Describing Dynamic and Variable Software Architecture Based on Identified Services From Object-Oriented Legacy Applications

Seza Adjoyan

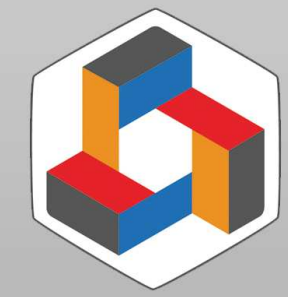
THESIS
to obtain the title of
PhD of Science

Granted by the **UNIVERSITY OF MONTPELLIER**

Prepared at the doctoral school **I2S**
and the research unit **LIRMM**

Specialty: **Computer Science**

Presented by **Seza ADJOYAN**



Describing Dynamic and Variable Software Architecture Based on Identified Services From Object-Oriented Legacy Applications

Defended on the 30th of June 2016 in front of the jury composed of:

Mr. Philippe ANIORTÉ	Prof.	Univ. of Pau and Pays de l'Adour	Reviewer
Mr. Henri BASSON	Prof.	Univ. of the Littoral Opal Coast	Reviewer
Mr. Mourad OUSSALAH	Prof.	Univ. of Nantes	Examinator
Mr. Fabien MICHEL	Dr., HDR	Univ. of Montpellier	Examinator
Mr. Roland DUCOURNAU	Prof.	Univ. of Montpellier	Advisor
Mr. Abdelhak SERIAI	Dr.	Univ. of Montpellier	Co-Advisor

Acknowledgments

First and foremost, I would like to express my gratitude to the members of my dissertation committee, **Mr. Philippe Aniorté**, **Mr. Henri Basson**, **Mr. Mourad Oussalah**, and **Mr. Fabien Michel** for having accepted to judge my thesis work. I highly appreciate your patience as well as the time you took out of your busy schedule to read and evaluate my thesis work. Thank you for having provided me your constructive comments and inspiring remarks which definitely improve the quality of the conducted research and its results.

I would like to thank my thesis advisor, **Mr. Roland Ducournau**, for having trusted in me and for having given me the freedom to develop my initiatives throughout the PhD years. Moreover, you have always supported me and received me with patience and kindness.

I would also like to thank my thesis co-advisor, **Mr. Abdelhak-Djamel Seriai**, for his continuous investment in my thesis, for his remarkable human qualities and for his pointed advices as much on the methodological side as on the technical side. Mr. Seriai, indeed, you have supported me from my first arrival at LIRMM until the final completion of my dissertation. Completing this thesis would not have been possible without your help and encouragement, whether on the academic or personal level, for which I will always remain grateful.

I would like to extend my sincere thanks to **every permanent member of MaREL team** for having welcomed me in their team over these last few years. You have granted me support every time I needed it. Simply, I appreciate to have known you and hope to meet you again during further scientific events and/or to continue coordination with you.

More generally, I would like to thank the **laboratory LIRMM** for giving me the opportunity to work and complete my PhD degree under the best working conditions. Likewise, I would like to thank the teaching staff of the Faculty of Sciences of the **University of Montpellier**, who put their confidence in me and wholeheartedly supported me to accomplish my teaching missions.

I would like to thank **Erasmus Mundus** program for providing me the opportunity as well as the scholarship to integrate and eventually obtain my PhD degree from such a high-quality EU institution as the University of Montpellier. I would also like to convey my appreciation to **Calouste Gulbenkian foundation** for providing me the fund to complete my research in favourable conditions.

I would like to thank **my friends / colleagues** (current and former PhD students) at LIRMM for the wonderful time I spent with their company. Thank you for the quality moments and for the enriching scientific discussions we had together as well as for sharing together the everyday concerns of a PhD student. I wish each and everyone of you the best of luck and success in your future endeavors and look forward to meeting you again. I share my success with you!

I would like to thank my beloved husband **Yenovk Tokatelian** for all the sacri-

fices he has made on my behalf. Your generosity, love, patience and encouragements were the only way forward for me to overcome all obstacles and challenges. I owe a lot to you!

Thank you my lovely daughter **Lia** for her unconditional love and innocent smile that always lifted my spirit. Thank you for letting mom work on her dissertation late at nights whenever you slept. For the source of inspiration you have been for me, I dedicate this thesis to you, my dear Lia.

Lastly, I would like to sincerely thank **my parents and my parents-in-law** for having transmitted to me the taste and interest in studies and research. Thank you for the important moral support you provided to me during my PhD adventure, despite all the suffering and difficulties you have experienced. Your choice and determination to stay and not quit your homeland Syria in this difficult period taught me, among others, braveness and courage. Although, I wished you could attend my PhD defense, however I wish you can survive and continue living in peace. I love you and miss you very much.

Thanks to my professors, family and friends who helped me along the way; I am lucky to have you in my life.

Describing Dynamic and Variable Software Architecture Based on Identified Services From Object-Oriented Legacy Applications

Abstract: Service Oriented Architecture (SOA) is an architectural design paradigm which facilitates building and composing flexible, extensible and reusable service-oriented assets. These latter are encapsulated behind well-defined and published interfaces that can be dynamically discovered by third-party services. Before the advent of SOA, several software systems were developed using older technologies. Many of these systems still afford a business value, however they suffer from evolution and maintenance problems. It is advantageous to modernize those software systems towards service-based ones. In this sense, several re-engineering techniques propose migrating object-oriented applications towards SOA. Nonetheless, these approaches rely on ad-hoc criteria to correctly identify services in object-oriented legacy source code.

Besides, one of the most distinguishing features of a service-oriented application is the ability to dynamically reconfigure and adjust its behavior to cope with changing environment during execution. However, in existing architecture description languages handling this aspect, reconfiguration rules are represented in an ad-hoc manner; reconfiguration scenarios are often implicit. This fact hinders a full management of dynamic reconfiguration at architecture level. Moreover, it constitutes a challenge to trace dynamic reconfiguration description/ management at different levels of abstraction.

In order to overcome the aforementioned problems, our contributions are presented in two axes: First, in the context of migrating legacy software towards SOA, we propose a service identification approach based on a quality measurement model, where service characteristics are considered and refined to metrics in order to measure the semantic correctness of identified services. The second axis is dedicated to an Architecture Description Language (ADL) proposition that describes a variant-rich service-based architecture. In this modular ADL, dynamic reconfigurations are specified at architecture level. Moreover, the description is enriched with context and variability information, in order to enable a variability-based self-reconfiguration of architecture in response to context changes at runtime.

Keywords: Service-Oriented Architecture (SOA), re-engineering, variability, Architecture Description Language (ADL), reconfiguration, dynamic architecture

Architecture Dynamique Basée sur la Description de la Variabilité et des Services Identifiés Depuis des Applications Orientées Objet

Résumé: L'Orienté Service (SOA) est un paradigme de conception qui facilite la construction d'applications extensibles et reconfigurables basées sur des artefacts réutilisables qui sont les services. Ceux-ci sont structurés via des interfaces bien définies et publiables et qui peuvent être dynamiquement découvertes. Les applications SOA peuvent être conçues selon deux démarches différentes. La première est la démarche classique qui conçoit le système à partir de spécification de besoin (i.e. forward engineering en anglais). La deuxième démarche consiste à créer le système SOA par la réingénierie d'un système existant (i.e. re-engineering en anglais). Beaucoup d'approches ont été proposées dans la littérature pour la réingénierie d'applications existantes développées dans des paradigmes pré-services, principalement l'orienté objet, vers SOA. L'objectif est de permettre de sauvegarder la valeur métier de ces d'applications tout en leur permettant de bénéficier des avantages de SOA. Le problème est que ces approches s'appuient sur des critères ad-hoc pour identifier correctement des services dans le code source des applications existantes.

Par ailleurs, l'une des caractéristiques les plus distinctives d'une application orientée service est sa capacité de se reconfigurer dynamiquement et d'adapter son comportement en fonction de son contexte d'exécution. Cependant, dans les langages de description d'architecture (ADL) existants dont l'aspect de reconfiguration et pris en compte, les règles de reconfiguration sont représentées d'une manière ad-hoc; en général, elles ne sont pas modélisées d'une manière explicite mais enfouillées dans la description de l'architecture. D'une part, ceci engendre une difficulté de la gestion de la reconfiguration dynamique au niveau de l'architecture et d'autre part, la traçabilité de la description de la reconfiguration dynamique à travers les différents niveaux d'abstraction est difficile à représenter et à gérer.

Afin de surmonter les problèmes précédents, nous proposons dans le cadre de cette thèse deux contributions. D'abord, nous proposons une approche d'identification de services basée sur un modèle de qualité où les caractéristiques des services sont étudiées, raffinées et réifiées en une fonction que nous utilisons pour mesurer la validité sémantique de ces services. La deuxième contribution consiste en une proposition d'un langage de description d'architecture orientée service (ADL) qui intègre la description de la variabilité architecturale. Dans cet ADL les services qui peuvent constituer l'architecture, les éléments de contexte dont les changements d'état sont à l'origine des changements architecturaux, les variantes des éléments architecturaux sélectionnées en fonction des états des éléments de contexte et le comportement architectural dynamique sont ainsi spécifiés de façon modulaire.

Mots-clés: Architecture orientée service, réingénierie, variabilité, langage de description d'architecture, reconfiguration, architecture dynamique

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Contents

1.1 Context

Human supervision to reconfigure the behavior of software systems which are subject to environment changes is considered a costly and time-consuming task [Salehie 2009]. To keep pace with the increasing development of such systems notably in terms of their size and complexity and to guarantee the quality of system's adaptation and its efficiency, availability and responsiveness against changing external conditions, rendering the system self-configurable and dynamically adaptable could be an effective solution.

Self-configurable (or self-adaptive) systems aim to adapt their various artifacts (attributes or their behavior) autonomously in response to changes in the operating environment (i.e. context changes) without human interaction [Classen 2008]. Such systems are capable to dynamically adapt their configuration. This allows improving flexibility and responsiveness to users' preferences or varying operating conditions [Jaggernauth 2015], [Fiadeiro 2013]. The notion of dynamicity indicates that any adaptation occurs during system's execution. This is a required property in systems where stopping the execution to make modifications on it might cause dramatic effects. Self-configurable and dynamically adaptable systems are used in different domains of application such as natural catastrophe prevention (flood warning), traffic control system, e-commerce applications, etc.

1.1.1 Service Oriented Architecture's Support to Self-Adaptive Systems

Service Oriented Architecture (SOA), whose main bricks are services [Lewis 2005], has become a trend of computing paradigm to describe business functionalities and application logics [Chen 2005], [Zhang 2005]. In SOA, a system is structured into a set of loosely coupled [Chen 2009], [Nakamura 2009], [Papazoglou 2007] and interoperable business services that can be easily composed [Lewis 2005], reused [Lewis 2005] and shared [Corporation 2008] regardless of their physical location. Services could either be entirely deployed on a single machine, residing on several machines of company's internal network, or even distributed on several systems over Internet [Brown 2002]. Moreover, having solid service-oriented architecture in place will provide the infrastructure needed to successfully deploy services in a cloud environment. SOA enables the composition of heterogeneous third-party sub-systems that run in a varying execution environment [Griffiths 2010]. As a framework, SOA facilitates the creation of flexible, extensible and reusable assets (i.e. services) with different granularities hidden behind well-defined interfaces that describe service's functionality. This activity is called service encapsulation [Endrei 2004]. Realizing Service-Oriented Architecture (SOA) system via Web service technology can provide the required dynamicity and flexibility [Papazoglou 2008] and hence support the design of dynamically self-adaptive systems. Using Web service technology offers many advantages for implementing a distributed system; particularly the possibility to provide interoperability across heterogeneous software artifacts [Papazoglou 2008].

1.1.2 Variability Modeling as a Support to Self-Adaptive Systems

Variability modeling, which is one of the key activities of Software Product Lines (SPL), explicitly represents the variation among software products of the same product family in terms of features [Abu-Matar 2011]. It defines which features are mandatory, optional or alternative in a system in addition to specifying the cross-cutting conditions of requiring and exclusion between those features [Capilla 2014]. As a matter of fact, SPL aims to build a collection of similar products from a single core asset [Bachmann 2005]. This is achieved by identifying commonalities and variations of a product family at different levels of abstraction and representing them in a variability model. This variability model represents commonalities and variations of a product family at different levels of abstraction such as requirement, architecture, or implementation level. In dynamically self-adaptive systems, variability modeling can be applied in order to define system's runtime adaptations to environment changes and to reconfigure its composition accordingly [Classen 2008].

1.2 Problem Statement

Self and dynamic adaptability is a required property to realize various kind of systems. Service-Oriented Architecture is one of the preferred technology to materialize this property.

On the one hand, many software applications have been developed using older and out-to-dated technologies. Those applications are designated as *legacy software*. Despite the fact that business-critical legacy software still probably affords a great business value for its users, it suffers from a particularly complex evolution and maintenance problem. Indeed, the core functionalities which the legacy system provides do not change over time but rather the technology and platform on which it was developed is changed. From software evolution concerns, those core functionalities of legacy software should first be identified within the source code and separated from platform-specific code. This is a challenging task, since legacy Object-Oriented (OO) applications are not necessarily built applying separation of concerns [Wahler 2015]. Even more, a complete documentation of such legacy software might also be missing, thus making the evolution complicated.
In software development, SOA has advantages over object-oriented programming due to the dynamic discovery and flexible combining of its structural elements in heterogeneous computing environments. Moreover, the advent of Web services to realize SOA has offered a larger facility to design and implement flexible, interoperable and portable services. In this regard, re-engineering OO software to SOA and particularly implementing it as Web services has become a major topic of interest during the recent years. Additionally, re-engineering legacy software towards SOA contributes in building self-adaptive systems. In this context, several service identification solutions from object-oriented legacy source code have been proposed. These approaches rely on ad-hoc criteria; they lack a clear mapping between object-oriented concepts and service ones and thus fail to correctly identify relevant services. Moreover, several approaches perform the re-engineering manually or in a semi-automatic manner, thus, are considered as expensive solutions.

On the other hand, self-adaptive systems evolve during system's execution against changes in operating environment. In existing software applications that apply SOA architectural style, dynamic reconfiguration of structural elements is achieved in an ad-hoc manner. This implies that the reconfiguration of software system is often discussed only at implementation level. Such a late reconfiguration hinders the traceability of configuration aspects and consistency between different levels of the software development life cycle. However, there are some approaches that discuss reconfiguration issues at early stages of development process, for example at architecture or design levels. Those approaches lack of an explicit representation of configurable elements; they model configuration but not configurable elements. In

other words, these approaches do not provide explicit support for capturing variation of structural elements, consequently they do not represent variability at architecture level. This makes the configuration description hard to understand and the reconfiguration hard to implement. It also constitutes a barrier for a flexible management of reconfiguration at architecture level as well as traceability issues between a dynamic description given at the architectural level and its counterpart at other abstraction levels.
Architectural reconfiguration can be specified at architecture level using a syntactical expressive language such as Architecture Description Languages (ADL)s. Variability modeling is an excellent instrument to model variations of software artifacts and their behavior within a self-adaptive system. However, existing ADLs that support dynamic reconfiguration do not explicitly model variation points, on which the reconfiguration is based.

1.3 Thesis Contributions

In this thesis, we treat the problems identified in section 1.2 and propose the following contributions:

1. We propose an approach to migrate Object-Oriented (OO) legacy code towards Service-Oriented Architecture (SOA). Therefore, we propose solutions for service identification and packaging problems.

   (a) First, we propose a correspondence between object concepts in OO paradigm and service concepts in SOA paradigm. Meanwhile, we conduct a rigorous study built on service characteristics and then propose a semantic model that refines those characteristics into measurable metrics. Finally, those metrics are used to evaluate the quality of candidate services.

   (b) We apply a set of activities related to service packaging. First, we deploy identified services using a wrapping technique. Meanwhile, we annotate those services and finally generate a service interface that describes service's functionality.

2. To obtain a dynamic reconfigurable service-based self-adaptive system, we propose a modular Architecture Description Language (ADL) called Dynamic Service Oriented Product Lines (DSOPL-ADL).

   (a) In addition to specifying structural information, we distinguish three variability types and manage them at architecture level.

   (b) We enrich the static description by specifying reconfiguration aspects. This enables a self-reconfiguration of system at runtime in response to a context change.

(c) We propose a preliminary process to generate, among several variable configurations described in reference architecture, one concrete configuration based on context values. Furthermore, we automatically generate an executable implementation code from our architectural description.

We position these contributions within a re-engineering and engineering processes towards respectively SOA and SOA-based self-adaptable systems, as demonstrated in figure 1.1. Our contributions are annotated and highlighted in yellow boxes.

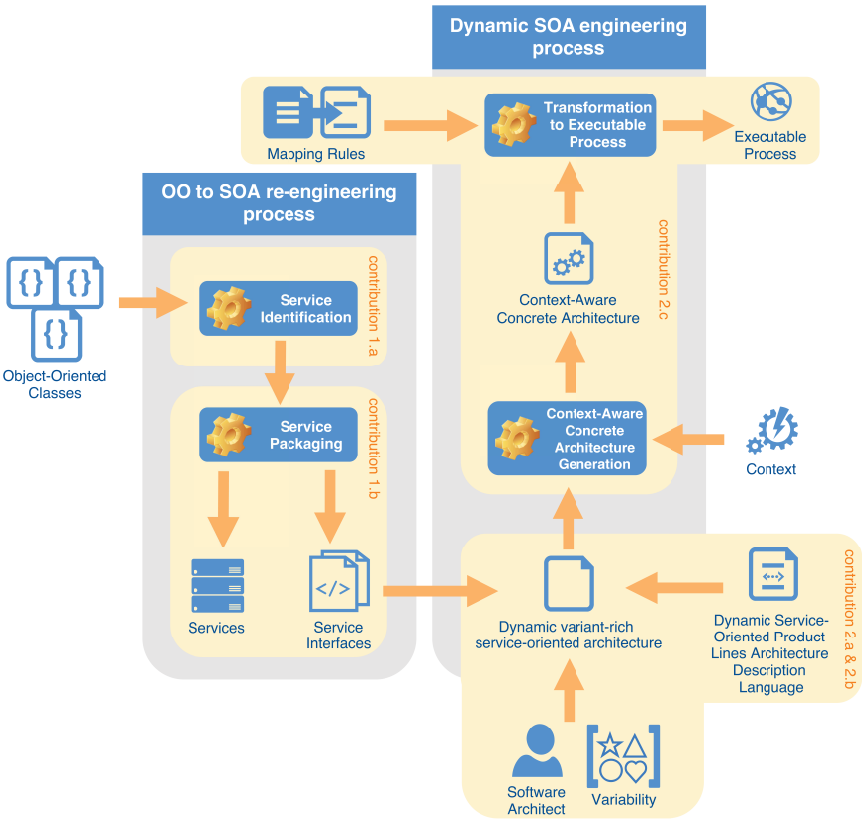


Figure 1.1: Positioning thesis' contributions within a re-engineering process towards SOA

1.4 Thesis Organization

The thesis is organized as follows. In **chapter 2 - State of the Art**, we present the main underlying concepts used in our work and investigate the related approaches that handle the problems discussed in section 1.2. In this regard, we propose a classification of SOA migration approaches mainly compared to service identification and packaging activities. We also propose a classification of ADLs compared

to their support to dynamicity and variability.

In **chapter 3 - Service Identification from Legacy Software Based on Quality Metrics**, we propose a migration approach towards SOA. Therefore, we propose a service quality measurement model in order to identify services in legacy object-oriented source code.

In **chapter 4 - Variable-Architecture Centric Reconfiguration of Service-Oriented Systems**, we propose a modular ADL that describes structural, variability, context and behavioral aspects of software architecture. The purpose of such exhaustive specifications is to enable a variability-based dynamic self-reconfiguration of software system at architecture level.

As part of validating our contribution, in **chapter 5 - Experimentation/ Validation**, we demonstrate some case studies on which we have evaluated our service identification and packaging approaches and interpret obtained results. We also demonstrate a concrete architecture generation as well as an executable process generation from our architectural description.

In **chapter 6 - Conclusion and Future Perspectives**, we resume the work realized in this thesis and give some perspectives and future directions.

Chapter 2

State of the Art

Contents

2.1 Outline

In chapter 1, we have presented the problem that we treat in this thesis. To provide a better understanding of this problem, in this chapter, we first present in section 2.2 the context as well as the main concepts related to the problem statement. In section 2.3, we present the related work mainly classified in two aspects: section 2.3.1 presents a classification of existing migration approaches towards Service Oriented Architecture (SOA). Moreover, we classify approaches that support service identification and service packaging activities. In section 2.3.2 we classify existing ADLs according to their structural description, their support to variability as well as dynamicity.

2.2 Context and Main Concepts

2.2.1 Legacy Software

Any software which has been developed using outdated technology [Sneed 2006], but still brings great value to the organization that uses it, is considered as a *legacy software* [Stehle 2008], [Sneed 2006]. [Sneed 2006] has even restricted the age of a software to turn into legacy software to five years. While current trend in building

applications is based on developing small entities which can be easily composed together and even reused in other applications, legacy software, in contrast, suffers from being formed of a single large block and often very complex to maintain. This is the reason of why upgrading a legacy system is considered a hard task. Overall, legacy software suffers from several disadvantages, such as:

- high cost of maintenance and upgrading
- lack of understanding
- lack of security
- difficulty to integrate with new systems that use recent technologies

However, despite above mentioned disadvantages, legacy software can neither be ignored due to its important business value nor easily converted to new technologies due to its complex infrastructure and its probably unstructured architecture [Cetin 2007]. Thus, a middle ground would be to re-engineer its architecture and put a migration plan towards a modern and flexible technology.

