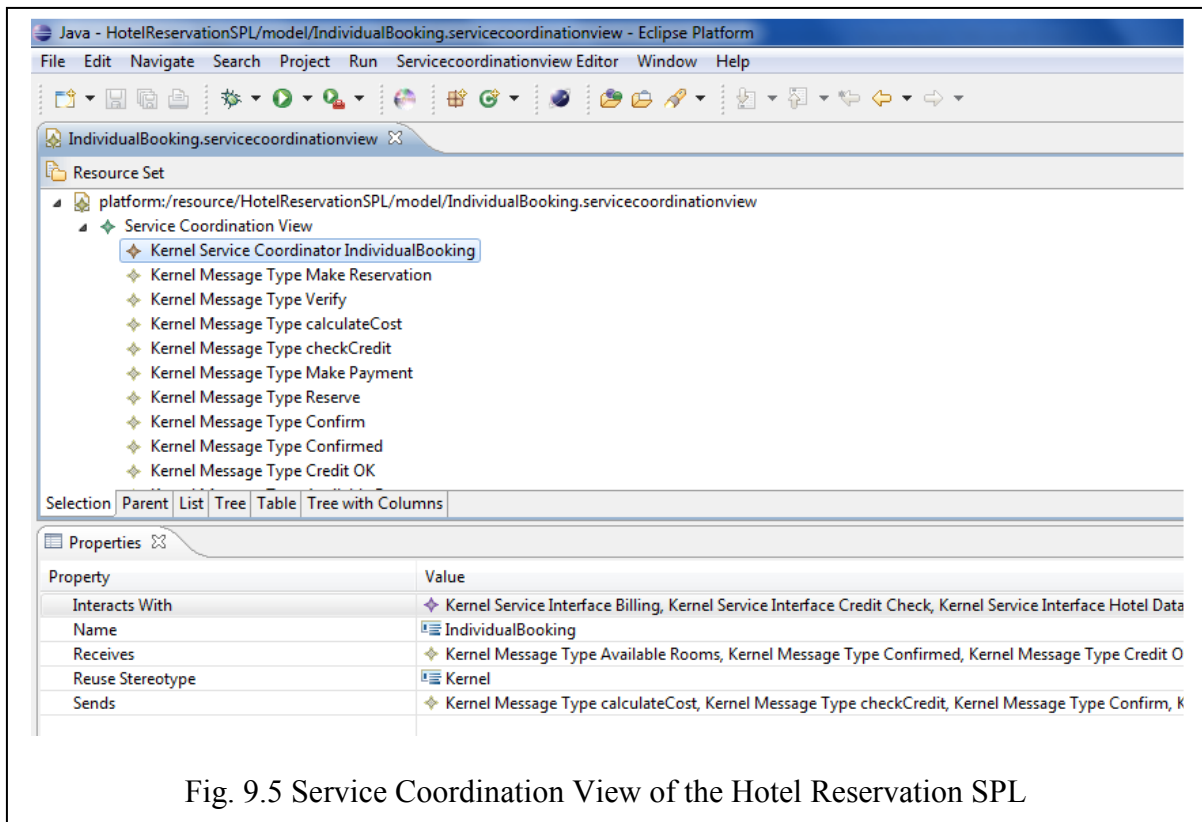


Fig. 9.5 Service Coordination View of the Hotel Reservation SPL

9.9. Feature View To Service Views Mapping

Once the 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, a mapping was constructed by associating 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 described in Chapters 5.

Based on the feature view (section 9.4) and the Hotel Reservations domain knowledge, the mapping was constructed as depicted in Fig 9.6.