2.2.1.1 Evolution of Technologies

Software architecture methodologies are evolving in parallel to the growing size of systems. In figure 2.1 inspired from [Audrey 2008], we display the evolution of software architecture design methodologies. Among existing software architecture methodologies, we focus only on those who build complex systems from simple separate entities.

First, in late 1960s and in 1970s, **modular programming** methodology was common in programming languages such as Turbo Pascal, Ada, etc., where common functionalities were written in one module and later composed with other modules to build an executable application [Lindsey 1978]. The biggest challenge in this type of programming was how to manage parts of the code to render it reusable. Another disadvantage of modular programming was that each module was used at most once in the application and thus could have one single state.

To overcome the problems of modular programming, in 1970s, **Object-Oriented Programming (OOP)** paradigm was introduced and became widely used in 1980s and early 1990s in languages such as C++, Java and Delphi. The main features of this paradigm are object composition, class inheritance, abstraction, encapsulation and polymorphism [Cox 1986]. However, the main problem with OOP languages is related to interoperability and platform independence.

In **component-based development**, components are independently deliverable entities of functionalities [Szyperski 2002] that provide access through their interfaces [Brown 2000]. Independently deliverable entity means that it can be bought, downloaded and deployed as a standalone executable package of software. A component can also be subject to composition with other components. Most important characteristics of component are; reusability of software components in other systems, interoperability between several technologies and encapsulation (i.e. component acts as a black box, whose implementation is completely hidden behind a well-defined interface).

Finally, **Service-Oriented Architecture (SOA)** is a methodology for building systems, which recommends using independent components (named services), which communicate by exchanging messages, and whose interfaces are well-described in a platform independent manner [Kuba 2007]. SOA is not a totally new concept; it has been built as an evolution of other previous concepts such as component or even object. Further information of its characteristics and features is detailed in section 2.2.2.2.

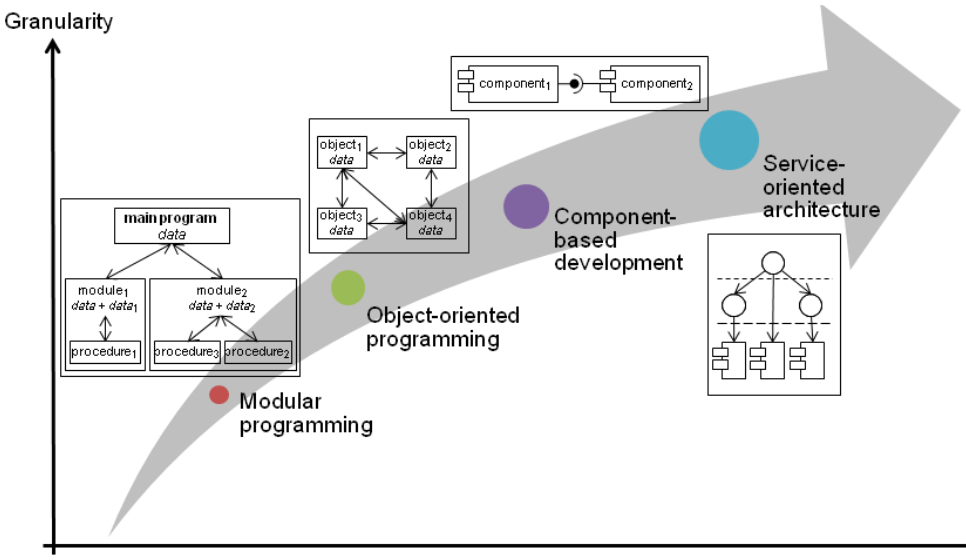


Figure 2.1: Evolution of software architecture design methodology

2.2.1.2 Legacy Software Modernization Towards SOA

The modernization of legacy software towards SOA is promising, given that SOA methodology allows the reuse of the core functionalities of the legacy software while

exposing larger visibility to clients through published and discoverable service interfaces [Khadka 2013b]. SOA concepts are given in more details in section 2.2.2.2. Several strategies have been proposed in literature in the context of modernizing an object-oriented system towards SOA. [Almonaies 2010], [Bisbal 1999], [Stehle 2008] and [Khadka 2013a] propose a classification of those existing modernization strategies. The conventional strategies of modernization are:

- **Replacement:** a complete rewrite of code from scratch is performed to replace the existing legacy code. Replacement is considered the least desirable solution for modernization to SOA [Almonaies 2010].
- **Wrapping:** an additional layer of interface wraps the legacy code and assures its accessibility by external entities. A significant work using wrapping technique was carried out in [Sneed 2006], which represents the functionality within the legacy code in form of Web services wrapped in XML shell. Wrapping, by far the most widely used modernization technique [Khadka 2013b], is considered to be as a quick and cost effective solution in case the code is relatively small but too expensive to completely re-write.
- **Migration:** a modernization technique that moves the legacy system to a more flexible and new environment while retaining system's data and functionality [Bisbal 1999]. It internally restructures and modifies legacy systems. This technique is further discussed in section 2.3.1.1.

It is worth noting that it does not exist any perfect solution for modernization [Almonaies 2010]. In some cases, more than one strategy could be chosen to modernize legacy software depending on the available resources, time and budget.

2.2.2 Dynamic and Variable Software Architecture

2.2.2.1 Software Architecture

Software architecture describes a computational system by specifying its structure which comprise software elements, interactions between them, and properties of both [Clements 2010]. A software architecture can be constructed from one or more viewpoints. Viewpoint signifies the perspective from which a view is constructed [Hilliard 1999]. Two well-known viewpoints that are used to describe a software architecture are **structural** and **behavioral** views.

1. **Structural viewpoint** defines the computational elements of a system and their organization [Oquendo 2008]. In particular, it describes the following information:
    - **Components:** The principal processing elements that comprise a system. Components might be services, processes, clients and servers, etc...

- **Connectors:** The interconnections among the elements of the system
- **Configuration:** The mechanism of composing components and connectors altogether

2. **Behavioral viewpoint** defines the functional aspect and the dynamic reconfiguration of the system. It allows specifying different operational modes for a dynamic system to adapt its behavior at runtime according to either environment changing conditions or different property values [Oquendo 2008]. Behavioral viewpoint is mainly specified in terms of:

   - **Action:** a fundamental unit in behavioral specification that comprises a set of activities that the system executes
   - **Runtime configurations:** different runtime configurations of architectural artifacts

Software architecture can be documented using several representations varying from informal notations (e.g. general purpose diagrams, views, natural language, etc.), textual languages (e.g. architecture description languages ADLs) or graphical representation (e.g. modeling languages) [Clements 2010]. A classification of architecture documenting notations are given in [Clements 2010]. We mainly classify these architecture documentation representations into two categories, syntactic expressive languages and graphical models:

- **Syntactic expressive languages:** usually specify software architecture using statements of a textual language. They describe and document software architecture at a high-level of abstraction rather than specifying implementation details [Vestal 1993]. Architecture Description Languages (ADLs) are feasible solution for such architectural documentation. ADL is a formalism that allows the specification of system's conceptual architecture [Medvidovic 2000]. It describes the high-level structure of the system rather than implementation details and techniques. It enables architects to describe and validate systems against stakeholders' requirements from one side, and ease the development and implementation process of complex systems, from another side. It often has a plain text syntax optionally accompanied with a graphical representation. Conventional ADLs support only static architecture description [Medvidovic 1996], [Deiters 2011]. Some ADLs provide a special formalism for SOA to describe service dynamicity [Jia 2007], [Oquendo 2008].

- **Graphical models:** use graphical notation or add meta-tags to already existing standardized models or views to represent major functional and non-functional requirements of the system [Abu-Matar 2011], [Niiyama 2008] and [Kruchten 1995]. Such models are used to describe a software system from the viewpoint of different stakeholders such as end-users, developers, or project

managers. Due to its understandable notation, this solution is usually preferred for communications with non-technical people (e.g. project managers, end-users).

The definition of software architectural concepts and behavior using ADLs is more precise and detailed, in addition to the fact that ADL renders automatic code verification during development easier.

2.2.2.2 Service Oriented Architecture

Several definitions to Service-Oriented Architecture (SOA) have been proposed in literature either by researches or standard organizations. [Footen 2012] defines SOA as *"an architecture of independent, wrapped services communicating via published interfaces over a common middle-ware layer"*. [Papazoglou 2008] defines SOA as *"a logical way of designing a software system to provide services to either end-user applications or to other services distributed in a network, via published and discoverable interfaces"*. Both definitions insist that SOA is not a product but rather an architecture style. One of the keys of applying SOA is that it adds a new layer of abstraction [Brown 2002] on top of existing layers in order to enable services to operate independently and in heterogeneous and distributed environment.

In addition to taking all aforementioned advantages of previous technologies which SOA was built on, we resume the main characteristics and features of SOA:

- **Service granularity:** SOA's main composing elements are coarse-grained and loosely-coupled autonomous services, thus modifying service's implementation does not require modifying other services as well as a service can be deployed by itself without other services [Brown 2002], [Clements 2010], [Zhang 2004], [Nakamura 2009], [Griffiths 2010].

- **Platform independent:** services in SOA communicate with each other through their interfaces and via standard messages. This standard messaging protocol [Papazoglou 2007] renders SOA a platform and language independent [Stehle 2008] architectural style.

- **Reuse:** SOA enables sharing services between several applications and consequently reduces development and maintenance costs [Griffiths 2010].

- **Composition:** system in SOA is built by aggregating and orchestrating multiple services in a distributed process [Papazoglou 2007].

- **Discoverable:** Services are discoverable at design-time as well as run-time [Brown 2002] by service brokers or end-user applications.

2.2.2.3 Variability

Variability Definition

Several definitions of variability have been given in the literature. According to [Galster 2011], variability is the ability of a software artifact to quickly change and adapt for a specific context in a preplanned manner. [Weiss 1999] defines variability as *"an assumption about how members of a family may differ from each other"*. [Svahnberg 2005] defines variability as *"the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context"*. [Pohl 2005] defines variability in time as *"the existence of different versions of an artifact that are valid at different times"* and the variability in space as *"the existence of an artifact in different shapes at the same time"*.

Variability is usually attached to Software Product Line (SPL) domain and is a core property to develop complex adaptable software systems such as telecommunication, pervasive, crisis management, surveillance and security systems. In such systems, due to environment changes, a dynamic re-configuration should be carried out without having to re-deploy the whole system. A great majority of approaches capture variability in a feature model, where commonalities and variabilities of a family of software products are modeled in addition to representing constraints and dependencies among those features [Lee 2012].

Variability Classification

Sources of variation vary and so do their representations vary. According to Bachmann [Bachmann 2001], variability has several sources: *variability in function*, where a particular function may or may not exist in a given product; *variability in data structure*, where a certain data structure may vary from a product to another; *variability in control flow*, where a sequence of control flow may change from a product to another; *variability in environment or technology*, where the operating system or hardware may vary, etc. [Svahnberg 2005] proposes another classification of variability; variability may occur at different levels: product-line level, architecture level, component level, sub-component level and code level.

Variability Representation and Management

Variability management is the key feature that distinguishes SPL engineering from conventional software engineering. It includes activities such as identifying, designing, implementing and tracing variable artifacts in product families [Voelter 2007]. It captures variant and common artifacts during software lifecycle and promotes the reuse of common assets across products to produce several distinct products of a SPL. Feature modeling is the most famous formalism for that purpose [Clements 2001]. It is a de-facto standard to model the common and variable features of a family of software products and their relationships [Kang 1990]. [Apel 2013] defines feature

as *"a characteristic or end-user-visible behavior of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle"*. Feature model is a tree-like hierarchy of features and constraints between those features [Czarnecki 2006]. Each feature (node), except root, has one parent and one or more child features. There are two types of relations between features: parent-child relationship and cross-tree constraints. In the first type of relationship, features might be either *mandatory*, *optional*, *alternative inclusive* (at least one variant should be chosen if parent feature is chosen) and *alternative exclusive* (only one variant among children must be chosen). As to the cross-tree constraints, *require* and *exclude* are typical examples of cross-dependencies between features. If feature A *requires* a feature B, this implies that feature B has to be included whenever feature A is included. If two features are in an *exclude* relation, this implies that only one of these features might exist in any valid product configuration.

Figure 2.2 depicts a simplified feature model of constructing a mobile phone. According to the example, any mobile phone must support calling feature and must have a screen. This latter can either be a basic screen, a colored screen or a high resolution screen. Furthermore, the mobile phone can optionally have a GPS navigation system as well as a multimedia support. As a media feature, any mobile phone can either have a camera or a MP3 or both. However, equipping the mobile phone with camera requires the screen to be a high resolution screen, since there is a required constraint from camera to high resolution feature. Finally, both features GPS and basic screen cannot be part of a same product since they are incompatible together (exclude constraint).

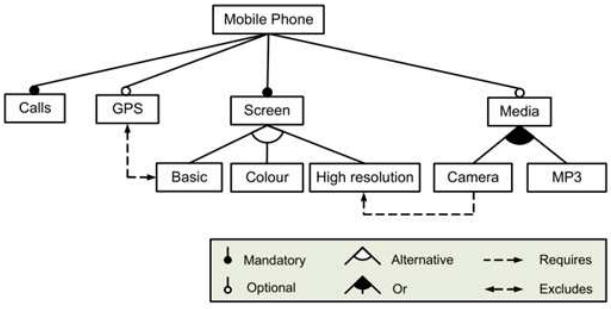


Figure 2.2: Feature model of mobile phone

2.3 Related Works

In the large context of service identification from object-oriented legacy applications and the representation of service-based dynamically reconfigurable and variable software architecture, we investigate and classify, in the following sections, existing relevant approaches. In section 2.3.1, we show the classical steps of object-oriented application migration towards SOA with a focus on the service identification and packaging phases. In section 2.3.2, we classify existing architectural representations in general, and ADLs in particular, mainly according to their structural descriptions and their support to dynamicity and/or variability.

2.3.1 Classification of Migration Approaches Towards Service- Oriented Architecture

Migrating legacy applications towards SOA allows systems to remain internally unchanged while exposing their functionality publicly through well-defined interfaces [Cetin 2007]. In this section, we demonstrate the related works for service identification and packaging within the context of migrating towards SOA and propose classifications of these works.

2.3.1.1 General Classification

Three methodologies, in general, are followed for creating SOA-based systems, as displayed in figure 2.3, inspired from [Audrey 2008]:

1. **Top-down SOA development:** which considers designing the target architecture and the orchestration of services having business rules and requirements as main input. This technique is advisable to apply for developing new systems since it does not consider the reuse of existing systems [Nakamura 2009].

2. **Bottom-up SOA development:** which adopts reverse engineering techniques and describes services in SOA by reusing at maximum existing software source code [Nakamura 2009]. Here, the input is existing legacy source code.

3. **Hybrid SOA development:** depending on available inputs and desired outputs, a combination of top-down and bottom-up methodologies can be adopted to achieve an improved service-based architecture. Here, existing legacy source code from one side and information of target system requirements from other side form the input of this type of migration towards SOA.

The migration process towards SOA can be seen as a re-engineering horseshoe model (see figure 2.4). The existing legacy system is represented on the left side of the figure and the target system on the right side. The model is also horizontally divided to two abstraction layers: implementation and design. In order to recover

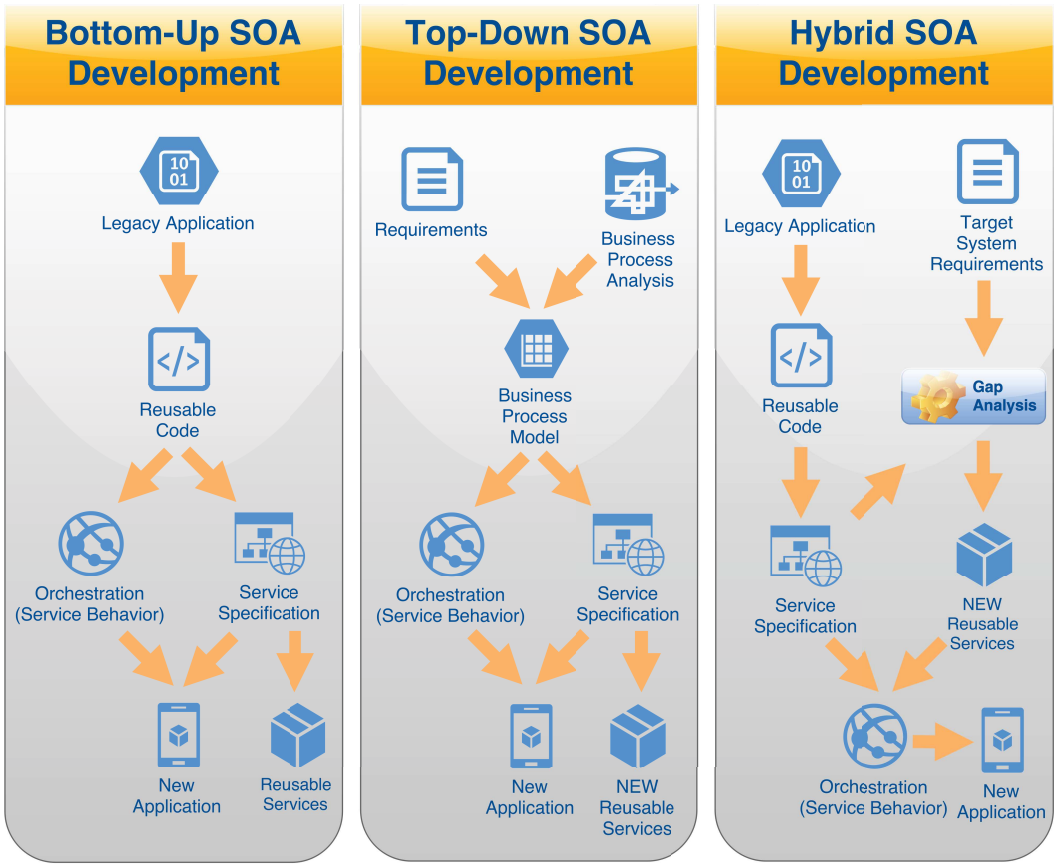


Figure 2.3: SOA development methodologies

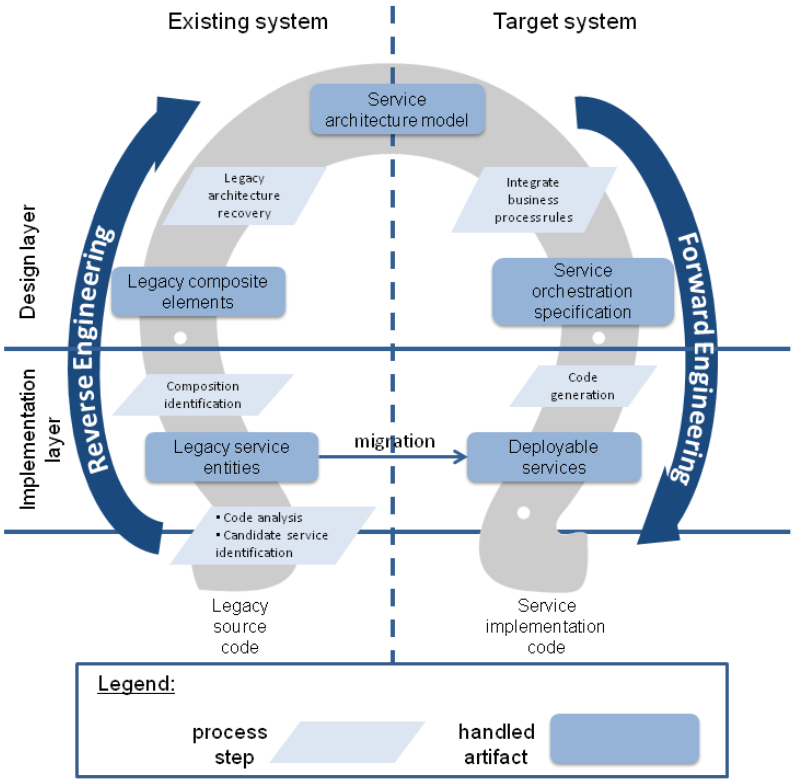


Figure 2.4: Legacy to SOA migration framework

the architecture of a legacy system, reverse engineering techniques are applied on legacy source code. First, existing legacy source code is analyzed and candidate services are evaluated to identify services. Then, the composition of those services are recovered and hence represented in an architecture model. On the side, in order to develop the architecture and generate deployable services and their orchestration, forward engineering techniques are applied. Architecture is restructured in terms of abstract services and some new requirements may be added.

Several approaches for legacy system migration towards SOA have been reported in literature [Sneed 2006], [Chen 2005], [Khadka 2013a], [Lewis 2005], [Khadka 2011], [Cetin 2007], [Channabasavaiah 2004], [Stehle 2008], [Matos 2009], [O'Brien 2005], [Zhang 2004]. Some of these approaches handle only the left part of the horseshoe process focusing on recovering the architecture of existing legacy software in term of services, whereas other migration approaches apply a full-circle re-engineering model. We mainly classify migration approaches into two categories:

- Migration approaches that are based on understanding and analyzing existing legacy system without considering target system requirements. These approaches follow the bottom-up SOA development methodology. [Matos 2009], [O'Brien 2005], [Nakamura 2009] and [Sneed 2006] are examples of such migration approaches.
- Migration approaches that are based on analyzing existing legacy software in parallel to understanding the requirements of the target system in order to match between existing artifacts and required functionalities and hence better guide the SOA system development. These approaches follow the hybrid SOA development methodology. Examples of such approaches include [Cetin 2007], [Chen 2009] and [Khadka 2013a].