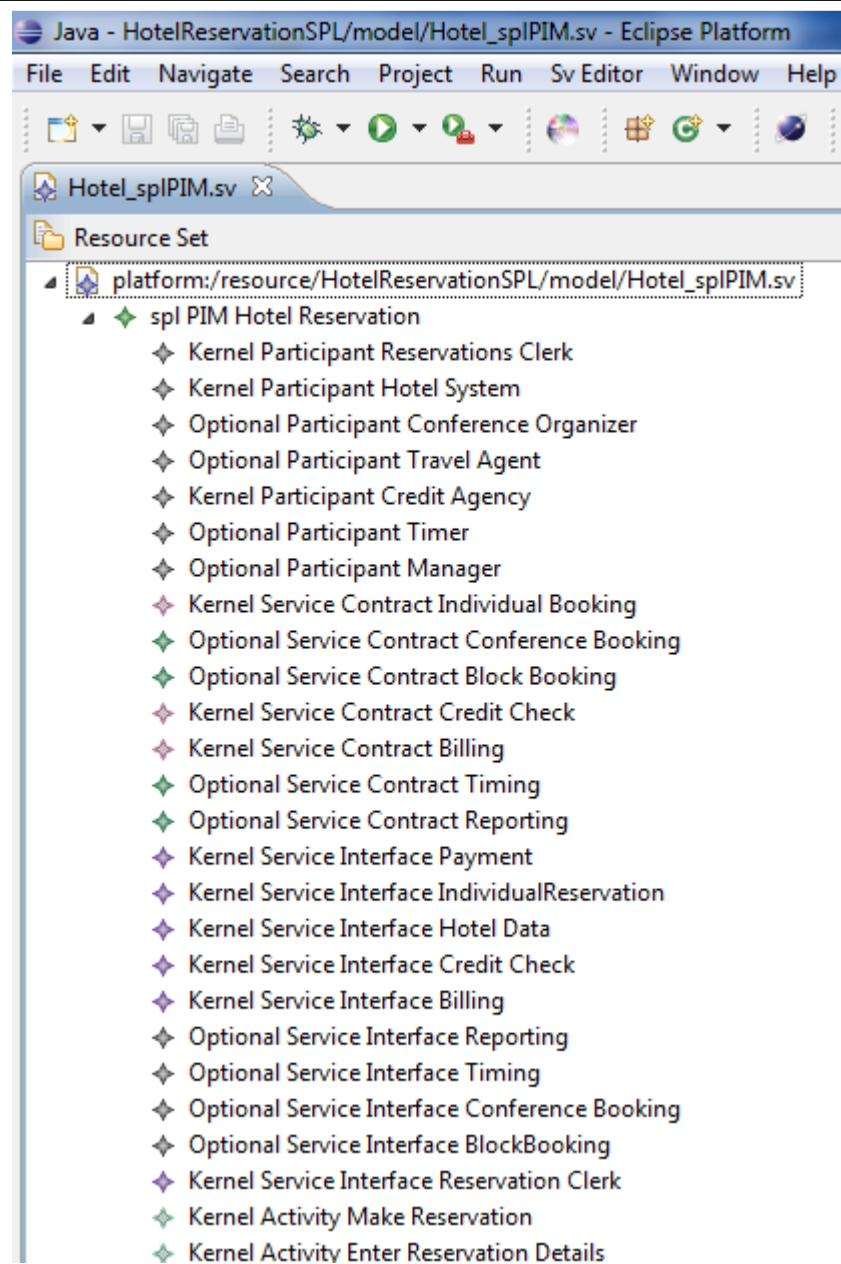


Fig. 9.6 Hotel Reservation splPIM

As the modeler was constructing the mapping, SoaSPLE automatically checked the mapping model against the multiple-view service variability meta-model (Chapter 5) 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 mapping.

It should be noted that this mapping represents the entire SPL platform-independent model, which was termed splPIM in Chapter 6. The splPIM will be used in the service member derivation phase below to derive service member applications.

9.10. Member Applications Derivation

At this phase of the case study, a complete multiple-view service-oriented Hotel Reservation SPL 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:

9.10.1. Conventional Rooms Application

To specify the member application, feature selection from the feature model (section 9.4) was configured as shown in Fig. 9.7.