Table 2.1 classifies existing migration towards SOA approaches. We demonstrate the different techniques used for both legacy and target systems' understanding. We also underline whether approaches that understand legacy system provide service identification and service packaging phases. Some approaches do not handle target system understanding, that is why their corresponding field of technique is marked with a '-'. [Khadka 2013b] summarizes existing legacy to SOA migration approaches through conducting a systematic literature review. Migration of legacy software towards a modern system is considered a hard and complicated task due to the lack of tools and available approaches to automate the process [Marchetto 2008]. We notice that the majority of migration approaches use the hybrid methodology, such as [Lewis 2005], [Cetin 2007], [Fuhr 2013].

In [Lewis 2005], authors present an initial migration approach called Service-Oriented Migration and Reuse Technique (SMART). It assists businesses to analyze the capacity of their legacy code that may be exposed as services in a SOA environment

by providing preliminary analysis of feasibility, strategy, cost and risk for the legacy migration to the SOA. It uses a hybrid migration methodology (combined top-down and bottom-up methodologies) to achieve the migration of legacy system towards SOA. The key activities of their approach are: identifying system's stakeholders, identifying expectations of future system and identifying migration concerns. Meanwhile, a list of candidate services is identified in existing legacy code. Finally, as a gap analysis, the results of those two phases are reconciled to identify a list of potential services. However, the proposed approach requires several sources of information (e.g. documentation) to support the analysis of the legacy system. Besides, the approach largely relies on human interaction; System analysts, maintenance programmers, etc. gather information through interviewing stakeholders in order to fill the gap between existing legacy system and target architecture. No technical details of legacy code extraction and service deployment is given in this approach.

An architecture-driven approach for migrating legacy systems to Service-Oriented Computing SOC, referred as mashup, has been proposed in [Cetin 2007]. This strategy consists of six steps: (1) model the target business requirements, (2) analyze existing legacy system, (3) identify services by mapping the target enterprise model to legacy components, (4) design concrete mashup server architecture, (5) define service level agreement, (6) implement and deploy services. This work is another good example of using hybrid migration methodology; top-down technique is used during the first step to analyze business requirements of the target system and model them, whereas bottom-up technique is applied in the second step to understand and recover valuable assets from existing legacy system.

In [Fuhr 2013], an architecture-based and requirement-driven service-oriented reengineering method is discussed, where services are identified by domain analysis and business function identification on the requirements abstraction level from one side and on the source code level from other side. On the source code level, legacy code is analyzed, architectural elements are identified and similar architectural elements are grouped into a component using hierarchical clustering algorithm. Later a matching is performed between business functionalities and legacy functionalities in order to determine the reusable legacy services. Since this approach is based on both requirements abstraction and source code levels, thus, it needs both architectural and requirement information to be available.

In order to evolve a legacy software towards a SOA, we have identified two major phases that are common in existing approaches that follow either bottom-up or hybrid SOA development methodologies: (1) *service identification* (or sometimes called *service extraction* or in more general term called *legacy system analysis*), where available software artifacts are analyzed to identify provided services and (2) *service packaging and deployment*, that leverages extracted legacy code as usable

services, wraps them by interfaces and orchestrates their operations.

Criteria/ Approach	Legacy system understanding				Target system understanding		Case study	Tool support
	Yes/No	Technique	Service identification	Service Packaging	Yes/No	Technique		
[Cetin 2007]	Yes	analyze legacy system, extract components and architecture	Yes	Yes	Yes	modeling business needs using BPMN	Yes	Yes
[Matos 2009]	Yes	source code analysis, service extraction and architecture representation	Yes	No	No	-	Yes	Yes
[Chen 2009]	Yes	static and dynamic source code analysis	Yes	Yes	Yes	Application domain analysis	Yes	No
[O'Brien 2005]	Yes	code analysis and architecture reconstruction	Yes	No	No	-	Yes	Yes
[Nakamura 2009]	Yes	reverse engineering legacy source code	Yes	No	No	-	Yes	No
[Sneed 2006]	Yes	data flow analysis	Yes	Yes	No	-	Yes	Yes
[Khadka 2013a]	Yes	reverse engineering	Yes	Yes	Yes	SOAP based web service	Yes	Yes

Table 2.1: Migration to SOA related work classification

2.3.1.2 Classifying Migration Approaches Regarding Service Identification

Many approaches have been proposed in literature to identify services by analyzing legacy software artifacts. The first major phase of migration is the identification of services in existing system. This phase becomes crucial, especially with the unavailability of certain resources (e.g. developers, architects) and poor documentation [Khadka 2013a], [Lewis 2005]. Even more, it is a challenging task, since legacy systems are not necessarily built with the vision of service. Service identification approaches mainly vary in terms of their source of information (input), service identification technique and degree of automation (human interaction). Table 2.2 summarizes some well-known existing SOA migration approaches with a particular focus on what service identification technique they apply.

As sources of information, several artifacts can be handled; Some researches rely only on source code while others need further artifacts such as documentation, system architecture or business requirements. Accordingly, legacy system understanding and modern architecture construction can be realized either by bottom-up reverse engineering techniques using source code as an input or a hybrid bottom-up and top-down techniques which is mostly the case [Lewis 2005], [Cetin 2007], [Fuhr 2013].

As to the source of information, we observe that most approaches assume the existence of large range of information about legacy systems such as their documentation, architecture and design documents [Lewis 2005], [O'Brien 2005], see table 2.2. Therefore, they are applicable to systems where such information is available. They cannot be applied to systems where only the source code is available [Nakamura 2009].

In regard to human interaction during the service identification phase, we observe that the majority of service identification approaches are carried out manually [Sneed 2006], [Lewis 2005], [Khadka 2011] as displayed in table 2.2. These solutions are considered as expensive in terms of expertise. Thus, some automatic or quasi-automatic approaches were proposed [Chen 2005], [Zhang 2005], [Chen 2009], [Matos 2009], [O'Brien 2005].

The main process in service identification is to evaluate candidate services. A detailed survey of all service identification methods is discussed in [Khadka 2013b]. In object-oriented legacy systems, candidate services are considered as groups of object-oriented classes evaluated in terms of development, maintenance and estimated replacement costs. There are several approaches to evaluate services. For example, [Sneed 2006] proposes an automatic approach to evaluate candidate services. It calculates a service's value based on cost analysis of the development costs, the maintenance costs, the estimated replacement costs and the annual business value contributed by that service.

Other service identification techniques propose to evaluate services either by code pattern matching and graph transformation [Matos 2009], feature location [Chen 2005] or formal concept analysis [Chen 2009]. In [Chen 2005] a feature location technique is proposed to identify features in the source code and to map them to services. It claims that services and features have many characteristics in common.

By observing existing service identification techniques and approaches, we notice that almost all existing approaches rely on ad-hoc criteria for evaluating candidate services. Therefore, they fail to identify relevant service. This results in a gap between identified services and expected ones.

2.3.1.3 Classifying Migration Approaches Regarding Service Packaging

Several terms are used to name the phase of deploying identified services, such as *service implementation*, *service packaging*, *service wrapping*, *determining service interface*, etc. However, the process is almost identical. Service packaging phase concerns with the deploying, describing and publishing activities of an identified service. Describing the functionality of the service serves to make the service visible

Criteria / Approach	Source of information (input)	Technique used for service identification	Case study	Human interaction
[Lewis 2005]	architecture data, design data, source code, interview stake-holders, etc.	high level requirement driven	pilot application of early version of SMART is applied at U.S. Department of Defense	manual (system analysts)
[Sneed 2006]	procedural source code	data flow analysis and code stripping (identifying variables and returned functions)	from roadmap to case study	automatic code extraction
[Zhang 2005]	source code	feature identification, Hierarchical clustering of components	Virtual Learning Environment Web-based system	human supervision for selecting cutting point in dendrogram
[Chen 2005]	object-oriented source code	feature analysis	library Management Information system	developers decide which identified classes to choose to generate Web services
[Khadka 2011]	source code	concept slicing, source code visualization, design pattern recovery	case studies in financial domain implemented in COBOL and C++	semi-automated service identification (manual investigation of source code)
[Fuhr 2013]	legacy code, business processes, architecture description model and interviews	domain analysis, business function identification and legacy code analysis and transformation to a TGraph representation (model)	GanttProject 2009	semi-automated (service designers for business modeling)
[Cetin 2007]	business requirements of target system and existing legacy components	N/A	financial gateway product line and black list management	manual
[Chen 2009]	source code, business requirements	formal concept analysis and ontology techniques for source code analysis	e-Workforce Management product originally implemented in C++ and later redeveloped in .NET framework	semi-automated
[Matos 2009]	object-oriented source code	annotating functionality in source code based on pattern matching rules, reverse engineering, graph transformation at architecture level, forward engineering	small banking application in Java	largely automated with some human interaction during code annotation phase
[O'Brien 2005]	source code, documentation, interviews	architecture reconstruction	Command and Control (C2) implemented in C++	automated
[Nakamura 2009]	procedural source code	reverse engineering, hierarchical data flow analysis	liquor shop inventory control system implemented in C	manual

Table 2.2: Service identification related work classification

to service consumers from one side, meanwhile, it serves to hide the service's internal implementation details from external clients.

In addition to the aforementioned phases which almost any service packaging technique performs, there are some additional improvements that can be carried out to the extracted services during service packaging phase. Examples of such improvements include refining the extracted reusable legacy code of the service extraction phase, constructing missing components and bridging legacy components to newly-built components [Zhang 2005].

[Khadka 2013b] has conducted a detailed literature review on service packaging techniques and tools. Among packaging techniques there are wrapping, code transformation, code generation and program slicing techniques. However, wrapping is the most widely used approach, where the functionalities of the legacy code are exposed through interfaces without altering nor transforming the legacy code to another language. There are several approaches or commercial tools that automatically wrap legacy code written in COBOL, PL/I and C++ without manual interaction. As for Web services, there exist some description languages and technologies that annotate Web services with ontologies e.g. OWL-S [Martin 2004], WSMO [Lausen 2005], WSDL-S [Akkiraju 2005] and SAWSDL [Farrell 2007]. Table 2.3 summarizes existing approaches that package extracted services within the process of legacy system migration towards SOA.

Approach	Packaging technique	Tool support
[Cetin 2007]	wrapping service, customizing existing components and develop new services	No
[Chen 2005]	wrapping service operations using delegation classes	Web Service Wrapper (WSW)
[Zhang 2006]	wrapping Web service and implementing a common interface	Java Native Interface (JNI)
[Sneed 2006]	wrapping	"Softwrap" tool
[Zhang 2004]	legacy code refinement, new service-oriented components integration, developing glue code and service complexity reduction	Axis SOAP processor

Table 2.3: Service packaging related work classification

2.3.1.4 Other Related Migration Approaches

In previous sections, we have presented related migration approaches towards SOA. However, other existing approaches can also be considered related to this problem, especially approaches focusing on migration towards component. The main difference behind both migration techniques relies on the difference between *service* and *component* concepts. Services in SOA and components in component-based archi-

tecture have several principles in common. Service-based systems and component-based systems are composed of services and components respectively that are interconnected to each other and can be decomposed to finer structural elements. However, there is a conceptual difference in designing both architectures. SOA aims at designing business processes and encapsulating them in services, whereas component-based development are implementation oriented which does not necessarily respects a given business rule.

Both services and components are self-contained and autonomous entities and whose functionalities are accessible through well-defined interfaces. In contrast to components, services are platform-independent entities that are distributed over network. Services are logical evolution of software components [Karastoyanova 2003] and middle-ware.

In fact, services are in a higher abstraction level than components [Tosic 2003]; services are built over an additional application architecture layer and components are the best way to implement those services [Brown 2002]. Likewise, Arsanjani et al. [Arsanjani 2007] define a nine-layer model for SOA called S3, where the *"services"* layer is above *"service components"* layer and followed by *"business process"* layer. A clear separation of concerns is implemented in their SOA solution, as demonstrated in figure 2.5, where three of those nine S3 layers are displayed. A *"service component"* is the realization of a *"service"* and represents the functionality of that service, whereas the *"service"* has a more abstract nature who exposes sufficient description about *"business process"*'s operation. Another difference between component and service is the instantiation time. While components are instantiated as needed, services are running instances that the client invokes [Brown 2002].

Despite the differences between service and component, services and software components have several characteristics in common, in particular, those related to their nature, structure and behavior. Both have the same main architectural properties; loosely coupled and coarse grained services (software components), interfaces and configuration (connection between architectural elements). Even more, Web service composition and component-based development have several practices in common [Iribarne 2004]. For that obvious reason, component identification techniques from object oriented legacy system could be considered as related to our research.

2.3.2 Dynamicity and Variability Representation and Management at Architectural Level

As we have mentioned in chapter 1, our goal is to propose a reconfiguration of system that comprises variability at architecture level. Being able to modify the architecture of a running system at such a high level of abstraction renders the system

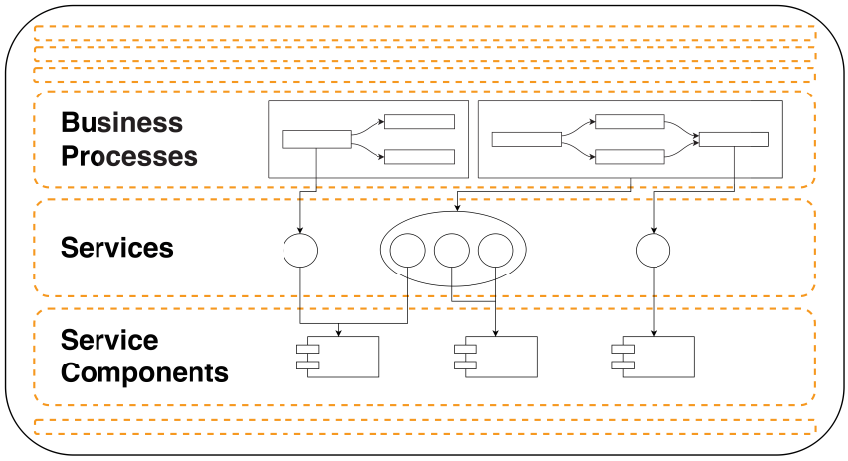


Figure 2.5: Some of S3 layers [Arsanjani 2007]

highly extensible, customizable and powerful [Medvidovic 1996]. For that reason, we present, in the following subsections, a classification of different approaches that handle dynamicity and/or variability issues at architecture level. In fact, both variability and dynamicity properties concern the architectural elements, that is why we first classify in section 2.3.2.1, existing architecture representations following their structural elements. Then, we investigate in section 2.3.2.2 some ADLs that provide special formalism to describe dynamicity. Likewise, we investigate existing ADLs that describe variability in section 2.3.2.3. In section 2.3.2.4, we investigate approaches that handle both variability and dynamicity issues.

2.3.2.1 Classifying Architecture Description Languages Compared to Structural Description

Regardless of whether existing ADLs in literature describe variability and/or dynamicity, all ADLs provide structural specifications of system's architecture that they represent. Table 2.4 lists the structural specifications of each existing approach. We also distinguish whether an ADL supports a composite hierarchical description of architecture or not. Traditionally all architectural descriptions have more or less the same structural specifications regardless of their names: composing element (component or service), provided/ required interfaces (or ports) and connectors (connection between those elements). Some architecture descriptions also specify hierarchical compositions such as [Magee 1995], [van Ommering 2000], [Barbosa 2011], [Jia 2007], [Medvidovic 1996], [Oquendo 2004] etc. Some ADLs, in addition to a syntactical expressive language, have a graphical representation to visualize the structural architecture. Table 2.5 lists some architectural descriptions, classifies them to ADLs and non-ADLs, indicates whether they have a graphical

visual support and classifies whether those representations are used for component-based systems or service-based systems.

Approach	structural specifications	composite element description
ADL		
Darwin [Magee 1995]	component, required/provided interfaces (called services), binding, component instantiation, hierarchy	Yes
KOALA [van Ommering 2000]	component, required/ provided interfaces, connects, configuration	Yes
Dynamic ACME	component, port (interface), connector, systems (configurations)	No
PL-AspectualACME [Barbosa 2011]	component, connector, role (provider/consumer), port, attachments	Yes
Dynamic-WRIGHT [Allen 1998]	component, port and role (as interface), connector, glue (as behavior), constraint	No
π-ADL for WS-Composition [Oquendo 2008]	service, connection, port	No
π-ADL [Oquendo 2004]	component, port & connection (interface), protocol, connector, architecture, behavior, compose	Yes
C2 SAD(E)L [Medvidovic 1996]	component, connector, port, topology	Yes
SOADL [Jia 2007]	service, provider/ requester port (interface), operation, message, behavior, sequence, receive/ send	Yes
xADL [Dashofy 2002]	component, connector, interface, sub-architecture, link	Yes
Plastik [Joolia 2005]	component, connector, port	No
non-ADL		
BPEL [BPE 2007]	service (partnerLink), interface (WSDL), operation, port	No
VxBPEL [Koning 2009]	service (partnerLink), interface (WSDL), operation, port	No

Table 2.4: Architecture's structural specifications' classification

Criteria/Approach	graphical visualization	main structural element	
		service	component
ADL			
Darwin [Magee 1995]	Yes	No	Yes
Koala [van Ommering 2000]	Yes	No	Yes
PL-AspectualACME	Yes	No	Yes
Dynamic Wright [Allen 1998]	Yes	No	Yes
Rapide [Luckham 1995]	Yes	No	Yes
Plastik [Joolia 2005]	No	No	Yes
π-ADL [Cavalcante 2015]	Yes	No	Yes
SOADL [Jia 2007]	Yes	Yes	No
π-ADL for WS-Composition [Oquendo 2008]	Yes - BPMN	Yes	No
π-ADL [Oquendo 2004]	No	No	Yes
C2 SAD(E)L [Medvidovic 1996]	Yes	No	Yes
xADL [Dashofy 2002]	Yes	No	Yes
non-ADL			
[Abu-Matar 2011]	Yes	Yes	No
BPEL [BPE 2007]	Yes	Yes	No
VxBPEL [Koning 2009]	Yes	Yes	No

Table 2.5: Classifying related works according to their nature and main structural element

2.3.2.2 Classifying Architecture Description Languages Supporting Dynamicity

Static versus Dynamic ADL

A software architecture can be classified in terms of its capability of evolution into two categories: **static** and **dynamic** [Oquendo 2008]. A static architecture specifies system's structure at design time. Traditional static ADLs describe in particular the set of *composing elements* that encapsulate a functionality and their *connectors* that coordinate the communication between those composing elements.

While ADLs have more or less agreed on what elements to represent regarding structural specifications, there is not yet a common agreement of what dynamic ADLs shall represent from behavioral point of view. It may happen that software architecture evolves after its deployment [Clements 2010]. Such architecture is called dynamic architecture. Several different definitions of dynamic architecture have been proposed in literature. For example, [Bradbury 2004] considers that dynamic software architecture modifies itself and adopts modifications during system's execution. In Rapide language [Luckham 1995], one of the earliest ADLs that tackle dynamicity, dynamic architecture has the capability of modeling an architecture in which the number of components, connectors, and bindings may vary while system's execution. Dynamic architectures, in addition to specifying the system in terms of components, connectors and configurations, they should also specify how these components and connectors are evolved or reconfigured at architectural level during system's execution. Defining those specifications in an ADL is considered a challenging task. Having dynamic architecture is considered crucial in several domains such as in air-traffic control, high safety-critical systems, etc. where stopping, reconfiguring and then restarting the system may cause catastrophic effects. Hence the importance to modify the architecture during system execution.

Dynamicity Management Types

It is evident that dynamicity is differently considered and perceived in different research communities, hence the importance to classify those literature works according to our own understanding and in accordance to our contribution. We mainly classify dynamic architecture descriptions, whether described in ADLs or other formalisms, into two types:

1. **centralized dynamicity management:** where all instructions of modifying system's architectural behavior are defined in a central configurator. Hence the behavioral description is independent from architectural elements' functionality definition. Various approaches have emerged to explicitly describe the interaction between architecture's structural elements in form of a sequence of activities such as in [Oquendo 2008], [Jia 2007], [BPE 2007].

2. **event-driven dynamicity:** where constraints in form of triggers or events are defined inside each architectural element of the ADL. Here, an internal observer listens to environment's changes and modifies elements' behavior (e.g. its connection with other elements) only if a pre-defined constraint or condition is satisfied. Darwin [Magee 1995], Plastik [Joolia 2005] and Dynamic Wright [Allen 1998] are examples that use this technique.

In global, the reconfiguration of architecture at runtime may happen through several dynamic actions:

- creating (instantiating) / removing an architectural elements from the architecture
- binding / unbinding architectural elements to the architecture
- reconfiguring architecture (modifying connections between architectural elements)
- upgrading existing architectural elements (substitution of architectural elements)

Dynamic Component-Based ADL
Among existing ADLs in the literature, only few of them support dynamic reconfiguration such as C2 SAD(E)L [Medvidovic 1996], Darwin [Magee 1995], π-ADL [Oquendo 2004], Rapide [Luckham 1995], Plastik [Joolia 2005] and Dynamic Wright [Allen 1998]. We classify in table 2.6 existing dynamic ADLs according to their support of the aforementioned dynamic actions. [Minora 2012] investigates four ADL's support to dynamic reconfiguration. These languages are: π-ADL, Plastik, C2 SAD(E)L and Dynamic Wright. It differentiates between foreseen reconfiguration and unforeseen once. The foreseen reconfiguration is programmed at design time but executed at runtime, whereas unforeseen reconfiguration concerns an ad-hoc and unplanned modification of architecture at runtime.

Our conviction is that component and services as structural entities have several principles in common. That is why during related work classification, we study the dynamicity (behavioral) aspects not only in service-based ADLs but also in component-based ones, such as Rapide, Koala [van Ommering 2000] and Dynamic Wright [Allen 1998], Darwin [Magee 1995]. Following, a brief description of how each ADL tackles dynamicity.

Plastik [Joolia 2005] has the following structural elements: *component*, *connector* and *port*. As to dynamic elements to describe behavior (a specific configuration), the expression *"on condition do operations"* is used to toggle between different choices at runtime. To replace an instance of component at runtime, `detach` and `attachment`

reference dynamic action	**Darwin** [Magee 1995]	**Dynamic Wright** [Allen 1998]	**SOADL** [Jia 2007]	**π-ADL** [Oquendo 2004]	**C2 SAD(E)L** [Medvidovic 1996]	**π-ADL for WS-Composition** [Oquendo 2008]	**Plastik** [Joolia 2005]
create architectural element	Y	Y	N	Y	Y	N	Y
remove architectural element	N	Y	N	N	Y	N	Y
bind architectural element to architecture	Y	Y	Y	Y	Y	Y	Y
unbind architectural element to architecture	N	Y	Y	N	Y	N	Y
reconfigure architecture (modify connections)	N	Y	Y	N	Y	N	Y
substitute architectural element (upgrade)	N	Y	Y	N	Y	N	Y

Table 2.6: Supported dynamic actions in existing dynamic ADLs

statements are used in operations' part in order to respectively unlink and link components and thus replace an instance of component at runtime.

Darwin [Magee 1995], one of the earliest languages addressing dynamicity aspect in ADL, is a configuration language that models in addition to static structure (i.e. component, interface, binding and component instantiation / composition) also some properties of dynamic architectures. It offers component dynamic instantiation and binding facilities but does not handle the creation or destruction of connections between component instances. It also uses the subclass concept to build more specific classes from generic ones. Operation model is described in π-calculus.

Dynamic Wright [Allen 1998] supports the description of architecture from both structural (static) and behavioral (dynamic) viewpoint. Its structural description contains the following elements: `component`, component's interface named `port`, `connector` and connector interface (`role`). As to dynamicity specification, system's behavior is specified separately in a `configuror`. Configuror is in charge of reconfiguring architecture's workflow by using `attach` and `detach` instructions. Configuror is composed of two sections:

1. An *initialization* section, where initial structural elements are instantiated using `new` and `attach` instructions followed by a definition of an initial sequence of actions.

2. A *reconfiguration* section that contains several alternative configurations. One of those alternative configurations are executed if its constraint is satisfies.

Koala [van Ommering 2000] is also an example of a component model, where all run-time reconfigurations are predefined at design-time. The dynamicity of this language is restricted only to *"switching"* between components according to predefined rules in order to bind selected component at run-time.

Dynamic Service-Based ADL

All ADLs that were previously detailed in this section are ADLs that describe a component-based architecture. There are also several ADLs that handle service-based architectures. For example, π-ADL for WS-Composition [Oquendo 2008] is a service-oriented ADL for Web Service (WS) composition that has the same roots as π-ADL [Oquendo 2004] and highly relies on BPMN's visual notation. It formally describes service-oriented dynamic architectures from both structural and behavioral viewpoints. π-ADL for WS-Composition is considered as dynamic ADL because some third party services can be discovered and bound to service broker at runtime while some other services are already bound at design-time. The definition of the architecture is divided in two parts:

- *Behavior*, where the instances of `components`, and `connectors` are defined abstractly and also the link between each component and connector
- *Structure* definition, where each component is defined (e.g `ports`)

Another example of service-based ADL supporting dynamicity is SOADL [Jia 2007], a service-oriented architecture description language which is used for modeling service-oriented architecture in an abstract level. Technically, SOADL adopts XML notation and is therefore independent of the platform and technologies. It specifies the architecture in terms of services, interfaces, behavior, semantics and quality properties. It also supports architecture-based service composition. By observing the pseudo-schema syntax of SOADL, we can distinguish four main parts:

- **Port** is the interaction point of service. It plays a provider or requester role
- **Behavior** consists of a sequence of actions, either a basic action or a composite one
- **subArchitecture** part describes the structure of the (sub)system of a composite service. More precisely this part includes three parts:
  - **Dependency** part declares local or external service types that the (sub) system may use
  - **Configuror** part specifies all possible configurations for the given system. Each configuration is triggered by an event. However, all configurations treat foreseen events (i.e. unforeseen events are not discussed).

  - **Constraint** part defines a set of temporal constraints between operations in one or more ports. It determines how an architectural design is permitted to evolve over time.

- **Properties** part describes properties of security, transaction, load balance, version, or information related to implementation

However, in SOADL the dynamic reconfiguration of services discusses only the substitution of service instances in case of unavailability of a main service. Substituting services are statically defined at design-time in the *configuror* part of *SubArchitecture* element.

All previously mentioned approaches provide certain dynamicity according to either planned (predefined) or unplanned changes in a given architecture [Oquendo 2008]. However, there are other paradigms than ADLs that tackle dynamicity particularly in Web service composition. In order to describe the composition of Web services and to create executable business processes, many languages have been proposed in literature. Among them, BPEL4WS (or referred as BPEL) [BPE 2007], an OASIS standard executable language based on XML notation for specifying executable and abstract business processes. A process is the ordering of activities, it has inputs and provides outputs. The composition of Web services is called `"process"` which contains a set of `"activities"` that communicate with each other through `"messages"`. The involved services in BPEL process are called `"partners"` which are invoked through their WSDL interfaces. In BPEL, it is possible to define variables, create loops and conditions, create parallel or sequential activities and assign values. BPEL has two types of activities: primitive and structured. Primitive are single activities such as `assign, receive, invoke, reply`, while structured instructions (e.g. `sequence, flow`) regroup several primitive activities.

2.3.2.3 Classifying Architecture Description Languages Supporting Variability

The notion of variability in the context of software architecture seems to be poorly discussed in literature. In software architecture and its representation, variability management is not often explicitly described, on the contrary to product lines domain, where variability is a first-class concern [Galster 2011]. Only few existing approaches were concerned about representing an architecture that encompasses variability [Nakagawa 2012] at architectural level such as [Dashofy 2002], [van Ommering 2000], [Zhu 2011], [Barbosa 2011], [Capilla 2014], [Abu-Matar 2011]. Among these approaches, [Dashofy 2002], [van Ommering 2000] and [Barbosa 2011] integrate variability notions directly within their proposed ADL, while other approaches manage runtime variability at architecture level in general.

Among existing ADLs that handle variability, xADL [Dashofy 2002] is an ADL for modeling runtime and design-time architectural elements of software systems. It is defined as a set of XML schemas. This gives xADL a full extensibility and flexibility, as well as basic support from many available commercial XML tools. xADL 2.0 integrates product lines concepts in the form of three schemas: *versions*, *options*, and *variants* schemas. Concerning the integration of product lines concepts within xADL; this approach suffers from the limitation of expressing constraints (i.e. requires, excludes) between elements of different variation points.

Koala [van Ommering 2000] is a component model with an architecture description language that supports product-line modeling by modeling variation points in architecture. Inspired by Darwin [Magee 1995] language, and implemented in C. Its main elements are *interfaces (provided/ required)*, *components* and a *configuration*. Its dynamic reconfiguration is limited to using a `switch` to bind a component's interface to the system based on a statically defined condition. The main limitation in Koala is its static nature; any deployed configuration cannot be changed at runtime and will require application recompilation, thus it is not suitable for dynamic architectures.

PL-AspectualACME [Barbosa 2011] enriches Aspectual ACME description language by adding a variability dimension description at architecture level. Structural elements are described in terms of `type of components`, `connectors`, and `ports`. Variabilities are modeled using `representation` elements for identifying product variations, whereas `port` elements are used for representing the mechanism of variability selection. Features are described as `component Type` elements.

In a related context, Dynamic Software Product Line (DSPL) extends conventional SPL perspective by delaying the binding time of product's composing elements (i.e. features) to runtime, a feature called late variability [Baresi 2012]. It produces autonomous and reconfigurable products that are able to reconfigure themselves to select a valid configuration during runtime [Cetina 2008]. Even though there is no concrete agreement of what aspects a dynamic SPL should exactly treat, most approaches agree that the main characteristic of any dynamic SPL framework is the runtime variability, which provides the following common activities at runtime:

- managing the dynamic selection of variants
- autonomous activation/ deactivation of composing elements
- substitution of composing elements
- dependency and constraint checking of changed elements [Capilla 2014]

Except previously described approaches that address variability at architecture level,

we notice that variability management is not often described neither in the context of service-based systems nor at architecture level, therefore we investigate at other levels of abstraction how variability is described. For example, in implementation (business process) level, [Koning 2009] extends the process description language grammar in BPEL to provide explicit variability support. New elements are added to BPEL to support the dynamic reconfiguration of variants during system's execution. Those elements are `variation points` to indicate the place where an adaptation may occur and `variants` which describe a BPEL activity that will be executed if a variant is selected. It is worth noting that VxBPEL supports several variability actions in particular describing service replacement and the possibility to modify system's composition at runtime. However, it does not provide any mechanism to check constraints among different variants or different variation points.

2.3.2.4 Classifying Architecture Description Languages Compared to Variability and Dynamicity Support

So far, we were mainly interested in classifying Architecture Description Languages (ADLs) according to their support to dynamicity or variability. However, we have also noticed that the reconciliation between SPL and SOA to model software architecture could have a different nature than a syntactical expressive language (e.g. ADL). In this section, we present other approaches that treat dynamicity and/or variability. However, these existing approaches have different nature than an ADL.

Variability modeling of service-family architecture is not necessarily always expressed in an ADL. For example, [Abu-Matar 2011] presents a service variability model by applying SPL concepts to model SOA systems as service families. It integrates feature modeling with service views using UML and SoaML. In this approach, feature modeling is the unifying view that provides added dimension to the variability in service-oriented product line architecture. The multi-view SOA variability model consists of two requirements views (service contract and business process) and two architectural views (service interface and service coordination). Each view is modeled using an UML diagram which is extended by stereotypes to express variability notions. Unfortunately, the repartition of information in multi-views renders it difficult to convert it to a formal language that can be converted to executable system.

[Zhu 2011] proposes a model of product line architecture. It describes variability at architecture level using the following elements: *components*, *connectors*, *interfaces* and *links*. Variability in product lines architecture is usually represented by optional and alternative architecture elements. However in this approach, regarding alternative elements, it discusses only components' alternativeness. Rather than representing architecture-level variability of each architecture element sepa-

rately (fine-grained variability), it identifies a configuration of variation elements and groups them as a bigger grain variation constructs.

2.3.2.5 Summary of Architecture Description Classifications

First, we have classified existing architecture description approaches to syntactical expressive languages (i.e. ADL) and graphical models. We have also classified these architectural descriptions regarding to their support to dynamicity and variability. Concerning architecture descriptions (whether as an ADL or other formalisms), we have noticed that the level of dynamicity varies ranging from only binding an architectural element at runtime up to specifying a complete dynamic behavior of the architecture where structural elements can be bound/ unbound, and the whole architecture can be reconfigured at run-time without the need to re-compile the system. In what concerns variability, we have noticed that only few number of works were interested in describing it at architecture level and as an ADL.

We have also noticed that most existing works use components as a main composing architectural element. We could only find few service-based ADLs. Basically all ADLs describe the structural specifications of the architecture before treating variability or dynamicity aspects. Dynamic and variability aspects may either be embedded in the structural specification of the architecture or it can be specified in a separate section assuring the concept of separation of concerns. For example, for dynamicity specification, in the first case, each architectural composing element is a self-managing entity which is responsible for its connections to other entities. Whereas in the second case, there is a orchestrator which is in charge of communication between several entities. In is worth noting, that one of the advantages of separating the behavior specification from the structural specification is the possibility to define more than one configuration for the same set of structural elements.

2.4 Conclusion

In this chapter, we have first presented existing approaches in relation to SOA migration. We have observed that a migration process towards SOA goes through two main phases: service identification and service packaging. As to the **service identification** phase, we have found a lack of using SOA quality properties to guide the selection of good candidate service. Existing approaches have either used ad-hoc criteria to evaluate candidate services or candidate services were extracted depending on previous knowledge on expected software services and their functionalities. Ad-hoc means that characteristics of services are not used to identify relevant services. As to **service packaging** phase, some migration approaches do not handle the deployment and packaging of identified services. Existing packaging approaches widely use wrapping technique to expose service's provided functionality.

Second, from the point of view of an architecture and its representation, we have classified different approaches according to their support to dynamicity and variability. Figure 2.6 summarizes studied ADLs by distinguishing them into two major classifications, their **support to dynamicity** and their **support to variability**. Even more, among ADLs that support dynamicity, we distinguish two groups, those who consider **service as a main structural element** and those who handle **other forms of structures** (often components). Approaches that handle services as a main architectural element are considered dynamic, since services are dynamic by nature. Nevertheless, these ADLs are not able to describe service variants. From another side, existing approaches that describe architectural elements' variations at architecture level such as xADL [Dashofy 2002], Koala [van Ommering 2000], etc. are not based on service-oriented systems. Approaches that reconcile SOA and SPL were also studied, but those approaches were not designated at architecture level, but rather at requirement level.

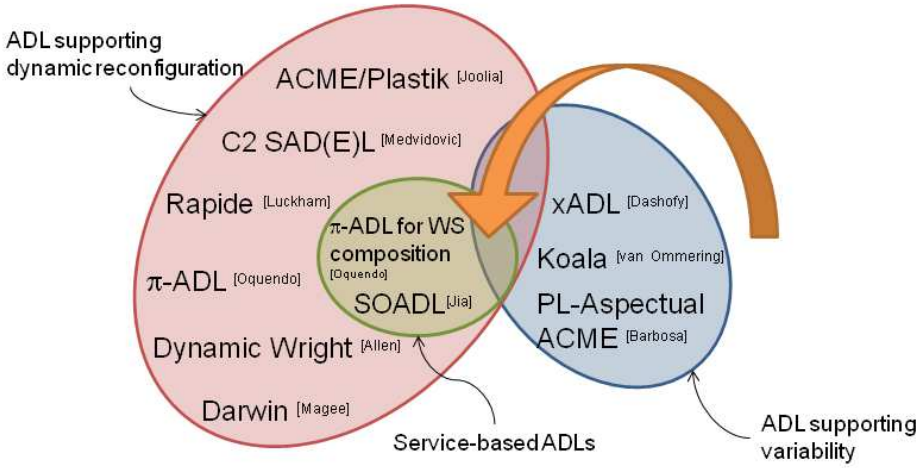


Figure 2.6: Panorama of existing ADLs

To resume, we have not found any approach that handles dynamic reconfiguration of service variability at architecture level hence the idea to propose such an architecture description language in chapter 4.

Chapter 3

Service Identification from Legacy Software Based on Quality Metrics

Contents

3.1 Introduction

Service Oriented Architecture (SOA), as a design philosophy, fulfills the requirements of modern systems, such as providing encapsulated and loosely coupled business units, which can be dynamically bound or unbound to the system at runtime. Moreover, services, which are the fundamental building blocks of SOA, are independently developed but can be flexibly composed with each other.

Unfortunately legacy object-oriented software cannot be blindly transferred to SOA paradigm. As a consequence, a migration towards SOA is the best way forward to

follow new technological advances and yet to conserve the business value of existing legacy object-oriented software. One of the most common approach to realize SOA is to implement it using Web services.

Our contribution in this chapter is a migration approach that comprises of two main phases: service identification and service packaging. Our **service identification** proposition automatically identifies services as groups of classes from legacy software object-oriented source code. We base our legacy system analysis on the source code, since it is the only resource that is always available, while other resources such as documentation or software architect could often be missing. Unlike other existing approaches that identify candidate services in source code manually or in an ad-hoc manner, we propose an automatic identification method of candidate services. In our approach, we refine well-known service characteristics to measurable metrics and define a fitness function that measures semantic correctness of each group of source code elements to be considered as a service.

Service identification is followed by a set of processes that are regrouped in a **service packaging** phase. This phase serves as a preparation to make services deployable. It includes activities such as making each cluster become interface-based by raising the dependencies between classes into dependencies between clusters that communicate via their interfaces. Meanwhile, we apply an annotation algorithm to name identified clusters. Finally, an interface is generated for each class which exposes services' functionalities.

This chapter is organized as follows: In section 3.2, we present a mapping model between concepts of Object Oriented Programming (OOP) and Service Oriented Architecture (SOA). In section 3.3, we specify service characteristics and refine them to measurable quality metrics to evaluate potential services. We also group similar classes to form coarse-grained and loosely-coupled services. In section 3.4, we present the packaging steps towards deploying those identified services.

3.2 Object-to-Service Mapping Model

In order to be capable to identify services from object-oriented source code, we define a mapping between object-oriented and SOA concepts as presented in figure 3.1. We consider a *service* as a group of *classes* defined in object-oriented source code. Among these classes, some define the operations provided by the service, whereas others are internal classes. *Internal classes* are those which only have internal connections to other classes of the same service. Classes that define the operations provided by the service are the classes that define its *interface*. Internal classes do not define operations provided by the service. *Operations* provided by

the service are class's *public methods*.

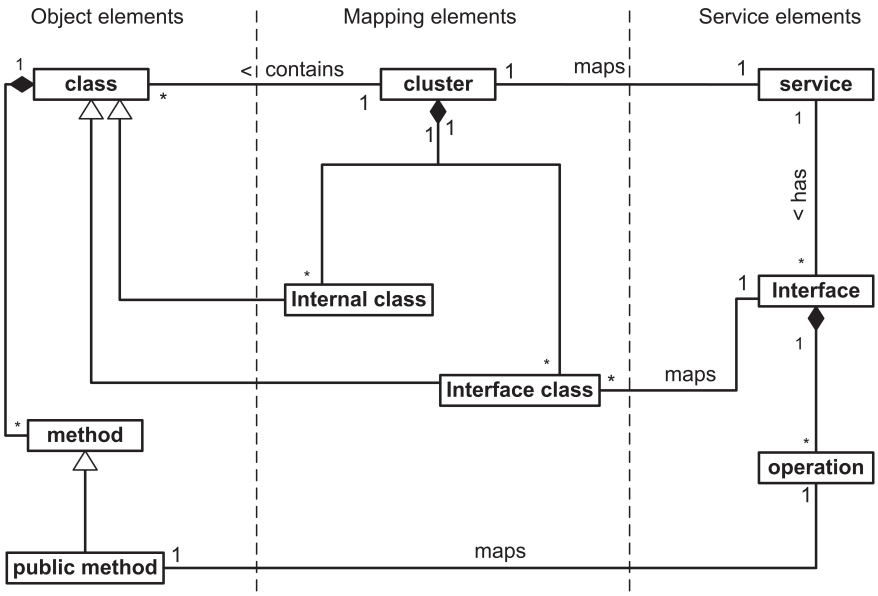


Figure 3.1: Object to service mapping model

3.3 Quality Measurement Model of Services

As we have mentioned earlier, a service is identified from a group of object-oriented classes. Initially, each group of classes is considered as a candidate service. A qualified service is selected from candidate ones based on a function that measures its quality. Diverse studies have been proposed in literature for measuring qualitative properties of SOA systems. Most of these works either assess systems that are already service based or evaluate systems only after their implementation. Unfortunately, such approaches are not adapted to the context of re-engineering an object-oriented system towards service oriented system. For example, [Aldris 2013] proposes a framework to measure the degree of service orientation in SOA systems. It focuses on the internal SOA attribute, decomposes a selected attribute to a set of factors and maps each factor to a set of measurable criteria. Each criterion is typically evaluated by a set of software metrics, though no dedicated metrics are defined for each criterion.

As to our approach, we adopt the ISO standard for software quality ISO/IEC 25010:2011 [ISO 2011] to evaluate identified services. ISO/IEC 25010 has defined

eight software product quality characteristics which are refined to thirty-one sub-characteristics, as demonstrated in figure 3.2. Each sub-characteristic is further divided into properties. These properties are attributes which can be measured or verified for any software product evaluation. Likewise, we define a quality function of services based on a set of service characteristics that are mapped to a set of properties. Each property is later measured using a set of metrics. In the next subsections, we will study different service and Web service characteristics, classify them, refine them to properties and then into measurable metrics. These metrics will form a fitness function by which candidate services will be evaluated. Finally, similar classes will be grouped in a cluster using a clustering algorithm.

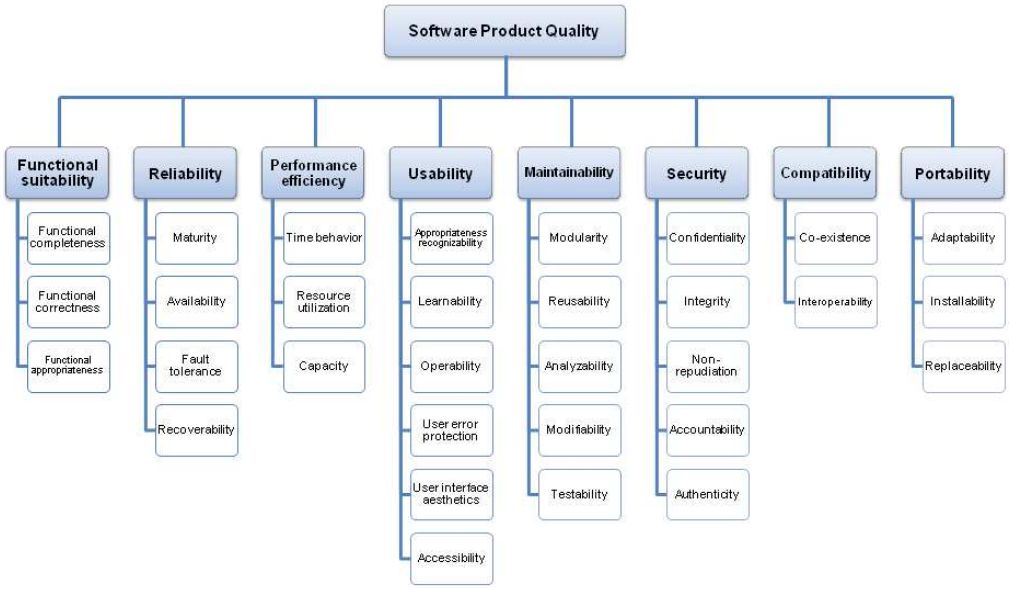


Figure 3.2: ISO/IEC 25010 software product quality model

3.3.1 Characteristics of Services

We deduct the quality characteristics of services based on the analysis of the most commonly used definitions of services in literature.