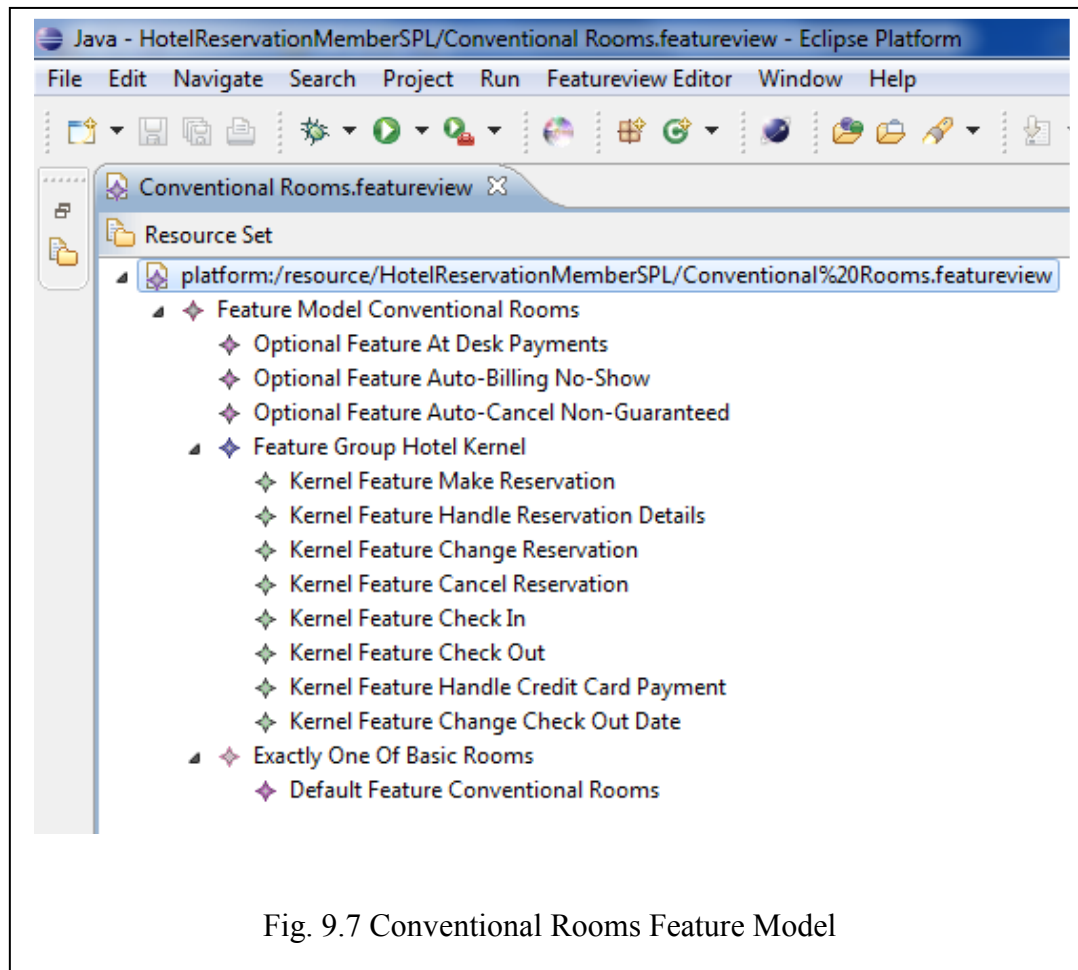


Fig. 9.7 Conventional Rooms Feature Model

To derive the Conventional Rooms application, the Member Application Derivation facility of SoaSPLE (Chapter 6) was invoked. Next, the Code Generation Facility (Chapter 6) was invoked to transform the derived 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 Conventional Rooms member application was deployed and executed. The execution logs show how each feature of the feature model is

exercised by tracing the sequence of the activities of the corresponding business process through testing the sequence of the corresponding ServiceCoordinator and service invocations.

9.10.2. Residential Rooms Application

To specify the member application, feature selection from the feature model (section 9.4) was configured as shown in Fig. 9.8.

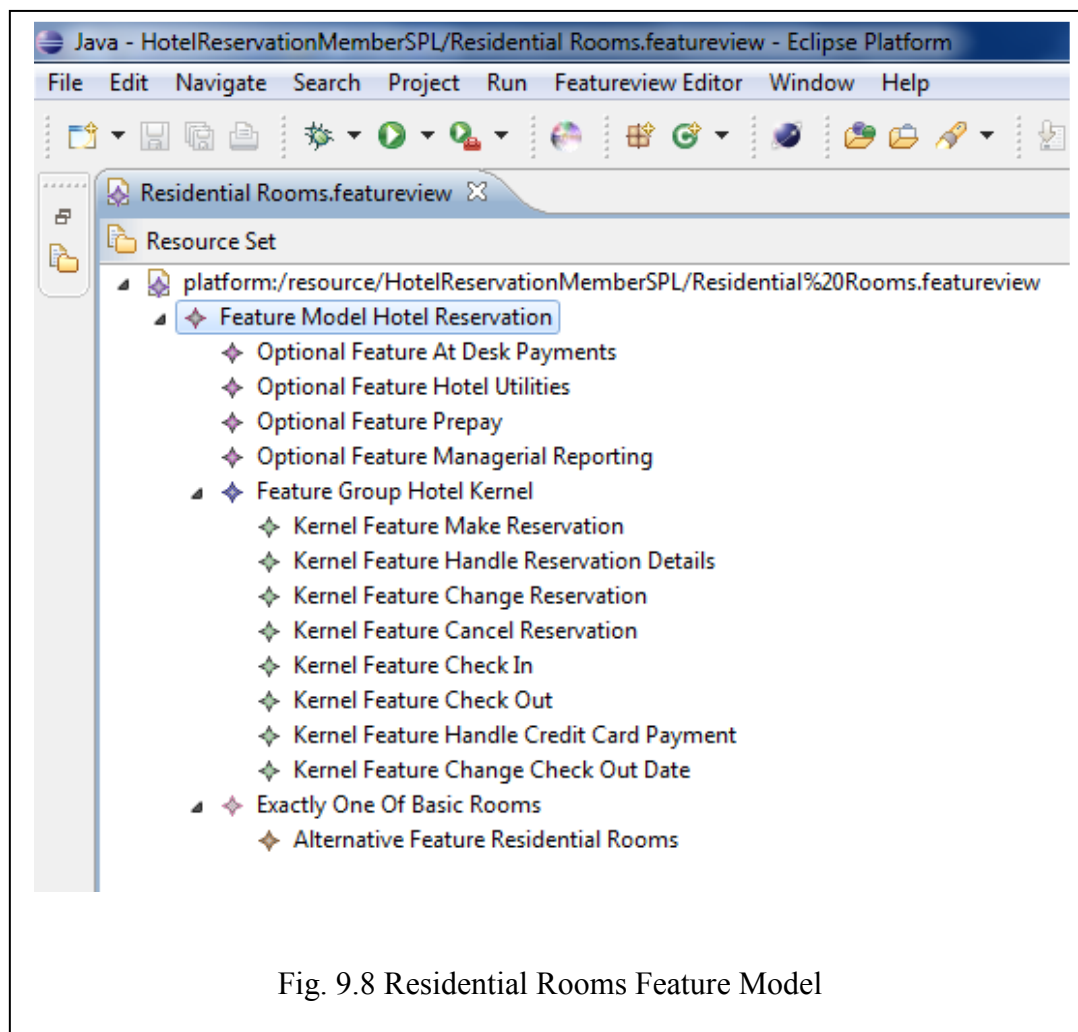


Fig. 9.8 Residential Rooms Feature Model

Next, the Code Generation Facility (Chapter 6) was invoked to transform the derived 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 Residential Rooms member application was deployed and executed.

9.11. Case Study Conclusion

The modeler in the case study modeled the commonality and variability of a Hotel Reservations 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. Screen shots of these errors are depicted in the appendix.

This case study validated the research approach of this dissertation by satisfying the three properties set forth in Chapter 7, namely: Consistency, Compliance, and Legitimacy. The tool prototype (SoaSPLE) was used to carry out the case study. 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 ServiceCoordinator and service invocations.

10. Conclusions

10.1. Introduction

This research has introduced a multiple-view modeling and meta-modeling approach that addresses variability concerns of service oriented application families. The purpose of this research was to develop a multiple-view variability modeling approach to address the design and implementation of variable service-oriented applications families in a *systematic, unified, and platform-independent manner*.

In particular, this research has described the integration of Software Product Lines (SPL) concepts of feature modeling and commonality/variability analysis techniques with service views using the Unified Modeling Language (UML) and the Service Oriented Modeling Language (SoaML).

To achieve the aforementioned goals, this research developed a multiple-view service variability meta-model that specifies the relationships between variable service views (Chapter 5). In addition, this research developed a multiple-view service variability model based on the aforementioned meta-model (Chapter 4). Furthermore, this research created consistency checking rules to ensure the consistency of the multiple views of service-oriented application families (Chapter 5). This research also developed mappings between the Requirements views and the Architectural views of the multiple-view service

variability meta-model (Chapter 5). This research has created derivation rules to *automatically* derive service member applications of service application families (Chapter 6). In addition, this research has developed an automated model-driven service-oriented framework for SPL engineering (Chapter 6).

A proof-of-concept prototype (SoaSPLE) was developed to realize the aforementioned framework. SoaSPLE was used to model and execute service oriented application families and to ensure the consistency of the multiple views of service families.

Validation of this research consisted of Unit Testing and System Testing. Unit testing tests each element and relationship in the multiple-view service variability meta-model.

System Testing is a system-wide testing that tests the derived service-oriented applications of the SPL end-to-end. This research used SoaSPLE to conduct two case studies in the E-Commerce and Hotel Reservation domains (Chapters 8 and 9).

It should be pointed out that although this research described the multiple-view service variability meta-model by using UML and SoaML, the approach could also be used with other service modeling notations.

This chapter briefly outlines the contributions of this research and points out future research goals.