In literature, there are several definitions of services [Brown 2002], [Zhang 2005], [Nakamura 2009]. According to [Zhang 2005], a service is an abstract resource that performs a coherent and *functional* task. [Nakamura 2009] considers a service as a process that has an open interface, *self-containedness* and *coarse granularity*. It can be easily *composed* and decomposed to implement various business workflows. W3C [Booth 2004] defines a service as "an abstract resource that represents a capability

of performing tasks that represents a *coherent functionality* from the point of view of provider entities and requester entities". [Brown 2002] defines the service in terms of its characteristics: A service is a *coarse-grained* and *discoverable* software entity that interacts with applications and other services through a *loosely coupled*, often *asynchronous*, *message-based* communication model.

- ***Coarse-grained*** means that services implement more than one functionality and operate on larger data sets.
- ***Discoverable*** means that services can be found at both design time and run time, not only by unique identity but also by interface identity and by service kind.
- ***Self-contained*** refers to the self-sufficiency a service has, where context or state information is not required from other services.
- For ***loosely coupled***, services are connected to other services and clients using standard, dependency-reducing, decoupled message-based methods such as XML document exchanges.

3.3.2 Characteristics of Web Services

The most common form of realizing a SOA system is via Web services [Lewis 2005]. Web services are special types of services that are built via XML grammar in order to expose their functionality over Internet (or private network). Web services use a standard XML messaging system to communicate to each other, therefore they are independent from any operating system or programming language. All the characteristics mentioned for services in the section 3.3.1 such as *loosely-coupled*, *self-contained* and *coarse-grained* entities that interact *dynamically*, apply to Web services. However, Web services have some other specific characteristics which worth to be studied. [Papazoglou 2008] defines Web service as *"a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application."* Web services also have the following characteristics:

- ***state*** property: Web services could either be stateful or stateless. Stateful services maintain state information, whereas in stateless services, the Web service does not have any memory to preserve state information. It simply performs a requested operation without keeping any track of that invocation.
- ***synchronization***: we can distinguish between two principal messaging styles among Web services: synchronous and asynchronous; Synchronous is a request-response operation. In synchronous communication, the client sends its request as a method call with a set of arguments and the synchronous Web service responds in a return value. The client requires an immediate response

from the other service. Whereas in asynchronous communication, client invokes a Web service and does not wait for the response. Once the latter completes processing, it sends the result to the client. Asynchronous communication is a key factor in enabling loosely coupled services.

- ***well-definedness***: The functionalities which a Web service provide are described in a service interface using Web Service Description Language (WSDL). In addition, this description specifies the rules of how to interact with the service. Service interface will render service's functionality visible to external services without the need to expose its internal implementation details. So in Web services there is a clear distinction between service's interface and its implementation.

3.3.3 Service Characteristics Classification

Table 3.1 lists the characteristics of services as mentioned in the definitions above. We have categorized them into two categories: those related to the structure and behavior of services and others related to the SOA platform. Structural and behavior characteristics (such as coarse-grained, loosely-coupled and composable) reflect the semantic properties of service and thus they could be measured in object-oriented legacy source code, whereas characteristics that depend on SOA platform (such as discoverable) do not reflect any semantic property of service. Thus we could not base our identification metrics on that type of characteristics. Consequently, in order to measure the semantic correctness of candidate services, we select from the aforementioned characteristics the ones that define service's structure and behavior. These characteristics are: self-containment, composability and coarse-grained (functionality).

Characteristic	**Type**	
	Structural and behavioral	**SOA platform**
coarse-grained = functionality	✓	
discoverable		✓
self-contained = loosely-coupled	✓	
dynamic-binding		✓
composable	✓	
message-based		✓
synchronous / asynchronous		✓
well-defined		✓
stateful / stateless		✓

Table 3.1: Characteristics of services

3.3.4 Refinement of Service Characteristics

The former selected characteristics are refined to measurable quality properties.

- A service can be completely self-contained if it does not require any interface, i.e. it can be deployed as a single unit without depending on other services [Nakamura 2009]. Thus, the property number of interfaces the service requires gives us a good indication on the self-containment of the service. The higher the number of required interfaces is, the less the service is self-contained.

- A service is subject to composition with other services. This composition is realized without internal modifications but through service interface. A decomposition of the legacy system will be effective with the principle of composing those services with high cohesion and loose coupling, i.e. two services are composed with each other if their interfaces are cohesive. Thus, the average of services' cohesion within an interface gives us a good indication on the composability of the service.

- A service is more likely to be coarse-grained and hence represent complex, rich and high-level business functionality. However, it may sometimes be fine-grained [Channabasavaiah 2004] and hence represent low-level primitive functionality. Choosing the right level of granularity is the key for a successful service reuse. The bigger the service grains are, the less the service becomes reusable. It is relatively difficult to determine from source code the exact number of functionalities that the service provides. However, several factors can help measuring the functionality of a service:

  1. A service that provides several interfaces may provide numerous functionalities, thus the higher the number of interfaces is, the more the service provides functionalities.
  2. An interface whose services are highly cohesive probably provide single functionality.
  3. A group of interfaces with high cohesion are most favorable to provide single or limited number of functionalities.
  4. When the extracted code of candidate service is highly coupled, this means that the service probably provides very few or single functionality.
  5. When the extracted code of candidate service is highly cohesive, this means that the service probably provides very few or single functionality.

  Thus, we suggest binding the functionality characteristic to properties as indicated in table 3.2.

Functionality Characteristic	Property
A service that provides several interfaces may provide numerous functionalities, thus the higher the number of interfaces is, the more the service provides functionality.	Number of provided interfaces
An interface whose services are highly cohesive probably provide single functionality.	Average of service's interface cohesion within the interface
A group of interfaces with high cohesion are most favorable to provide single or limited number of functionality.	Cohesion between interfaces
When the extracted code of candidate service is highly coupled, this means that the service probably provides very few or single functionality.	Coupling inside a service
When the extracted code of candidate service is highly cohesive, this means that the service probably provides very few or single functionality.	Cohesion inside a service

Table 3.2: Binding functionality characteristic to properties

3.3.5 Quality Metrics

In our approach, according to the characteristics and properties of services we have chosen above, we build our quality metrics to evaluate the quality of candidate services. This quality will be the factor in distinguishing the extracted candidate services. The property functionality requires coupling and cohesion measurements, while composability only requires a cohesion measurement (see figure 3.3). As to [Patidar 2013], cohesion of a service measures how strong the elements within this service are related to each other. A service is considered as highly cohesive, if it performs a set of closely related functions and cannot be split into finer elements. The metric LCC Loose Class Cohesion proposed by [Bieman 1995] measures the overall connectedness of the class. It is calculated by:

$$LCC = \frac{number\,of\,direct\,and\,indirect\,connections}{maximum\,number\,of\,possible\,connections} \tag{3.1}$$

Coupling means the degree of direct and indirect dependence of a class on other classes in the system. Here, two measures are counted: method calls and parameter use, i.e. two classes are considered coupled to each other if the methods of one class use the methods or attributes of the other class. In our approach, $Coupl(E)$ measures the internal coupling of the candidate service E and is calculated by the ratio between number of classes inside the service that are internally called to the

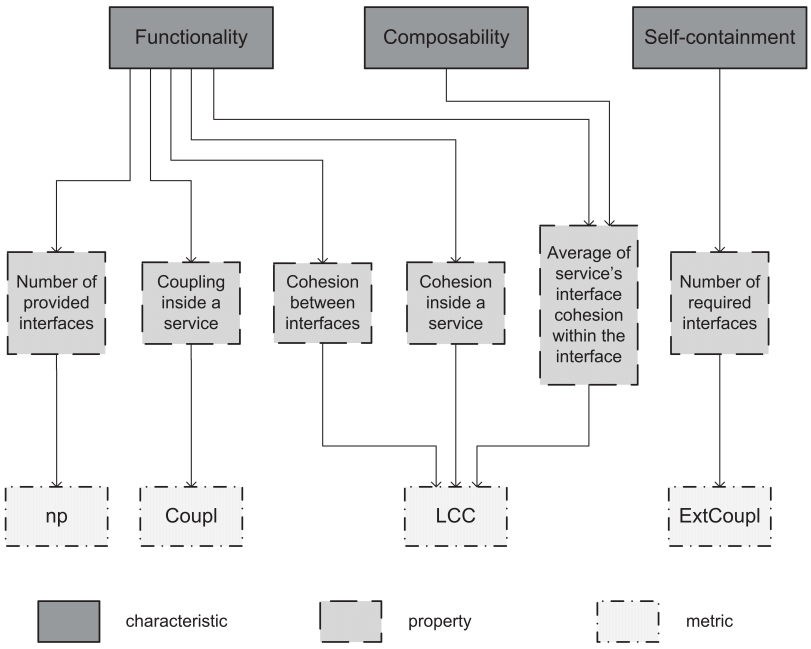


Figure 3.3: Refinement model of service characteristics

total number of classes within the candidate service E. $ExtCoupl(E)$ measures the coupling of the candidate service E with other services. It is calculated as $ExtCoupl(E) = 1 - Coupl(E)$.

3.3.6 Fitness Function Definition

We define a fitness function $FF(E)$ for an identified candidate service E as a linear combination between the 3 characteristics of services previously defined, $F(E)$ for functionality, $C(E)$ for composability and $S(E)$ for self-containment as follows:

$$FF(E) = \frac{\alpha F(E) + \beta C(E) + \gamma S(E)}{n} \tag{3.2}$$

; where α, β, γ are coefficient weights for each characteristic that are determined by software architect and $n = \sum(\alpha, \beta, \gamma)$.

The characteristics functionality $F(E)$, composability $C(E)$ and self-containment $S(E)$ are measured according to their definition as follows:

$$F(E) = \frac{1}{5}\Big(np(E) + \frac{1}{I}\sum_{i \in I} LCC(i) + LCC(I) + Coupl(E) + LCC(E)\Big) \tag{3.3}$$

; where $np(E)$ refers to number of provided interfaces, $LCC(i)$ refers to the average of service's interface cohesion within the interface, $LCC(I)$ refers to the cohesion between interfaces, $Coupl(E)$ refers to the coupling inside a service, and $LCC(E)$ refers to the cohesion inside a service.

$$C(E) = \frac{1}{I}\sum_{i \in I} LCC(i) \tag{3.4}$$

; where i refers to interface

$$S(E) = ExtCoupl(E) \tag{3.5}$$

3.3.7 Service Clustering

In order to recover services from OO legacy code, we group classes based on their dependencies. For that purpose, we propose a hierarchical agglomerative clustering algorithm which generate clusters by gradually grouping system's classes using a similarity measure.

Algorithm 1: Agglomerative Hierarchical Clustering

Input: OO source code classes
Output: A hierarchy of clusters (dendrogram)

```
1 let each class be a cluster;
2 compute fitness function of pair classes;
3 repeat
4 |   merge two "closest" clusters based fitness function value;
5 |   update list of clusters;
6 until only one cluster remains;
7 return dendrogram
```

The hierarchical agglomerative clustering algorithm groups together the classes with the maximized value of the fitness function. At the outset, every class is considered as a single cluster. Next, we measure the fitness function between all pairs of clusters. The algorithm merges the pair of clusters with the highest fitness function value into a new cluster. Then, we measure the fitness function between the new formed cluster and all other clusters and successively merge the pair with the highest fitness function value. These steps are repeated as long as the number of clusters is bigger than one, as illustrated in Algorithm 1. As a result, the legacy system is expressed in hierarchical view presented in a dendrogram, as illustrated in figure 3.4.

To obtain a partition of disjoint clusters, the resulting hierarchy needs to be cut at some point. To determine the best cutting point we employ the standard Depth First Search (DFS) algorithm. Initially on the root node, we compare the similarity of the current node to the similarity of it child nodes. If the current node's similarity value exceeds the average of similarity value of its children, then the current node is a cutting point, otherwise, the algorithm continues recursively through its children.

By applying the aforementioned clustering algorithm, we evaluate the legacy system and represent its classes in coarse-grained and loose-coupled disjoint set of services. An example of partitioning legacy system's classes to services is illustrated in figure 3.4.

3.4 Service Packaging and Deployment

A software service is an independent functional business entity that is hidden behind its well defined interface which ensures the easy discovery and the use of service by other service invokers (e.g. automatic agents, end user applications). The encapsulation of implementation logic and data of service is called service packaging. Among

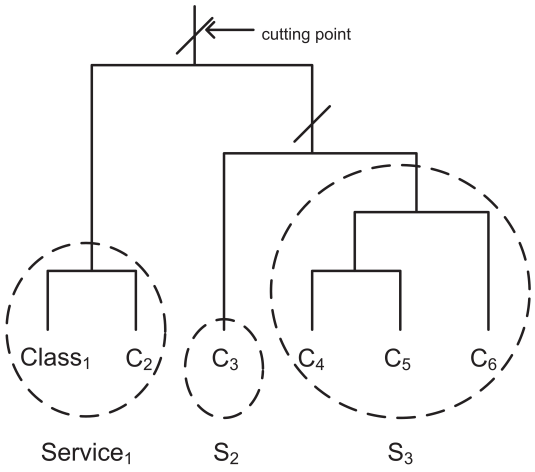


Figure 3.4: Dendrogram with set of services

several service deployment techniques, we choose the wrapping strategy because it helps representing system's functionalities to be accessible externally without changing its inner architecture or implementation.

Here, we suppose that services of the target system are provided as Web services. The W3C [Booth 2004] defines the Web service as *"A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards"*. We consider that extracted services are Web services for the following reasons:

- Web services share the characteristics of more general services [Brown 2002], however they are written in standard Internet technologies and have a standard accessing interface
- Enterprises heavily rely on Web services, since they can be physically located beyond company's internal network anywhere in the World Wide Web
- The implementation of Web service is totally independent of the user's operating environment

From an implementation point of view, a Web service is typically composed of two parts:

1. Business logic (i.e. implementation code), which actually carries out the real work to provide service functionality

2. Interface implemented in XML, which describes what the service provides as functionality in addition to how to communicate with that service (i.e. how a service can be called and what results it returns).

In the following sections, we demonstrate how each of those parts are built or adapted within the migration process from Object-Oriented (OO) legacy system towards SOA. Our service packaging and deployment phase consists of the following steps:

1. First, we prepare interface-based clusters by transforming dependencies between classes of different clusters into dependencies between clusters
2. We assign appropriate names to each cluster (from now an called service) using an annotation algorithm
3. Then, we generate an interface in SOA paradigm for each existing service which will expose services' provided functionalities
4. We finally register these services in a central registry so that they can be found and accesses by service invokers

3.4.1 Service Deployment

In OO structure, we rather differentiate 2 types of classes:

1. **internal class** C_i, which has only internal connections to other classes of the same cluster.
2. **interface class** F_j, which has at least one external connection to classes of other clusters.

For example, in figure 3.5b, classes C_1 and C_2 are examples of internal classes, whereas F_1, F_2, F_3 and F_4 are examples of interface classes.

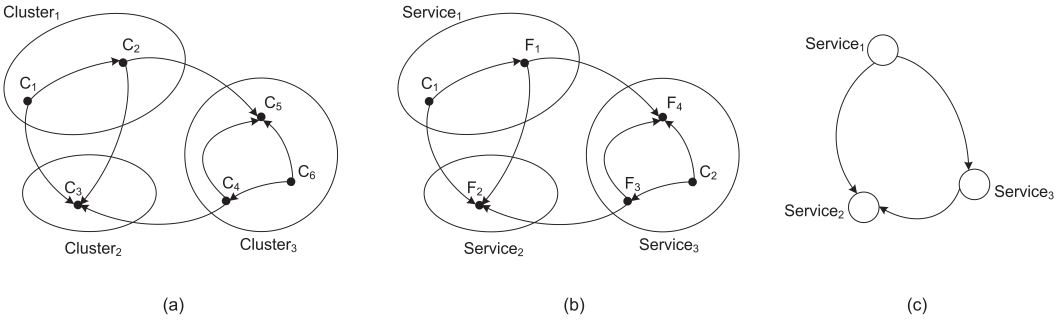


Figure 3.5: Transforming OO dependencies to interface-based dependencies

In figure 3.5a, a collection of certain classes are grouped in one cluster $Cluster_k$ based on the quality metric defined in section 3.3.5 and the clustering algorithm in section 3.3.7 on page 44 and 46, respectively. We remind that at the end of the clustering phase, we have identified which set of classes will form a service. Every cluster, in our approach, is transformed into a service $Service_k$ (see figure 3.5b and 3.5c) in SOA paradigm.

However, the service's concrete implementation is still resided within each object-oriented class separately and the dependencies are still between classes not services. In order to raise the dependencies between classes into a higher level of abstraction (service level) and to reduce the coupling between classes (see figure 3.5b and 3.5c), we need to make dependencies between classes of different clusters to become interface-based. Therefore, we need to set up a new intermediate class that serves as a communication port between clusters. This new established class called *"delegator class"* has an interface role; it exposes the provided functionalities of its composing interface classes F_i.

To build a delegator class, we inspire from delegation design pattern as demonstrated in figure 3.6.

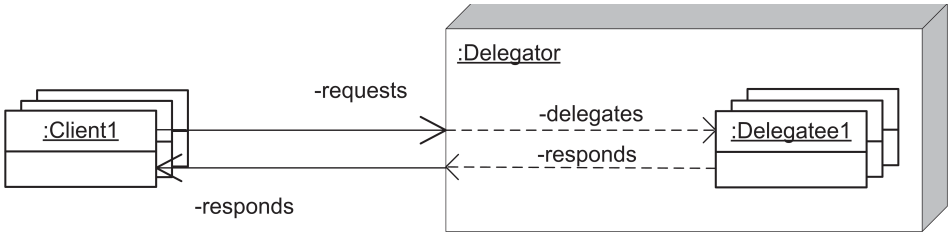


Figure 3.6: Delegation design pattern

This delegator class defines the same methods of all interface classes with their same signature. However these methods are only delegation methods; the delegator class captures requests from external, instantiates an object of a proper class, and delegates the execution to that class.

An example is demonstrated in figure 3.7b, where $DelegationClass3$ defines a method $m5$, instantiates an object of class $C5$ and delegates its execution to $C5$. While originally the class $C2$ was invoking the class $C5$ in figure 3.7a, this invocation becomes an invocation between their delegator classes (i.e. $DelegationClass1$ invokes $DelegationClass3$), as demonstrated in figure 3.7b. In this figure, delegation is designated with dashed arrows. While invocations from and to delegator classes

are added during the service deployment phase, the part of architecture which is designated in gray in figure 3.7b in addition to its implementation remain unchanged. By transforming dependencies between classes of different clusters into dependencies between delegator classes, we make interface-based clusters. This process allows hiding classes' internal dependencies and exposing only provided methods of interface classes.

Transformation rules presented in [Alshara 2015] can be applied in this service deployment phase in order to automatically transform dependencies between classes to dependencies between clusters (i.e. services).

3.4.2 Service Annotation

Identified services should be distinctly annotated, in a manner that they reflect their composing classes. For that, we observe that a good object-oriented design has some basic organizational and metaphoric conventions that software developers follow in their designs such as component naming conventions. Class naming usually follows camelCase naming convention.

In order to automatically label each extracted service (i.e. delegator classes) with the most relevant name, we exploit the linguistic information found in the names of classes that compose a service and use a word-frequency technique. Our labeling algorithm has the following steps: (1) extracting vocabulary terms, (2) weighting terms and (3) composing terms to form new service name.

1. First, we extract vocabulary terms from each composing classes' composite names by breaking up these composite names using a standard camelCase splitting heuristic. We then exclude stop-words and programming language special words.