10.2. Research Contributions

The following subsections briefly detail the contributions of this research:

10.2.1. Multiple-View Service Variability Meta-Model

The main contribution of this research is the development of a multiple-view service variability meta-model that defined Service Requirements and Architectural views. The meta-model described the relationships between the views of service-oriented product lines. As described in Chapter 2, previous research has focused on variability analysis and management of individual views of SOA systems, for example on variability of business process flow only or on variability of service interfaces. A major contribution of this research is to enable modelers to analyze what changes in one view if a change happens in another view. In another words, this research has presented an approach to model and manage variability in SOA systems in a *unified* manner.

The multiple-view service variability meta-model described:

- a) Meta-models for variable service Requirements Views – Service Contract and Business Process meta-classes, their attributes, relationships, and constraints.
- b) Meta-models for variable service Architectural Views – Service Interface and Service Coordination meta-classes, their attributes, relationships, and constraints.
- c) Meta-model for a Feature View that describes the variability of the SPL, which is dispersed in the requirements and architectural views. *Thus, the Feature View acts as a unifying view for all other views.*
- d) Intra-View Relationship – The associations and dependencies inside each view were described.
- e) Inter-View Relationships – The associations and dependencies between the different service views were described.

- f) Mapping between Requirements and Architectural Views – the mapping between the Feature View and the Requirements and Architectural Views was described.

10.2.2. Multiple-View Service Variability Model

This research developed a multiple-view service variability model based on the aforementioned meta-model. This model contains service modeling elements expressed in UML and SoaML. This model can be used by modelers to create Requirements and Architectural views of service-oriented SPLs and to create the mappings between the Feature view and the SOA elements of the Requirements and Architectural views.

10.2.3. Consistency Checking and Mapping Rules

This research has developed consistency checking rules to ensure the consistency among the multiple views of service-oriented SPLs. In addition, these rules ensure the proper mapping between the Feature View and the service Requirements and Architectural views of the service-oriented SPL. The consistency checking rules are expressed in the Object Constraint Language (OCL) and are based on the meta-classes and meta-associations of each view of the multiple-view service variability meta-model.

10.2.4. Model Driven Framework for Service Oriented SPLs

Unlike traditional Model Driven Architecture (MDA) approaches, this research introduced the notion of having *multiple* Platform Independent Models (PIMs) to accommodate service-oriented SPLs:

- A software product line PIM – this PIM models the entire service-oriented software family (SPL) with all variability details.
- Service member application PIMs – these PIMs model the derived service member applications of the service-oriented SPL. Each member PIM is derived from the SPL PIM based on feature selection (Chapter 6).

By developing these PIMs, this research helps in facilitating variability modeling of service families in a platform independent way. For example, the approach does not restrict the representation of service interfaces to WSDL or restrict business workflows execution to BPEL.

10.2.5. Service Member Applications Derivation Rules

This research has developed derivation rules for the automatic derivation of service member applications of service-oriented SPLs. The derivation rules derive service member applications PIMs from the SPL PIM. These rules were expressed in Java and they operate on the meta-classes of the multiple-view SPL PIM.

10.2.6. SoaML Variability Notation

This research extended the OMG's SOA Modeling Language (SoaML) with variability notations to enable modeling of service-oriented SPLs. This extension was accomplished by extending the SoaML meta-model with variability meta-classes as explained in Chapter 5.

10.2.7. Explicit Modeling of Service Coordination Variability

Existing SOA SPL variability modeling approaches modeled variability in business process workflow only (Chapter 2). A major contribution of this research is to introduce a Service Coordination View that explicitly models the architectural variability of service coordinators and their service invocations (Chapter 5).

In other words, this research has associated a Service Coordinator with each business process in the Business Process View. This coordinator encapsulates variability information of the business process and coordinates service invocations based on this variability information.

10.2.8. Proof-of-Concept Tool Prototype

This research has developed a proof-of-concept tool prototype (SoaSPLE) to realize the automated model driven SPL framework and to validate the multiple-view service variability approach. SoaSPLE (Chapter 6) embodies the multiple-view service

variability meta-model and provides automatic and semi-automatic capabilities to model, design, implement, and deploy service-oriented SPLs.

SoaSPLE was developed by using the Eclipse Modeling Framework (EMF) and utilized several SOA and JAVA technologies as explained in Chapter 6. SoaSPLE was used to conduct two case studies to validate this research (chapters 8 and 9).

10.3. Future Research

This section describes potential future research inspired by the current research.

10.3.1. Service Variability Mediation Layer

In service-oriented systems, the architecture is not fixed, because the main elements of the architecture are services normally provided by external providers. In addition, a major characteristic of SOA is the decoupling between consumers and provided services [15] where both can change independently of each other. Hence, *the challenge is how to design variability for service consumers independent of provided services and yet use services in a way that satisfies variable application scenarios?*

To solve the aforementioned problem, a Mediation Layer [59] is proposed that sits between service consumers and provided services. This is an intermediary architectural layer that decouples service consumers and providers. This layer consists of Mediation Modules that handle the variability needed by service consumers when they interact with external service providers.

10.3.2. Evolution of Service Oriented SPLs

Once service member applications of the SPL are derived and deployed, new features may be added, modified, or deleted to/from the Feature View. This research is concerned with designing an approach that can handle feature evolution and their impact on existing SPL member applications. The proposed approach should satisfy the evolved requirements while guarantying the operation of existing member applications.

10.3.3. Feature Based Discovery of Service Oriented SPLs

Services could be discovered on demand based on features in the Feature View. Once service member applications are derived, a Service Discovery mechanism is proposed to discover services for member applications. Once services are discovered, service composition commences to build service member applications.

10.3.4. Enhancements of the Tool Prototype (SoaSPLE)

Some enhancements to SoaSPLE are proposed:

- Synthesis of SoaSPLE components into an Eclipse Plug-in [64] which could be used as a standalone modeling tool.
- Replacement of the Java based derivation rules with a standardized transformation language like ATL [68] or QVT [69].

- Addition of a Simulation Environment to enable modelers to simulate the execution of service-oriented SPLs before code generation.

Bibliography

Bibliography

- [1] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.
- [2] P. Kruchten, “The 4+1 View Model of Architecture,” *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995.
- [3] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley Professional, 2004.
- [4] H. Gomaa and M. Saleh, “Software product line engineering for Web services and UML,” in *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, Washington, DC, USA, 2005, p. 110–vii.
- [5] N. Y. Topaloglu and R. Capilla, “Modeling the Variability of Web Services from a Pattern Point of View,” in *Web Services*, vol. 3250, L.-J. (LJ) Zhang and M. Jeckle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 128-138.
- [6] R. Capilla and N. Y. Topaloglu, “Product Lines for Supporting the Composition and Evolution of Service Oriented Applications,” in *Principles of Software Evolution, International Workshop on*, Los Alamitos, CA, USA, 2005, vol. 0, pp. 53-56.
- [7] S. Apel, C. Kaestner, and C. Lengauer, “Research challenges in the tension between features and services,” in *Proceedings of the 2nd international workshop on Systems development in SOA environments*, New York, NY, USA, 2008, pp. 53–58.
- [8] F. M. Medeiros, E. S. Almeida, and S. R. L. Meira, “SOPLE-DE: An Approach to Design Service-Oriented Product Line Architectures,” in *Software Product Lines: Going Beyond*, vol. 6287, J. Bosch and J. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 456-460.
- [9] Abu-Matar, M., Gomaa, H., Kim, M., and Elkhodary, A.M., “Feature Modeling for Service Variability Management in Service-Oriented Architectures,” in *SEKE(2010)*, 2010, pp. 468-473.

- [10] M. Abu-Matar and H. Gomaa, "Feature Based Variability for Service Oriented Architectures," in *2011 9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2011, pp. 302-309.
- [11] M. Abu-Matar and H. Gomaa, "Variability Modeling for Service Oriented Product Line Architectures," in *Software Product Line Conference (SPLC), 2011 15th International*, 2011, pp. 110-119.
- [12] M. Abu-Matar and H. Gomaa, "Service Variability Meta-Modeling for Service-Oriented Architectures," presented at the ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems MODELS 2011, Wellington, New Zealand, 2011.
- [13] "OASIS SOA Reference Model TC | OASIS." [Online]. Available: <http://www.oasis-open.org/committees/soa-rm/>. [Accessed: 23-Jul-2011].
- [14] M. N. Huhns and M. P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75-81, 2005.
- [15] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.
- [16] "Elements of Service-Oriented Analysis and Design," 02-Jun-2004. [Online]. Available: <http://www.ibm.com/developerworks/webservices/library/ws-soad1/>. [Accessed: 09-May-2011].
- [17] M. Colombo, E. Nitto, M. Penta, D. Distanto, and M. Zuccalà, "Speaking a Common Language: A Conceptual Model for Describing Service-Oriented Systems," in *Service-Oriented Computing - ICSOC 2005*, vol. 3826, B. Benatallah, F. Casati, and P. Traverso, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 48-60.
- [18] I. H. Kruger and R. Mathew, "Systematic development and exploration of service-oriented software architectures," pp. 177- 187, Jun. 2004.
- [19] Arsanjani, Ali, "Service-oriented modeling and architecture," 09-Nov-2004. [Online]. Available: <http://www.ibm.com/developerworks/library/ws-soa-design1/>. [Accessed: 13-May-2011].
- [20] Johnston, Simon, "UML 2.0 Profile for Software Services," 13-Apr-2005. [Online]. Available: http://www.ibm.com/developerworks/rational/library/05/419_soa/. [Accessed: 13-May-2011].