2. A weight is attributed to each extracted term within a service according to its frequency and its position within the entire class name. For example, terms that are located in the first position of the class name are more likely to express the main purpose and are consequently more important and indicative than the terms that are located on the second position. Thus, a higher weight is attributed to the term that is located in the first position. Even more, as we have differentiated in section 3.4.1 between internal classes C_i and interface classes F_j, different weights are attributed to terms of each type of class; internal classes either deal with internal operations or they are utility classes and thus are not much concerned about the main functionality that the service provides. Whereas, interface classes and especially provided interface classes (for example, C_3 in figure 3.5a) are more likely to play a main role in service's provided functionality. Terms of such provided interface classes are weighted

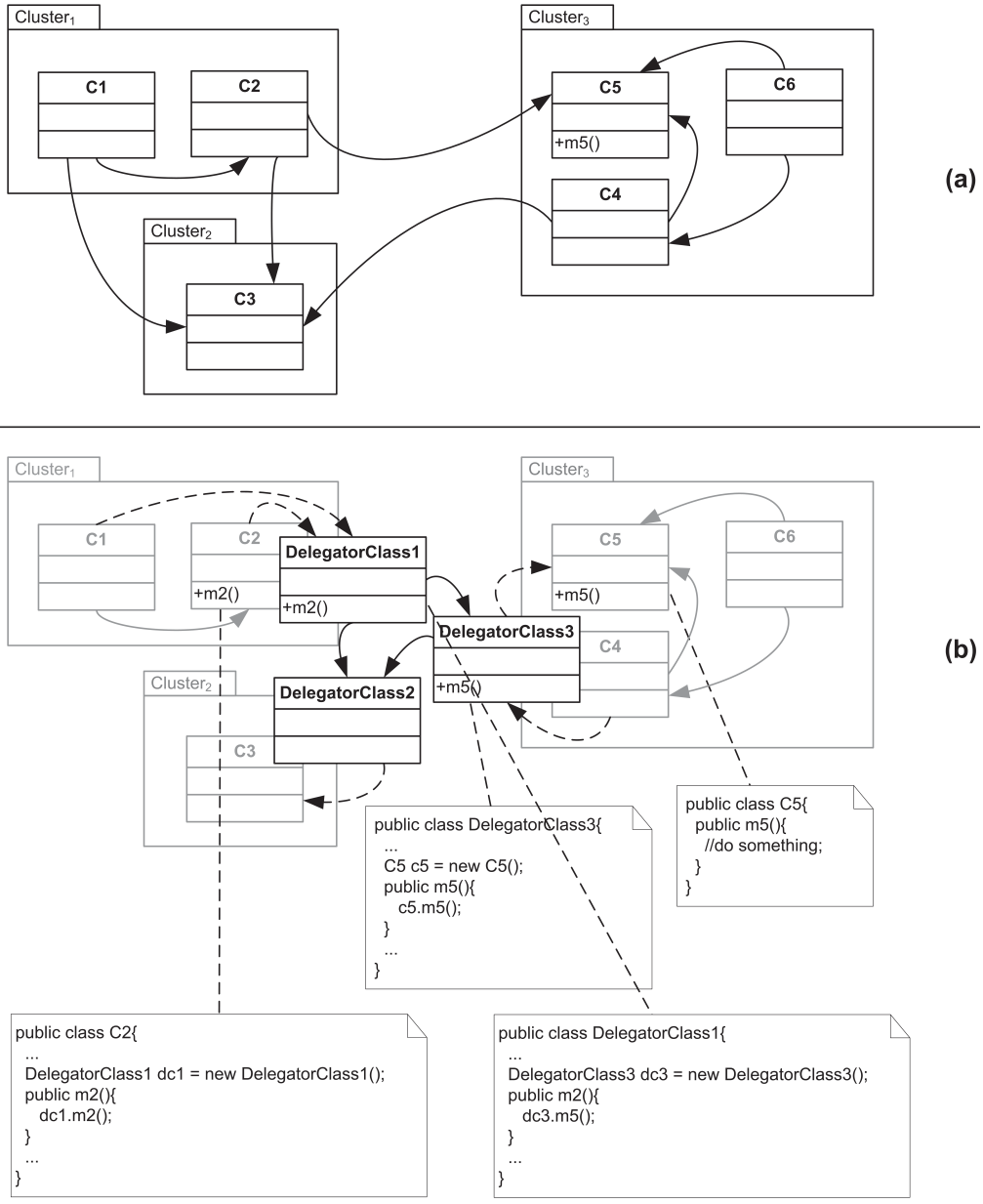


Figure 3.7: Service interface development

higher than terms of internal classes. For each term t, the weight $w(t)$ is calculated by the following formula:

$$w(t) = \frac{1}{\sum_{1 \in i}^{4} n_i}(\alpha \times n_1 + \beta \times n_2 + \gamma \times n_3 + \delta \times n_4) \tag{3.6}$$

; where

- $\alpha, \beta, \gamma, \delta$ are coefficient weights attributed to each factor. $\alpha, \beta, \gamma, \delta \in \Re[0, 1] \wedge (\alpha > \beta > \gamma > \delta)$.
- n_1 is the number of occurrences of term t on the first position of the provided interface classes' name.
- n_2 is the number of occurrences of term t on the first position of the internal classes' or required interface classes' names.
- n_3 is the number of occurrences of term t on a position other than the first of the interface classes' names (provided or required interface).
- n_4 is the number of occurrences of term t on a position other than the first of the internal classes' names.

3. For each cluster, we sort the list of terms in a descending order according to their weights. Finally, to attribute a new cluster name, we compose the terms with the highest weights respecting once again the camelCase naming convention. The number of terms that the a cluster name can be composed of is determined by the architect. We recommend that a name is composed of not more than five terms. In case more than one term has the same weight, both terms are added to cluster's name.

3.4.3 Service Interface Generation

In SOA paradigm, the description of service's functionality and properties is defined separately from its implementation. It is the service interface which encloses and exposes the functionality of its service. In other words, using interface allows service's internal implementation details to be hidden from external clients. Every service S_k is represented by an interface I_k that describes its functionalities. This interface also defines the types of messages' exchanges in addition to a description about the service. Service description should be enough sufficient so that it will enable external services or automatic agents to discover the service and use it.

In Web services, Web Services Description Language (WSDL) interface file represents service's interface. Its main role is to abstractly describe the operations

that the service provides, in addition to the parameters that the service accepts and returns, regardless of the hardware platform or the implementation language of the service. More precisely, a WSDL file contains information on:

- All publicly available functions that are externally visible
- All message requests and message responses
- Binding information about the transport protocol to be used
- Address information for locating the specified service

Our goal in this phase is to provide an interface to each service that was identified from the legacy OO code. For Web service, this interface is summarized in a WSDL document. Service interfaces are built based on delegator classes which we have set up in OO paradigm. Similar to delegator classes that ensure their composing classes' external connections by describing all their public methods, in service interface I_k (WSDL file), we define all public methods which the delegator class provides.

Once the delegator class in OO paradigm is correctly set up, its interface in SOA paradigm can be automatically generated. There are tools (such as in Eclipse environment) that can automatically generate Web service's interface specified in Web Service Description Language (WSDL). It is worth noting that this automatic procedure is applied only in case we want to expose the functionalities implemented within one single class. Since we have regrouped all public methods of interface classes F_j in a separate class called $DelegatorClass_k$, we can automatically generate a separate WSDL file in compliant with each $DelegatorClass_k$. This automatic transformation is demonstrated in figure 3.8; in OO paradigm an implementation of $DelegatorClass_3$ containing one delegate method m_5 (which is concretely implemented in class C_5) is transformed to SOA paradigm.

3.4.4 Service Registration

After the creation of services and interfaces, extracted services in SOA are registered in a warehouse such as in a Universal Description Discovery and Integration (UDDI) central repository, an optional but recommended element of a service-oriented architecture. The registration process will further help client applications (service consumers) to easily search for a service (provider service) and locate it at runtime by simply querying the UDDI repository.

Using UDDI has several advantages, among them:

- Dynamic reconfiguration: UDDI allows the dynamic reconfiguration of SOA application. This is useful when there's a need to replace versions of components with no system interruption.

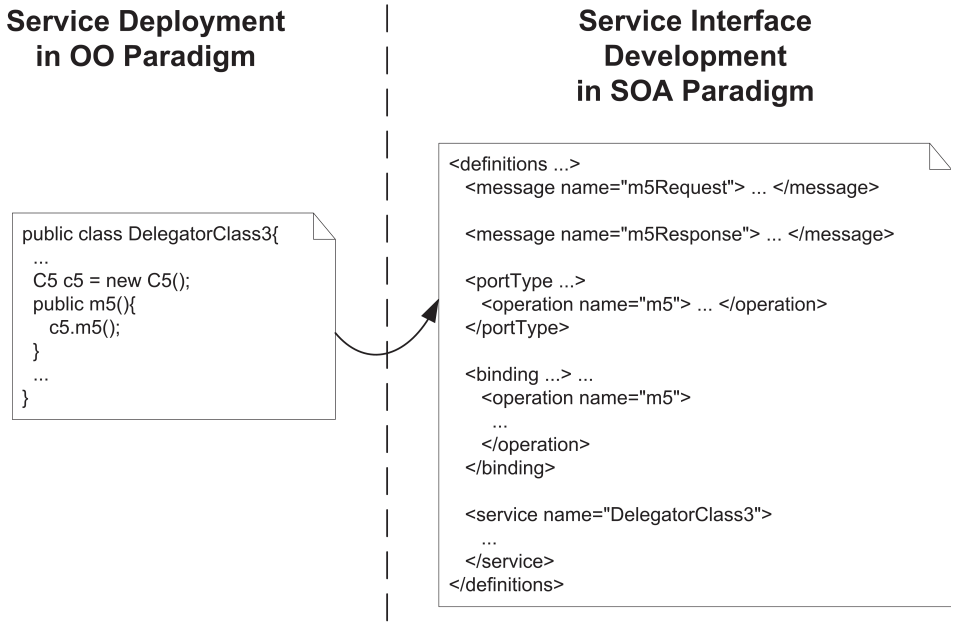


Figure 3.8: WSDL generation

- Protocol independent: UDDI is not restricted to Simple Object Access Protocol (SOAP)[1] based web services. Rather, UDDI can be used to describe any service type whether it is SOAP, CORBA or Java RMI services.
- Private UDDI registries: it is possible for a company to set up its private UDDI registries that registers only internal web services. This registry will not be synchronized with the public UDDI registries which are available on the Internet.

In section 3.4.1 and 3.4.2, we have deployed the services in OO paradigm, in section 3.4.3, we have generated their interfaces in SOA paradigm and finally in this section we have registered deployed services in the service registry. In order the Web services to communicate between each other, several elements are needed. Figure 3.9 displays those elements.

1. First, a client, who wants to invoke a service, searches for the appropriate service within the available services that are registered in the registry (i.e. UDDI registry).
2. Each registry records in addition to service's meta-data, a pointer to the service interface description (i.e. WSDL file) of each given service.

[1]SOAP is a standard XML-based communication protocol for consuming Web services through message exchanges over the Internet

3. Service interface informs the client how to invoke and desired Web service by providing information about its binding, operation and input/output messages.

4. The client can now connect to the given service over the specified binding (usually SOAP) and invoke its remote methods.

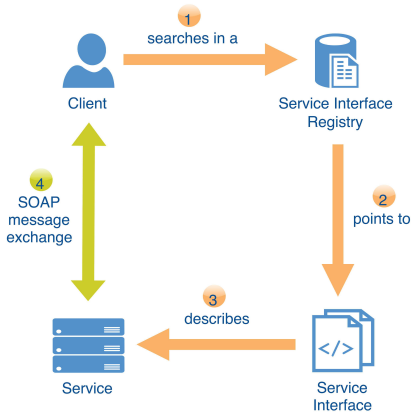


Figure 3.9: Web service invocation procedure

3.5 Conclusion

The main contributions of the work presented in this chapter is the identification of services from legacy OO source code and their packaging. First, we have identified services in OO source code based on service quality characteristics. For this purpose, we first set a mapping model between object and service concepts. Then, we have studied service and in particular Web service characteristics and classified those characteristics to two categories; those describing structure or behavior and those describing the environment (i.e. SOA platform). We have noticed that only structural and behavioral characteristics would help to extract services in an existing legacy code. Therefore, we used those characteristics as measurement metrics. Unlike most ad-hoc service identification approaches, we introduced a fitness function that measures the quality of identified services. The measurement metrics of fitness function are based on a refinement model of service's semantic characteristics.

In a second phase, we have grouped classes with similar functionalities in one cluster using hierarchical agglomerative algorithm and introduced a delegator class which exposes the provided methods of one cluster to public, while hiding service's internal implementation and reducing the inter cluster coupling. This delegator class

helped us to automatically generate from its methods' signatures an interface in SOA paradigm. As a result, we obtained an interface (WSDL document) for each cluster, describing its functionality.

It is worth noting that this approach is especially applicable to modernize legacy systems for which no software assets but the source code is available.

In chapter 5, we will demonstrate experimentation of proposed service identification and packaging approaches.

Chapter 4

Variable-Architecture Centric Reconfiguration of Service-Oriented Systems

Contents

4.1 Introduction

4.1.1 Context and Motivation

Software systems are growing in terms of size and complexity. Accordingly, the need for software systems to become flexible and to support dynamic reconfiguration is increasing. In dynamic reconfiguration, parts of the system can evolve to cope with changing environment during system's execution. Throughout the life cycle of software development, architecture provides the required level of abstraction to deal

with those dynamic reconfiguration and adaptability issues; this eases managing reconfiguration as well as facilitates traceability between dynamic descriptions at architecture level and their counterparts at other abstraction levels.

In the context of describing dynamically reconfigurable software architecture, different Architecture Description Languages (ADLs) have been proposed to specify reconfigurable artifacts of a dynamic architecture. However, the reconfiguration decisions within these ADLs are expressed in an ad-hoc manner. On the other side, variability modeling allows an explicit specification of configurable artifacts. Combining variability modeling to context and reconfiguration descriptions allows software systems to systematically adapt their behavior at runtime in respond to surrounding context changes without an explicit external or manual intervention. This increases system's autonomy, usability and effectiveness.

Our contribution in this chapter is to describe, through a specific ADL called Dynamic Service-Oriented Product Lines (DSOPL), the dynamic architecture of a self-reconfigurable software system in which reconfiguration scenario is based on variability descriptions. This allows the architecture to adapt its behavior to context information which itself is subject to change. We propose an XML-based ADL that allows describing four main types of information:

- architecture's structural description in terms of services, operations, interfaces and service composition
- an architectural variability description (i.e. variability points and alternatives), on which system's reconfiguration is based
- context information, to which service reconfigurations adapt
- an architectural configuration description (i.e. reconfiguration rules based on context and variability information)

We choose to use XML as a description language to facilitate understandability and analysis of the described architecture. In addition, XML-based description facilitates tool-support design and interoperability.

4.1.2 Illustrative Example

We will use throughout this chapter an illustrative example to exemplify concepts related to our proposed approach. This example is about a simplified on-line sales scenario between four actors; customer, retailer, warehouse and shipment services, as modeled in figure 4.1. The customer accesses retailer's website, browses the catalog, selects some items and commands an order. The retailer fulfills customer's order request and inquires the warehouse to prepare all items of the order. Once the order is prepared, the shipping service handles the delivery of items to the customer.

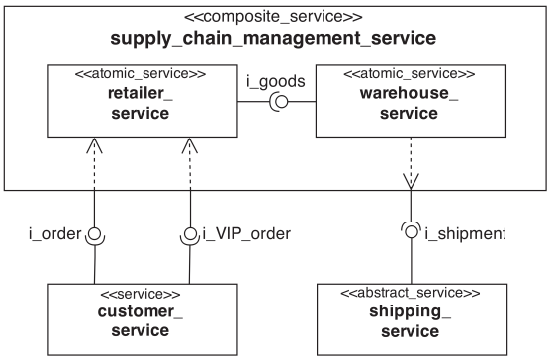


Figure 4.1: Illustrative example: On-line sales scenario architecture

4.1.3 Chapter Organization

The rest of this chapter is organized as follows: In section 4.2, we present the architectural language specifications; In subsection 4.2.1, we introduce our modular ADL. In subsection 4.2.2, we present the structural elements that our ADL specifies. In subsection 4.2.3, we investigate different types of variability and integrate its elements to our ADL. In subsection 4.2.4, we model context information and integrate it as a separate concern to our ADL. In subsection 4.2.5, we present the behavioral aspect of our ADL. Here, we demonstrate the different communication activities between services and differentiate between common and variable configurations. We end up, in section 4.3, with a concrete architecture generation and a transformation of this concrete architecture to an executable language. Finally, in section 4.4, we present a conclusion to the chapter.

4.2 Dynamic Architecture Description Language Based on Variability Specification

In basic Architecture Description Languages (ADLs), services can be selected and composed at design time. Such ADLs are considered static by default. The architecture of a static system is specified basically in terms of architectural composing elements (components, services, etc.), connections between those elements (links, connectors, etc.) and composition of those elements (one configuration and sub-architecture). Static architecture has only one configuration which is defined at design time.

In contrast, Service-oriented or context-aware architectures, have a dynamic nature and are composed of loosely coupled architectural elements that should be

reconfigurable at runtime, thus a special ADL should specify, in addition to structural elements' information, also the dynamic behavior of the system that can be reconfigured at runtime due to a context and environment changes. In dynamic environment, parts of the software can be instantiated or evolved at runtime. Therefore, we need to describe behavioral information of the running system at architecture level. Furthermore, behavioral information is described and configured based on variability information, thus the need to specify also variability information as first-class elements at architecture level.

4.2.1 DSOPL: A modular ADL for Describing Dynamicity Based on Variability

A fundamental element of developing dynamic reconfigurable system is its architecture description. We adopt the reference architecture term to refer to architecture in which variation points are modeled at architecture level and in which reconfigurations are based on those variation points. Reconfiguring a reference architecture and generating a concrete architecture from it is based on context changes. The context consists of any element that influences the behavior and/or the structure of the architecture. It can be related either to system's environment (e.g. escalator state in the case of crisis management software), evaluated quality of service (e.g. time to response to a query), hardware architecture changes (e.g. server failure), etc. Thus, context element needs to be described in a dynamic ADL. We include these context description as a first-class architectural element to allow context-aware configurations (i.e. autonomous run-time adaptation according to context changes).

Variability description is an excellent solution to specify dynamically reconfigurable architectures and hence manage artifacts and their interconnection variations. In fact, the advantage of integrating variability description at architecture level can be twofold:

- A consistent management of artifacts' variances at early stages of design facilitates fast and correct development of software system that incorporates variability.
- Easier control of any modification and its reflections on later development phases, since architecture plays a reference role for all development activities of software system life cycle.

In order to describe the dynamic reconfiguration of a service-based system that encompasses variability and at architecture level, we propose an ADL that is structured in four parts. A detailed description of each of those four parts is given in the following sections.

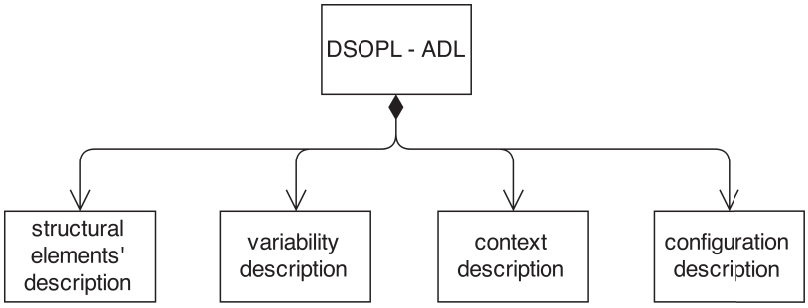


Figure 4.2: Modular DSOPL-ADL

1. **Structural element description:** defines all types of the abstract structural entities of the system (services, interfaces, operations).

2. **Variability description:** here, variation points are defined and also all alternative services of each variation point with the constraints related to each alternative.

3. **Context description:** variability and configuration descriptions are based on information about context. Thus, information about context elements is described in a specific section of the ADL.

4. **Configuration description:** here, the rules used to create concrete services and connections are specified to describe how to configure (generate) concrete architectures based on structural, variability and context elements.

Our approach implicitly separates the four aforementioned architectural concerns from each other as it is summarized in figure 4.2. This modular separation of architecture description in four sections, each of them specifying one type of architectural description, has above all the following advantages:

- ✓ It facilitates the modification and re-utilization of each of the four sections of ADL.

- ✓ It allows the description and analysis of the architecture by separating the four concerns (structure, variability, context and configuration).

- ✓ It allows controlling the traceability links of each type of information among several abstraction levels. For example, the variability described in feature model at requirement level is translated at architecture level through its variability section.

4.2.2 DSOPL Structure Description

Structural information can be specified independently from reconfiguration or variability information. In structural description, we specify the structure of all artifacts that make part of the architecture, whereas in configuration, we specify which artifacts are instantiated and their behavior, and in variability description, we describe which structural artifacts are alternatives of one variation point.

As structural description, we represent the main artifacts of service-based system: a *service* is an encapsulated and self-contained unit. It interacts with other services through *interfaces*. The system itself is a composite service and is hierarchically decomposed into finer-grained services. We call leaf services as *atomic services*. All other services in the hierarchical tree, that are composed of at least one leaf service, are considered *composite*. A composite service does not execute or implement any functionality by itself, but it delegates this task to one of its child services. Each composite service is described as sub-architecture.

Each service has a number of *provided interfaces* and may require a number of *required interfaces*. *Interfaces* define a collection of methods or operations that are supported by the service. Since services are developed independently from their future exploiting systems, they should have well-defined interfaces that describe their functionalities and operations. Interfaces are two types, either *provided interface* or *required interface*. *Provided interface* of a service is an interface that the service realizes, whereas *required interface* is an interface that the service needs in order to operate. Services communicate to each other through provides/ consumes relationship via their provided/ required interfaces. Hence an interface has a set of *operations*.

A service is described based on the following architectural attributes, as displayed in figure 4.3. Every service has:

1. a name specified by `service_name` attribute
2. a `textual_description` that explains in plain text the main functionalities of the service, its inputs and expected outputs. This information could be used by service clients that would like to invoke a the service
3. `is_atomic` has a Boolean value to indicate whether the service is atomic or composite

Listing 4.1 shows the structural section description of the architecture related to our illustrative on-line sales example. The example is composed of the following atomic services: `retailer_service`, `warehouse_service`, `customer_service`, `relay_point_shipping_service` and `home_delivery_shipping_service`.

`retailer_service` and `warehouse_service` are grouped in a composite service named `supply_chain_management_service`. Each service has some interfaces. For example, `retailer_service` has two interfaces; a provider interface named `i_order` and a required interface named `i_goods_request`. We also notice that two operations named `submit_order_request` and `get_catalog` are provided by `retailer_service` through `i_order` interface.

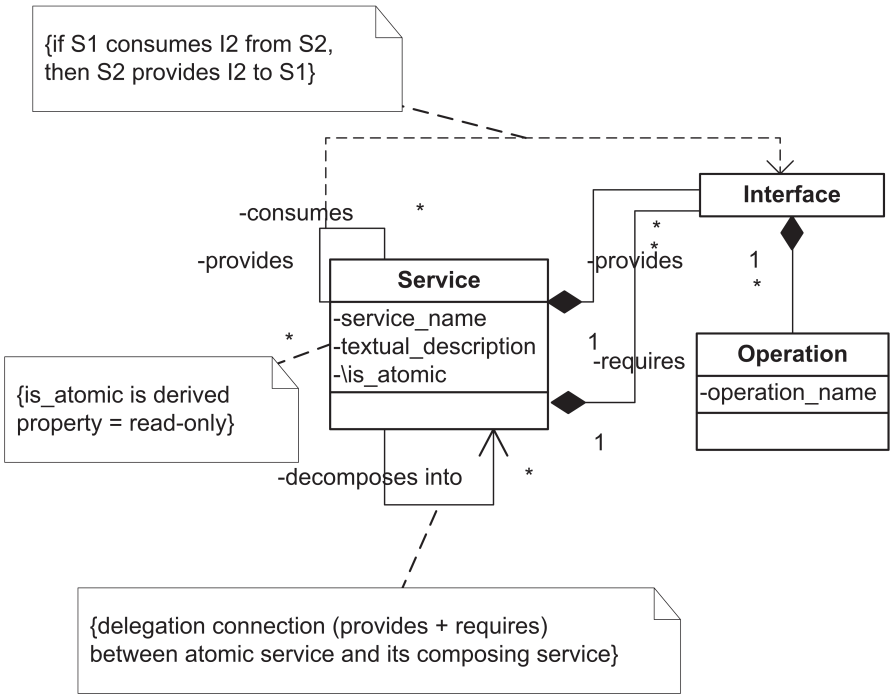


Figure 4.3: Structural description meta-model of DSOPL-ADL

Listing 4.1: Structural Description of Sales Scenario

```
<DSOPL-ADL>
  <structural_description>
    <service name="supply_chain_management_service" textual_description="this is a
        composite service that provides sales ordering functionalities to customers
        " is_atomic="N">
      <interfaces>...</interfaces>
      <sub-architecture>
        <service name="retailer_service" ... is_atomic="Y">
          <interfaces>
            <interface name="i_order" role="provides">
```

```
            <operations>
              <operation name="submit_order_request" ...> </operation>
              <operation name="get_catalog" ...> </operation>
            </operations>
          </interface>
          <interface name="i_goods_request" role="consumes"> ... </interface>
        </interfaces>
      </service>
      <service name="warehouse_service" ... is_atomic="Y">
        <interfaces> ... </interfaces>
      </service>
    </sub-architecture>
  </service>
  <service name="customer_service" ... is_atomic="Y">
  ...
  </service>
  <service name="relay_point_shipping_service" ... is_atomic="Y">
  ...
  </service>
  <service name="home_delivery_shipping_service" ... is_atomic="Y">
  ...
  </service>
 </structural_description>
 <variability_description> ... </variability_description>
 <context_description> ... </context_description>
 <configuration_description> ... </configuration_description>
</DSOPL-ADL>
```

4.2.3 DSOPL Variability Description

In our variability description of DSOPL-ADL, we inspire from the variability modeling at requirements level through feature models and reflect it in our ADL as variability management at architecture level.

4.2.3.1 Variability Description Specification

We specify in this section the different variation points that exist in the system at architectural level. The meta-model of variability description is given in figure 4.4. A *variation point* specifies the part of the architecture that can be variable. Each variation point has the following attributes:

1. `variation_name` indicating its unique name.
2. `variation_type` that specifies the type of this variation. Possible values of `variation_type` are either `service`, `connection` or `composition`.
3. `variation_time` specifies whether this variation may occur at `compile-time` (i.e. before runtime) or at `runtime`. Contrary to traditional SPL approaches

where variability is clearly and entirely specified at design time [Galster 2010], `variation_time` attribute is important in SOA systems, where selection of an alternative during runtime is totally possible. However, some variation points could be specified at compile-time. This reduces the overhead of loading the entire configuration at runtime.

Each variation point has several *alternatives*, which are possible elements to fill the selected variation point. Each alternative has a unique name `alternative_name`, `reference_element` which refers to a structural element and an order of `priority`. This attribute helps the system automatically determine which architectural element is chosen in case there is more than one valid configuration at a given time. The alternative with the highest priority `priority="1"` is the preferred one in a variation point.

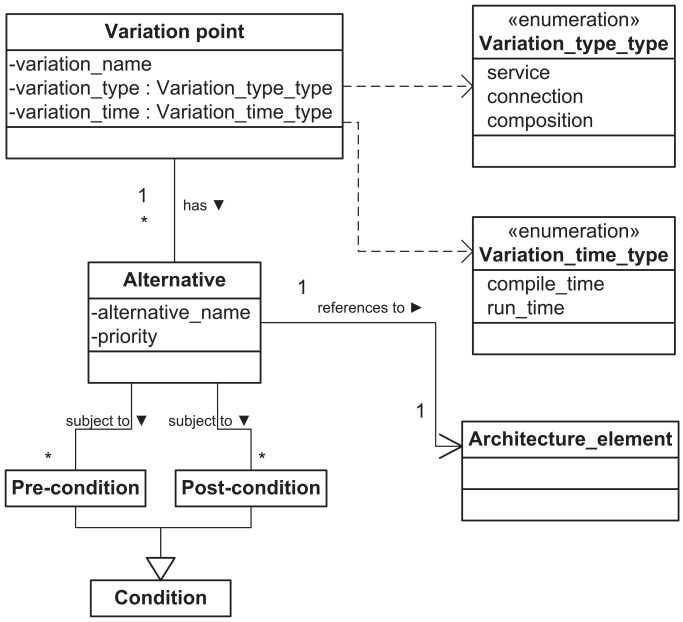


Figure 4.4: Variability description meta-model of DSOPL-ADL

4.2.3.2 Variable Artifacts

Variation and their possible choices may occur on different structural artifacts. Therefore, we distinguish three types of variabilities:

1. Service Variability Description:

   It represents binding an alternative service that satisfies pre-conditioned constraints on runtime. Back to our sales example, there are two alternatives of

shipment; either a relay point shipment or home delivery shipment, as shown in figure 4.5. The decision of which alternative to choose is taken automatically at runtime depending on customer's selection in addition to other environmental conditions such as the existence of a relay point service in customer's city, as depicted in listing 4.2.

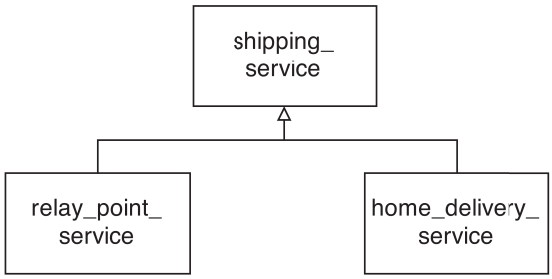


Figure 4.5: Example of service variability in sales scenario

Listing 4.2: Service variability description example

```
<variability_description>
  <variation_point name="shipping_variation_point" variation_type="service"
      variation_time="runtime">
    <alternatives>
      <alternative name="home_delivery_alternative" reference_element="
          home_delivery_shipping_service" priority="1">
        <constraints> ... </constraints>
      </alternative>
      <alternative name="relay_point_delivery_alternative" reference_element="
          relay_point_shipping_service" priority="2">
        <constraints> ... </constraints>
      </alternative>
    </alternatives>
  </variation_point>
</variability_description>
```

2. Connection Variability Description:

   It may exist several alternative connections between services. An appropriate connection is selected according to constraints' satisfaction. For example, the customer service in figure 4.6 can access the retailer service and thus command an order via two different connections; either a connection for a regular customer or a connection for a VIP customer which normally has some extra privileges. The *variation_point* `customer_variation_point` in listing 4.3 is an example of variability of connection. It has two alternatives,

either `regular_customer_alternative` which connects customer and retailer services via `i_order` interface or `VIP_customer_alternative` which connects customer and retailer services via `i_VIP_order` interface.

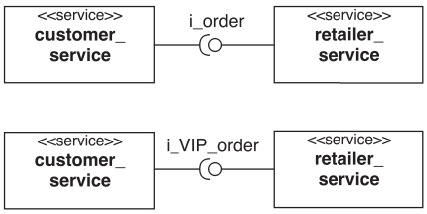


Figure 4.6: Example of connection variability in sales scenario

Listing 4.3: Connection variability description example

```
<variability_description>
  <variation_point name="customer_variation_point" variation_type="connection"
       variation_time="runtime">
    <alternatives>
      <alternative name="regular_customer_alternative" reference_element="
          i_customer_order" priority="1"> ... </alternative>
      <alternative name="VIP_customer_alternative" reference_element="
          i_VIP_customer_order" priority="2"> ... </alternative>
    </alternatives>
  </variation_point> ...
</variability_description>
```

3. Composition Variability Description:

   This type of variability concerns replacing not only a service or a connection, but replacing a set of interconnected services by another set of interconnected services within a composite architecture. Figure 4.7 illustrates another alternative composition of `supply_chain_management_service` than the one in figure 4.1. Here, in addition to the roles of retailer and warehouse services, the manufacturer service realizes requested items and returns them to the warehouse service. Listing 4.4 displays this composition variability description example. `supply_chain_composition_variation_point` has two alternatives, either a supply chain composition with manufacturing option or another composition excluding manufacturing.

Listing 4.4: Composition variability description example

```
<variability_description>
  <variation_point name="supply_chain_composition_variation_point"
       variation_type="composition" variation_time="compile_time">
```

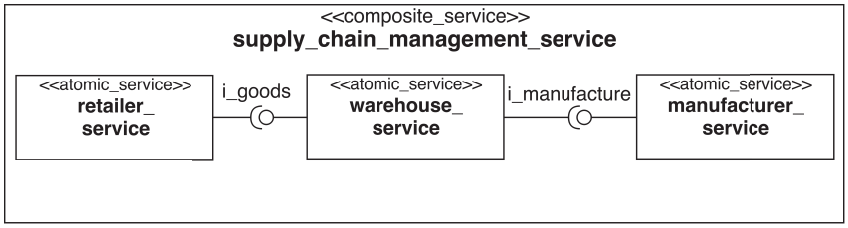


Figure 4.7: Example of composition variability in sales scenario

```
    <alternatives>
      <alternative name="supply_chain_without_manufacturing_alternative"
          reference_element="supply_chain_management_service" priority="1">
        <constraints> ... </constraints>
      </alternative>
      <alternative name="supply_chain_with_manufacturing_alternative"
          reference_element="supply_chain_management_service2" priority="2">
        <constraints> ... </constraints>
      </alternative>
    </alternatives>
  </variation_point>
</variability_description>
```

4.2.3.3 Constraints Related to Alternative's Instantiation

Each alternative has a set of constraints, in forms of *pre-conditions* and *post-conditions*, to operate properly.

- **Pre-condition** specifies a group of conditions that should be satisfied before selecting a given alternative (i.e. alternative can be selected, only if all constraints of pre-conditions are satisfied).
- **Post-condition** represents desirable outcomes when given alternative is selected successfully.

Pre- and Post-conditions are the equivalent of crosscutting *"requires"*, *"excludes"* and *"and"* constraints in Feature Model FM in SPL paradigm. For example, the pre-condition that states that in order to choose the alternative `"relay_point_delivery _alternative"`, the service `"relaying_point_service_in_city"` should be available (see listing 4.5), this statement is equivalent in FM to a *"requires"* constraint from `"relay_point _delivery"` feature to `"relaying_point_in_city"` feature. On the contrary, `condition="unavailable"` is equivalent to *"excludes"* constraint in FM.

Listing 4.5: Variability description of sales scenario

```
<variability_description>
  <variation_point name="shipping_variation_point" variation_type="service"
      variation_time="runtime">
    <alternatives>
      <alternative name="home_delivery_alternative" reference_element="
          home_delivery_shipping_service" priority="1">
        <constraints> ... </constraints>
      </alternative>
      <alternative name="relay_point_delivery_alternative" reference_element="
          relay_point_shipping_service" priority="2">
        <constraints>
          <pre-conditions>
            <pre-condition element_type="service" element="
                relaying_point_service_in_city" condition="available"/>
          </pre-conditions>
          <post-condidtions>
            <post-condition element_type="method" element="re-
                calculculate_total_amount" condition="execute"/>
          </post-condidtions>
        </constraints>
      </alternative>
    </alternatives>
  </variation_point>
</variability_description>
```

4.2.4 DSOPL Context Description

A context element could capture raw data from a single information source such as a GPS locator that locates customer's current location to search for a nearby relay point for the shipping service in our sales example. In this case, context element is considered as a *primitive_ context*. In some other cases, a single information source could not be sufficient to take decisions; in that case, different atomic information sources' values are collected, combined and analyzed in order to give sufficient and more accurate information about the context value. We call this context as *composite_ context*. We can consider the weather forecast example, where the weather is considered hot when both temperature and humidity sensors exceed a certain threshold.

A simplified meta-model of context is illustrated in figure 4.8. A context element can be described with the following attributes:

- a unique `name`
- a `context_type` to indicate to which family of contexts it belongs (e.g. contexts related to environment, user preferences, etc.)

- `values_type` that indicates the type of its values, either primitive types such as integer, double, etc. or user-defined types
- an `actual_value`

In listing 4.6, we show two primitive context descriptions from our sales scenario.

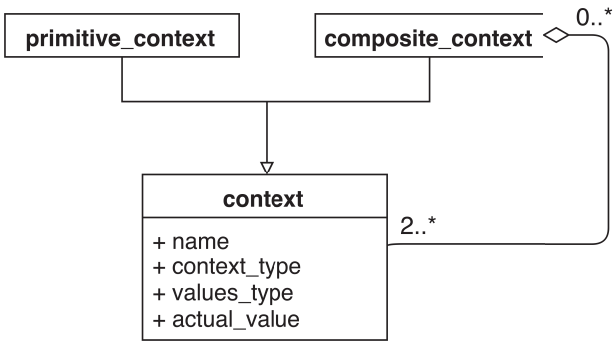


Figure 4.8: Context description meta-model of DSOPL-ADL

Listing 4.6: Some context descriptions from sales scenario

```
<context_description>
  <context_type name="environment">
    <context is_aggregate="N">
      <name> location </name>
        <values_type> double </values_type>
        <actual_value> Montpellier </actual_value>
    </context>
    <context is_aggregate="N">
      <name> shipping </name>
      <values_type> enumeration </values_type>
      <permitted_values>
        <possible_value> home </possible_value>
        <possible_value> relay_point </possible_value>
      </permitted_values>
      <actual_value> relay_point </actual_value>
    </context> ...
  </context_type> ...
</context_description>
```

4.2.5 DSOPL Reconfiguration Description

The configuration section of DSOPL-ADL allows describing all the configuration rules to generate a valid architecture. A valid architecture is a concrete architecture whose services and connections comply with configuration rules.

4.2.5.1 Behavioral Activities

In order to treat the dynamic reconfiguration at architecture level, the ADL should describe the order in which involved services are composed. To capture the behavioral aspect of a dynamic architecture we represent the flow from one operation to another. However, we focus only on the interaction between services (i.e. message flow from a service to another one). Interactions between services can be either single-directional invocations (receive and respond) or bi-directional invocations (invoke).

The inter-service communication (message flow) is realized through message passing. A message carries data between two services and has a textual format. Message passing is supported by three communication activities: *receive*, *respond* and *invoke*. In order for two services to communicate with each other, the requester service sends a message to provider service which receives it with the *receive* activity. Once the provider service finishes the required operation's execution, it returns a message to the requester with the *respond* activity.

Communication Message types

1. **receive:** In this type of communication, a consumer service (i.e. client) sends a message to the service provider without expecting any instant response. The uni-directional *receive* communication defines only an *input message* and requires no output message. The `<receive>` element specifies the consumer service using `consumer_instance` attribute and its interface using `consumer_interface` attribute. `provider_instance` reflects the name of the provider service and `provider_interface` reflects its interface. The consumer service triggers the execution of a specific *operation* at the provider's side in the `operation` attribute. Finally, the arguments passed to the operation are carried on `input_message` variable.

2. **respond** communication is used to reply to a message that was previously received through a `<receive>` activity. In that case, values of `consumer_instance`, `consumer_interface`, `provider_instance` and `provider_interface` attributes match the same attributes' values of the corresponding `<receive>` communication. The response message is passed on the `output_message` variable.

3. **invoke:** In *invoke* communication, the service provider receives a message from the service consumer and should in his turn return a message in response. The receiving message is carried on the *input message*, whereas the response is returned the *output message*. In order to invoke a provider service, `<invoke>` element is used. Here, the consumer service is identified

by the `consumer_instance` and its respective consumer interface `consumer_interface`, whereas the provider service is identified by `provider_instance` and its respective `provider_interface`. The concrete operation which is called at the provider's side is specified by the `<operation>` element. The invoke element as a bi-directional activity, specifies both input and output variables by respectively `input_message` and `output_message` attributes.

Figure 4.9 illustrates the three communication message types. In (a) and (b), the uni-directional communications (send / receive) are displayed, whereas in (c) the bi-directional invoke communication is displayed between two services.

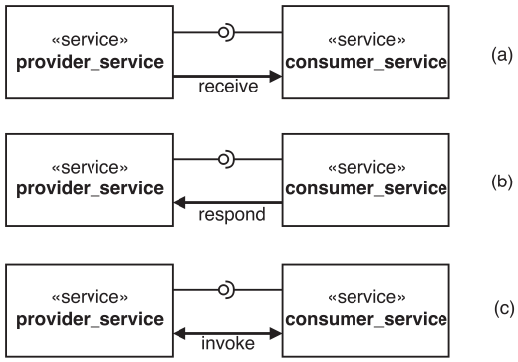


Figure 4.9: Inter-service communicating activity types

4.2.5.2 Configuration Description

The description of reconfiguration in DSOPL-ADL is based on the description of variability. This implies three issues:

- The variability of a system (variation points with all alternatives and their constraints) is described separately in an independent manner.
- In the re-configuration section, in addition to describing the behavior of the common part of the architecture, all possible configurations of variation points are described. This architecture is called a reference architecture.
- A concrete dynamic architecture is generated from the reference architecture according to context changes.

The configuration description section of DSOPL-ADL has a *(i) common configuration description* part, where common services of the architecture are instantiated and bound with a certain behavior and a *(ii) variable configuration description* part, where partial configurations describe the behavior of variation points. Next, each of these parts are detailed. Figure 4.10, presents the meta-model of configuration

description and listing 4.7 defines the configuration description language's specification.

Common Configuration Description

Common configuration description section contains two subsections: *initialization* and *behavior*. The *initialization* sub-section describes the entire structural information about the common services of the architecture. Here, all common services (i.e. those that are not subject to variability), are instantiated in `services` element and bound through `binding` elements. `deployable service instance` is used to create an instance with a name (referred as `service_instance` in the listing 4.8) from a particular service (referred as `service` in the listing 4.8). The `binding` part has two references to two different service instance interfaces, the one that calls an operation `consumer_interface` and the one that provides the operation `provider_interface`.

In the *behavior* sub-section, we describe the workflow of the entire architecture and shade the parts of the architecture that are subject to variability. The workflow is described in form of a sequence of activities between services. We distinguish 3 types of activities: receive, respond and invoke. The differences between those activities are explained in section 4.2.5.1. In each activity we identify the direction of flow by identifying consumer and provider instances and specifying the interfaces that will communicate from each side of consumer and provider instances in order to execute a particular operation. The information that is exchanged between consumer and provider services are carried through messages.

Variable Configuration Description

The second part of the configuration description is the *variable configuration description* which contains several *partial configurations*. Each partial configuration refers to an alternative point that is defined in the variability description module. A variable partial configuration is triggered by conditions that are specified in the `condition` part of the partial configuration rule. Any `partial_configuration` has two sections:

1. *condition* part: where we specify the condition that is driven by context elements. Once the condition satisfied, a given behavior is selected. In case several conditions are satisfied, alternative with higher priority is privileged to integrate.

2. *behavior* part: where we specify all dynamic activities that will be executed in order to integrate the concerning alternative to the existing architecture.

The behavior part of variable configuration description will first instantiate an instance of the reference alternative service, bind it to the existing common architecture and finally interact with the system by executing certain activities. In case

several conditions are satisfied, alternative with higher priority is privileged to integrate.

The specification of configuration description is given in listing 4.7. We choose to specify it in REgular LAnguage for XML, New Generation (RELAX NG) [Clark 2001], an OASIS standard schema language for XML. Relax NG comes in two versions; an XML syntax version and a compact non-XML syntax version. We choose to specify the DSOPL-ADL configuration description using RELAX NG compact version due to its simplicity and expressiveness; unlike other schema languages (e.g. XML Schema) it has a clean formal model.

Listing 4.7: Configuration description Specification

```
start =
  element configuration_description {
    element common_configuration_description {
      element initialization { services, bindings },
      element behavior {(invoke | receive | respond | element decision_point {
          attribute variability_reference})+ }
    },
    element variable_configuration_description {
      element partial_configuration {
        element condition {
          element context { element name, element value }
        },
        element behavior { services, bindings, (invoke | receive | respond)+ }
      }+
    }
 }
services =
  element services {
    element deployable_service_instance {
      attribute service,
      attribute service_instance
    }+
  }
bindings =
  element bindings {
    element binding {
      attribute consumer_instance,
      attribute consumer_interface,
      attribute provider_instance,
      attribute provider_interface
    }+
  }
receive =
  element receive {
    attribute consumer_instance,
    attribute consumer_interface,
```

```
    attribute input_message,
    attribute name,
    attribute operation,
    attribute provider_instance,
    attribute provider_interface
  }
respond =
  element respond {
    attribute consumer_instance,
    attribute consumer_interface,
    attribute name,
    attribute operation,
    attribute output_message,
    attribute provider_instance,
    attribute provider_interface
  }
invoke =
  element invoke {
    attribute consumer_instance,
    attribute consumer_interface,
    attribute input_message,
    attribute name,
    attribute operation,
    attribute output_message,
    attribute provider_instance,
    attribute provider_interface
  }
```

Dynamic Re-configuration - Illustrative Example

In our illustrative on-line sales example of figure 4.1, both "customer" and composite "supply chain management" services make part of the common architecture, since they are not subject to any variation. This implies that their instantiations and bindings can be specified at design time, as depicted in the `common_configuration_description` part of listing 4.8. In fact, instances of their services are instantiated using the `<deployable_service_instance>` element. Whereas in `<variable_configuration_description>` part, either "relay point shipping" service or "home delivery shipping" service is dynamically instantiated at run-time depending on the context value of `shipping`.