- [21] N. Milanovic and M. Malek, "Current Solutions for Web Service Composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51-59, Nov-2004.
- [22] P. Yu, X. Ma, and J. Lu, "Dynamic Software Architecture Oriented Service Composition and Evolution," in *Proceedings of the The Fifth International Conference on Computer and Information Technology*, Washington, DC, USA, 2005, pp. 1123–1129.
- [23] A. Kalnins and V. Vitolins, "Use of UML and Model Transformations for Workflow Process Definitions," *cs/0607044*, Jul. 2006.
- [24] A. Schnieders, "Variability Mechanism Centric Process Family Architectures," in *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, Washington, DC, USA, 2006, pp. 289–298.
- [25] "UML 2.0 Superstructure Specification," 15-Apr-2011. [Online]. Available: <http://dret.net/biblio/reference/uml20super>. [Accessed: 13-May-2011].
- [26] E. M. Olimpiew and H. Gomaa, "Reusable Model-Based Testing," in *Formal Foundations of Reuse and Domain Engineering*, vol. 5791, S. H. Edwards and G. Kulczycki, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 76-85.
- [27] Y. Choi, G. Shin, Y. Yang, and C. Park, "An Approach to Extension of UML 2.0 for Representing Variabilities," in *Proceedings of the Fourth Annual ACIS International Conference on Computer and Information Science*, Washington, DC, USA, 2005, pp. 258–261.
- [28] S. H. Chang and S. D. Kim, "A Service-Oriented Analysis and Design Approach to Developing Adaptable Services," in *Services Computing, IEEE International Conference on*, Los Alamitos, CA, USA, 2007, vol. 0, pp. 204-211.
- [29] F. Curbera, D. Ferguson, M. Nally, and M. L. Stockton, "Toward a Programming Model for Service-Oriented Computing," in *Service-Oriented Computing - ICSOC 2005*, vol. 3826, B. Benatallah, F. Casati, and P. Traverso, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 33-47.
- [30] K. Petersen, N. Bramsiepe, and K. Pohl, "Applying Variability Modeling Concepts to Support Decision Making for Service Composition," pp. 1-1, Sep. 2006.
- [31] S. Malek, N. Esfahani, D. A. Menasce, J. P. Sousa, and H. Gomaa, "Self-Architecting Software SYstems (SASSY) from QoS-annotated activity models," in *2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, Vancouver, BC, Canada, 2009, pp. 62-69.

- [32] N. Esfahani, S. Malek, J. P. Sousa, H. Gomaa, and D. A. Menascé, “A Modeling Language for Activity-Oriented Composition of Service-Oriented Software Systems,” in *Model Driven Engineering Languages and Systems*, vol. 5795, A. Schürr and B. Selic, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 591-605.
- [33] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malex, and J. P. Sousa, “A framework for utility-based service oriented design in SASSY,” in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering - WOSP/SIPEW '10*, San Jose, California, USA, 2010, p. 27.
- [34] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, 3rd ed. Addison-Wesley Professional, 2001.
- [35] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 1st ed. Springer, 2005.
- [36] H. Gomaa, *Designing Concurrent, Distributed (text only) by H. Gomaa*. Addison-Wesley Professional, 2000.
- [37] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.
- [38] M. L. Griss, J. Favaro, and M. d'Alessandro, “Integrating Feature Modeling with the RSEB,” in *Proceedings of the 5th International Conference on Software Reuse*, Washington, DC, USA, 1998, p. 76–.
- [39] D. M. Weiss, “Software Synthesis: The FAST Process,” *IN PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON COMPUTING IN HIGH ENERGY PHYSICS, RIO DE JANEIRO*, 1995.
- [40] M. Matinlassi, “Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA,” in *Proceedings of the 26th International Conference on Software Engineering*, Washington, DC, USA, 2004, pp. 127–136.
- [41] M. Morisio, G. H. Travassos, and M. E. Stark, “Extending UML to Support Domain Analysis,” in *Proceedings of the 15th IEEE international conference on Automated software engineering*, Washington, DC, USA, 2000, p. 321–329.
- [42] I. Philippow and M. Riebisch, “Systematic Definition of Reusable Architectures,” 2001.

- [43] S. D. Kim, S. H. Chang, and H. J. La, "Traceability Map: Foundations to Automate for Product Line Engineering," in *Software Engineering Research, Management and Applications, ACIS International Conference on*, Los Alamitos, CA, USA, 2005, vol. 0, pp. 340-347.
- [44] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," in *Generative Programming and Component Engineering*, vol. 3676, R. Glück and M. Lowry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 422-437.
- [45] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.
- [46] P. B. Kruchten, "The 4 1 view model of architecture," *IEEE SOFTWARE*, vol. 12, p. 42--50, 1995.
- [47] B. Nuseibeh, J. Kramer, and A. Finkelstein, "Expressing the relationships between multiple views in requirements specification," in *Proceedings of the 15th international conference on Software Engineering*, Baltimore, Maryland, United States, 1993, pp. 187–196.
- [48] E. O'Hara-Schettino and H. Goma, "Dynamic navigation in multiple view software specifications and designs," *J. Syst. Softw.*, vol. 41, no. 2, pp. 93–103, May 1998.
- [49] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008.
- [50] T. Clark, A. Evans, P. Sammut, and J. Willans, *Applied Metamodelling: A Foundation for Language Driven Development*. 2004.
- [51] S. J. MELLOR, K. Scott, A. Uhl, and D. Weise, *MDA Distilled*. Addison-Wesley Professional, 2004.
- [52] A. Kleppe, J. Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*, 1st ed. Addison-Wesley Professional, 2003.
- [53] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The*, 2nd ed. Addison-Wesley Professional, 2005.
- [54] "SoaML." [Online]. Available: <http://www.omg.org/spec/SoaML/>. [Accessed: 23-Jul-2011].

- [55] H. Gomaa and M. E. Shin, "Multiple-view modelling and meta-modelling of software product lines," *IET Software*, vol. 2, no. 2, pp. 94-122, Apr. 2008.
- [56] "MDA." [Online]. Available: <http://www.omg.org/mda/>. [Accessed: 26-Jul-2011].
- [57] A. Helferich, G. Herzwurm, S. Jesse, and M. Mikusz, "Software product lines, service-oriented architecture and frameworks: worlds apart or ideal partners?," in *Proceedings of the 2nd international conference on Trends in enterprise application architecture*, Berlin, Heidelberg, 2007, pp. 187-201.
- [58] J. Park, "An approach to developing reusable domain services for service oriented applications," New York, NY, USA, 2010, pp. 2252-2256.
- [59] M. Abu-Matar, "Mediation Based Variability Modeling for Service Oriented Software Product Lines," presented at the SEKE09, 2009.
- [60] Medeiros, F. M., Almeida, E. S. d., and Meira, S. R. d. L., "Towards an Approach for Service-Oriented Product Line Architectures," in *Workshop on Service-oriented Architectures and Software Product Lines*, 2009.
- [61] S. Gunther and T. Berger, "Service-oriented product lines: Towards a development process and feature management model for web services," in *12th International Software Product Line Conference*, 2008.
- [62] H. Gomaa, *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*, 1st ed. Cambridge University Press, 2011.
- [63] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. A. Menascé, "Software adaptation patterns for service-oriented architectures," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010, pp. 462-469.
- [64] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*, Dimensions: 7x9-1/4 ed. 2008.
- [65] "Apache ODE -- Index." [Online]. Available: <http://ode.apache.org/>. [Accessed: 25-Jul-2011].
- [66] "Apache CXF -- Index." [Online]. Available: <http://cxf.apache.org/>. [Accessed: 25-Jul-2011].
- [67] "Swordfish SOA Runtime Framework Project." [Online]. Available: <http://www.eclipse.org/swordfish/>. [Accessed: 25-Jul-2011].

- [68] “ATL.” [Online]. Available: <http://eclipse.org/atl/>. [Accessed: 25-Jul-2011].
- [69] “QVT 1.0.” [Online]. Available: <http://www.omg.org/spec/QVT/1.0/>. [Accessed: 25-Jul-2011].

CURRICULUM VITAE

Mohammad Abu Matar is a software engineering academic and practitioner with over 17 years of technical experience in teaching, research, management, architecture, systems engineering, training, software design and development.

Mohammad has earned a BS degree in Electrical Engineering (Wright State University), an MS degree in Information Technology (Regis University), an MS degree in Software Engineering (George Mason University), and a PhD in Software Engineering from George Mason University.

Mohammad's specialty is the architecture of multi-tier distributed software systems with a special interest in Service Oriented Architecture (SOA).

Mohammad is an independent IT Architecture Consultant and an Affiliate Adjunct Faculty at the MS in Software Engineering program of Regis University (Denver, Colorado).