In the `<common_configuration_description>` part, next, instantiated services are bound together through their interfaces in the `<binding>` part. In the `<behavior>` part, the workflow starts by a trigger activity from the "customer" service, which makes a sales order. The "supply chain management" service receives the ordered items from the `input_message = "receive_order_items"` of `<receive>` activity. The operation `operation = "prepare_order"` is called in the "supply chain management" service.

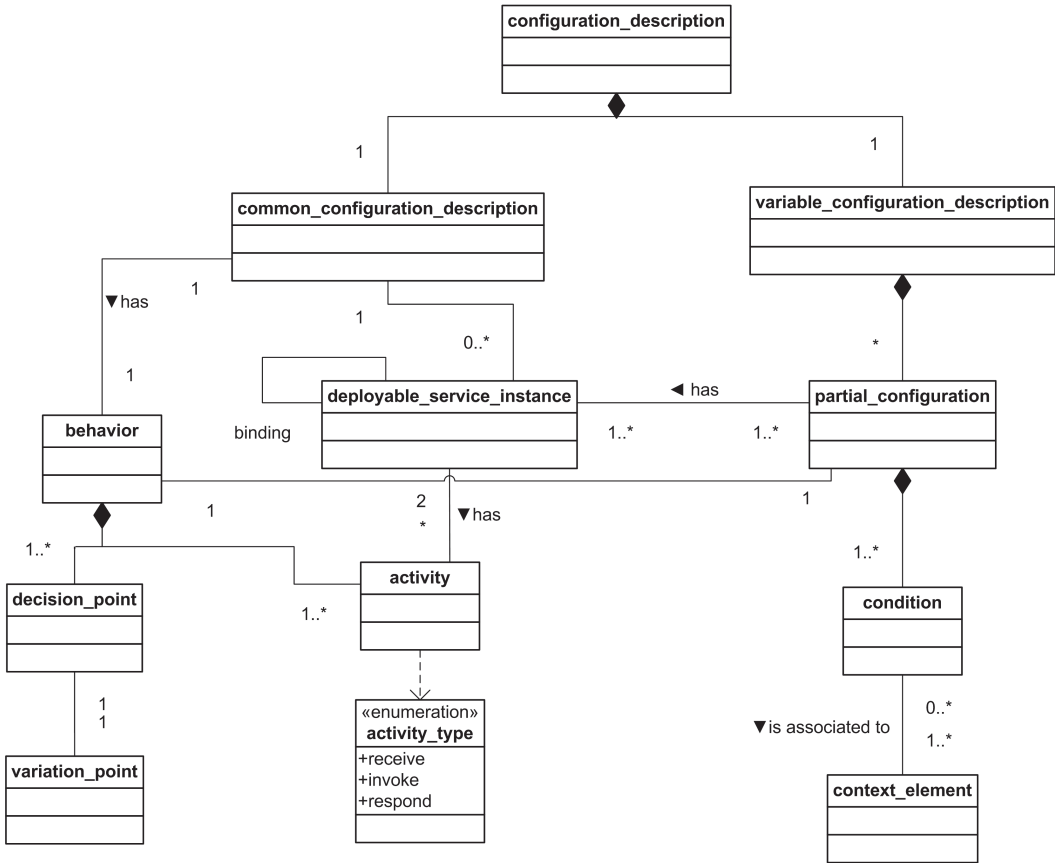


Figure 4.10: Configuration description meta-model of DSOPL-ADL

Next, a <decision_point> indicates the existence of a variation point with a reference to the "shipping_variation_point" which is explicitly defined in the *variability description* part of DSOPL-ADL. The <decision_point> part of the behavior will not be executed at design time but at run-time. The different partial configurations related to each variation point are described in the <variable_configuration_description> part of the *configuration description* of DSOPL-ADL. All related instructions of instantiating and binding either "home delivery shipping" service or "relay point shipping" service to the "supply chain management" service will be performed only at run-time according to the context value of "shipping" in the <condition> part of each <partial_configuration>. Once the corresponding shipping service is bound, "supply chain management" service will invoke one of these shipping services and execute either "order_delivery_to_home" or "order_delivery_to_relay_point" operations. Obviously, information about ordered items are passed from the "supply chain management" service to selected shipping service through the message input_message = "in_ship_order". Likewise, shipping information (such as shipping delays and costs) are returned through the message output_message = "out_ship_order".

Finally, as a respond to customer's initial order request, the "supply chain management" service executes the "prepare_order" operation and sends the order details to the "customer" service through an output_message = "out_order_details" within a <respond> activity. Here, both service instances (consumer_instance and provider_instance) as well as their interfaces (consumer_interface and provider_interface) are the same of the <receive> activity, since it is a respond to that request. Listing 4.8 displays in detail sales example's configuration and behavioral description and figure 4.11 displays graphically the sequence of activities and message communications between involved services.

Listing 4.8: Configuration description of sales scenario

```
<configuration_description>
 <common_configuration_description>
  <initialization>
   <services>
       <deployable_service_instance service="customer_service" service_instance="
           customer_service_instance"/>
       <deployable_service_instance service="supply_chain_management_service" .../>
   </services>
   <bindings>
    <binding consumer_instance="customer_service_instance" consumer_interface="
        i_customer_order" provider_instance="
        supply_chain_management_service_instance" provider_interface="
        i_order_delegation" />
   </bindings>
  </initialization>
```

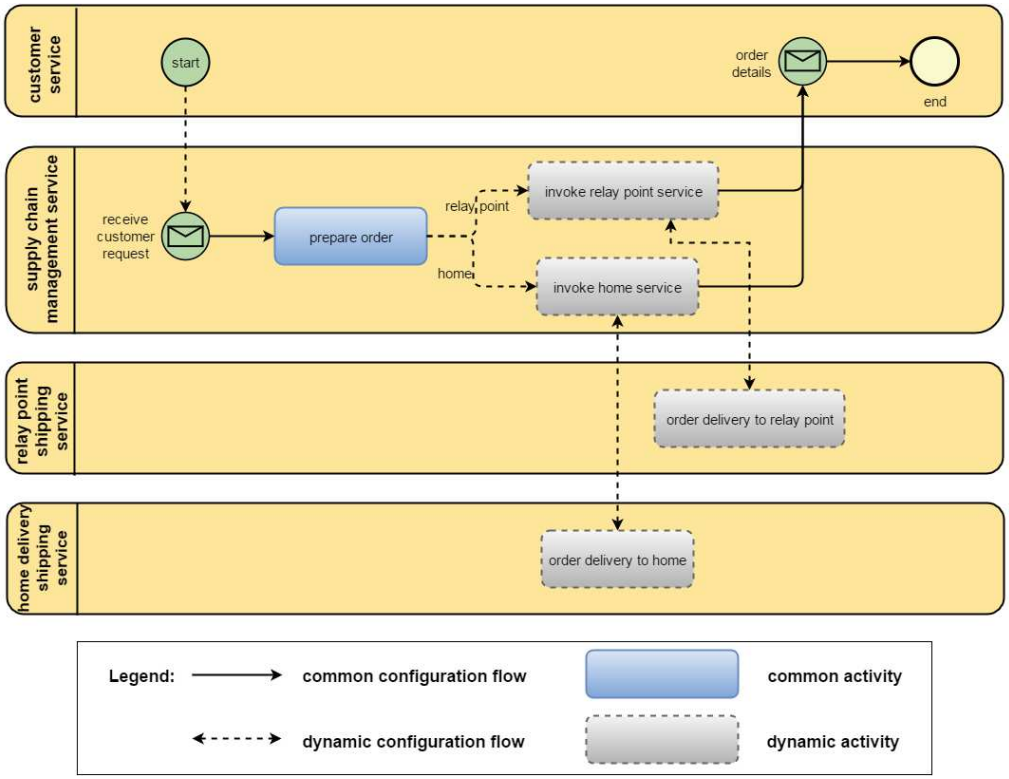


Figure 4.11: On-line sales scenario behavioral description

```
 <behavior>
  <receive name="receive_customer_request" consumer_instance="
      customer_service_instance" consumer_interface="i_customer_order"
      provider_instance="supply_chain_management_service_instance"
      provider_interface="i_order_delegation" operation="prepare_order"
      input_message="receive_order_items">
  </receive>

  <decision_point variability_reference="shipping_variation_point"/>

  <respond name="send_order_details" consumer_instance="customer_service_instance"
       consumer_interface="i_customer_order" provider_instance="
      supply_chain_management_service_instance" provider_interface="
      i_order_delegation" operation="prepare_order" output_message="
      out_order_details">
  </respond>
 </behavior>
</common_configuration_description>

<variable_configuration_description>
 <partial_configuration>
  <condition>
      <context>
       <name> shipping </name>
       <value> home </value>
      </context>
  </condition>
  <behavior>
   <services>
       <deployable_service_instance service="home_delivery_shipping_service"
           service_instance="home_delivery_shipping_service_instance" />
   </services>
   <bindings>
    <binding consumer_instance="supply_chain_management_service_instance"
        consumer_interface="i_shipment_ready_delegation" provider_instance="
        home_delivery_shipping_service_instance" provider_interface="
        i_home_delivery" />
   </bindings>
   <invoke name="invoke_home_delivery_service" consumer_instance="
       supply_chain_management_service_instance" consumer_interface="
       i_shipment_ready_delegation" provider_instance="
       home_delivery_shipping_service_instance" provider_interface="
       i_home_delivery" operation="order_delivery_to_home" input_message="in_ship-
       order" output_message="out_ship_order">
   </invoke>
  </behavior>
 </partial_configuration>
 <partial_configuration>
  <condition>
      <context>
       <name> shipping </name>
```

```
        <value> relay_point </value>
       </context>
  </condition>
  <behavior>
       ...
        <deployable_service_instance service="relay_point_delivery_shipping_service
            " ... />
       <bindings .../>
       <invoke .../>
  </behavior>
 </partial_configuration>
</variable_configuration_description>
</configuration_description>
```

4.3 Concrete Architecture and Executable Code Generation

4.3.1 Concrete Architecture Generation

A concrete architecture is generated from a given reference architecture <configuration_description> through the following consecutive steps:

1. The <common_configuration_description> part of the configuration description is copied to the concrete architecture description without any modifications, except copying <decision_point> which is treated differently in step 2.

2. Each variation point called <decision_point> in the <common_configuration_description> part of our ADL is replaced by an appropriate <partial_configuration> according to context value satisfaction of that partial configuration.

3. Consequently, all service instances related to that partial configuration in addition to their bindings both described in <services> and <binding> subsections of <partial_configuration> are integrated to the concrete architecture to <services> and <binding> sections, respectively.

Figure 4.12 demonstrates how a concrete architecture is generated after integrating a partial configuration to the common configuration part.

4.3.2 Executable Code Generation

In this section, we demonstrate the generation of an executable business process from the DSOPL concrete architecture specifications. Among existing business process specification languages, we choose to generate DSOPL's architectural description by means of Business Process Execution Language (BPEL), since it is the most dominant language [Griffiths 2010] and has become a de-facto standard for specifying

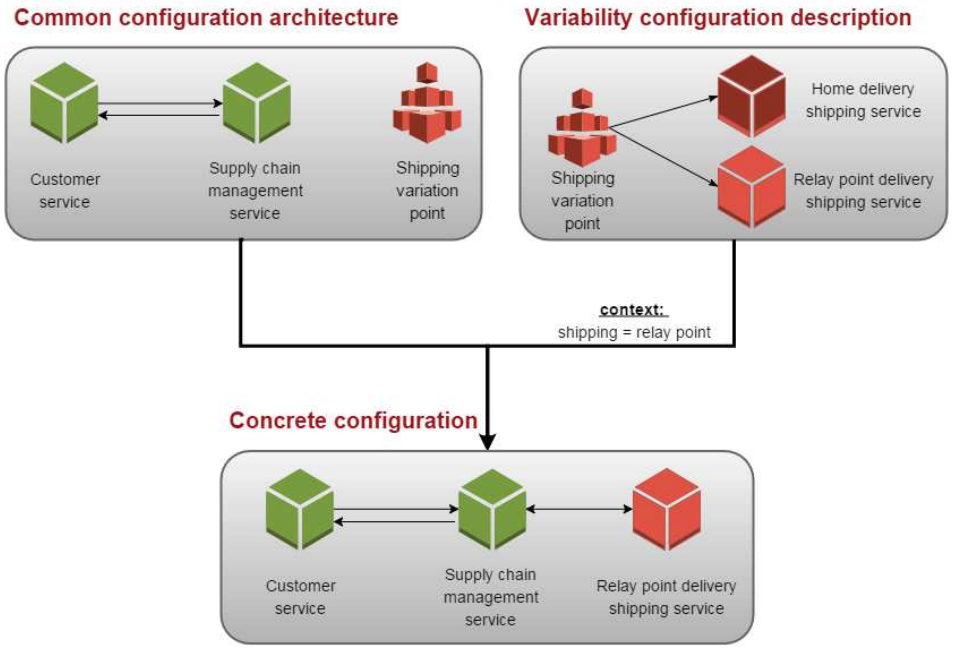


Figure 4.12: Concrete architecture generation

workflow in a service-oriented environment and hence executing business processes for web services composition [Albreshne 2009]. BPEL defines in principle two main types of activities: basic activities to interact with external services (invoke, receive, reply) and structural activities to control the internal business workflow by conditional choices, parallel activities and looping.

The most challenging part in BPEL's code generation is the workflow transformation of DSOPL-ADL's behavior into BPEL's activities within the sequence section due to the difference of workflow realization between both paradigms. Conceptually, there are two different perspectives of realizing a workflow: choreography and orchestration. In choreography, the configuration is realized between autonomous peer-to-peer collaborations, whereas in orchestration a single central workflow engine coordinates the execution flow between all involved Web services. In DSOPL-ADL, configuration is specified following the choreography perspective; i.e. configuration is realized through message passing from one service to another one without intermediaries. Figure 4.13 demonstrates the sequence of activities for the sales order concrete architecture. BPEL, in contrast, supports both choreography and orchestration perspectives [Juric 2007]. However, in order to obtain an executable process, orchestration perspective should be followed [Juric 2007]. That is why an adaptation should be done and transformation rules should be defined to accompany the transformation of DSOPL to orchestration perspective.

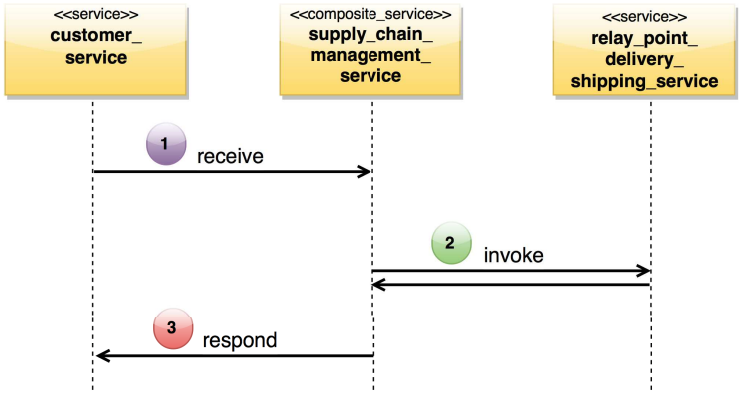


Figure 4.13: Sales order activities' sequence

The activities of behavioral description of DSOPL-ADL are similar to those of BPEL. Therefore, we propose a mapping between DSOPL-ADL concepts at architecture level and BPEL concepts at implementation level. Table 4.1 displays the concepts of our DSOPL-ADL and their corresponding concepts in BPEL paradigm. The purpose of this mapping is to generate a BPEL skeleton from DSOPL-ADL description. We can notice the existence of one-to-one mapping between both paradigms (such as the structural element "deployable_service_instance" in DSOPL-ADL is mapped to "partnerLink" in BPEL), one-to-many mapping (such as the activity "receive" in DSOPL-ADL is mapped to four consecutive activities in BPEL) and unfortunately there are concepts that do not have any correspondence in one of the two paradigms (such as the "binding" which does not have any correspondence in BPEL or "variables" in BPEL which do not have any correspondence in DSOPL-ADL).

According to the mapping rules presented in table 4.1, the group of sequential activities in the concrete architecture of sales order example (represented graphically in figure 4.13) are transformed to BPEL activities as demonstrated in figure 4.14. We can notice that a new actor is added called "Sales Order Process" which is a central process that coordinates the message passing and order of flow between services called "Partner Links". This central process must be either a consumer or provider part in all BPEL activities, this explains why each DSOPL-ADL activity is mapped to one or more activities in BPEL. The "receive" activity in DSOPL-ADL (designated by number 1) which receives customer's order and prepares it is represented by four activities in "Sales Order Process": receive, assign_customerRequest, invoke_prepareOrder and assign_orderInformation (all designated by number 1 too). The bi-directional activity "invoke" in DSOPL-ADL (designated by num-

ber 2 in figure 4.13) is transformed to two activities in "Sales Order Process": invoke_OrderDeliveryToRelayPoint and assign_CustomerOutput. This latter copies the content of returned message from relay_point_delivery_shipping_service and assigns it to a local temporary variable. Finally, the "respond" activity (designated by number 3 in figure 4.13), which is in charge to return order's confirmation and information about shipping to the customer, is transformed to the "reply" activity in BPEL (designated by 3 in figure 4.14).

DSOPL-ADL behavioral concepts	**BPEL concepts**
structural elements	
deployable_service_instance	partnerLink
operation	role's name & invoke's name
behavior	sequence
interactive activities	
receive	4 consecutive activities: receive, assign, invoke and assign
invoke	2 consecutive activities: invoke and assign
respond	reply
examples of missing correspondence	
binding	-
-	variables

Table 4.1: DSOPL-ADL to BPEL mapping

4.4 Conclusion

We have presented DSOPL-ADL, an architecture description language that allows the reconfiguration of a software architecture at runtime based on variability description. To manage the runtime reconfiguration at architectural level, we have proposed a modular language called DSOPL-ADL which is structured in and composed of four sections; structural, variability, context and configuration. For each part, its meta-model was presented and discussed in detail through an illustrative example. Moreover the architecture can adapt its behavior to environment changes that are specified as context elements and consequently generate at runtime an adequate concrete architecture from a given reference architecture. Finally, the concrete architecture is transformed to an executable language (BPEL).

In chapter 5, we demonstrate the generated concrete architecture of sales order example. Furthermore, we demonstrate the transformation of that concrete architecture to executable business process (BPEL).

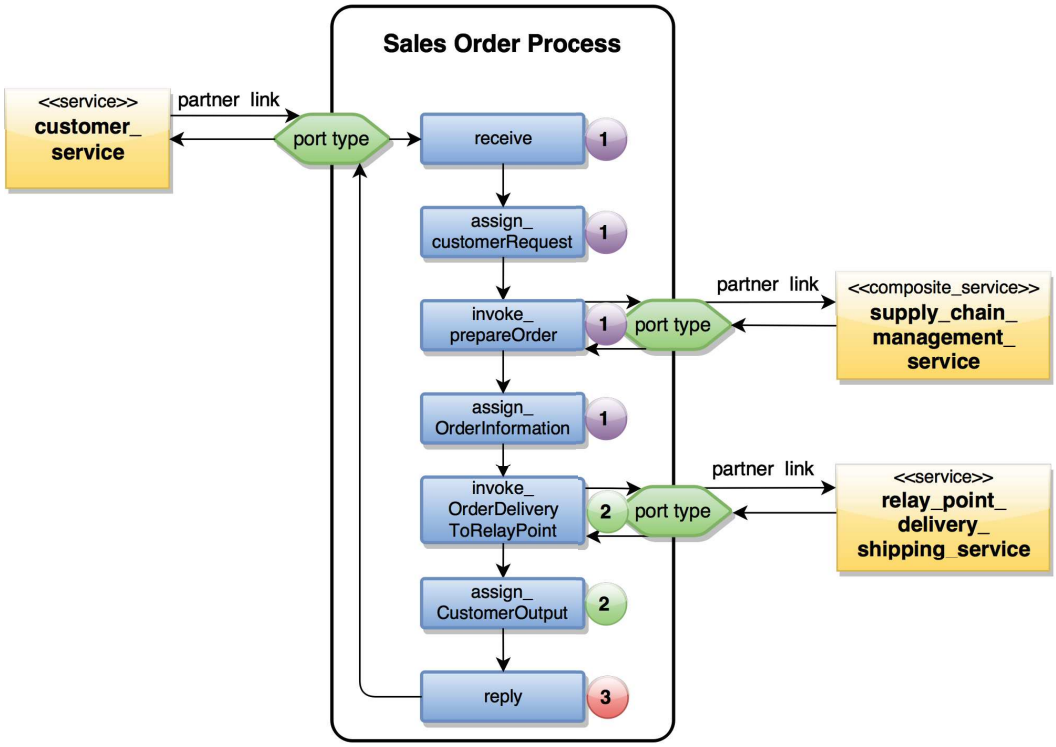


Figure 4.14: Transformation to BPEL - sales order example

Chapter 5

Experimentation/ Validation

Contents

5.1 Introduction

In this chapter, we present the experimentation done on case studies to validate our service identification and packaging approach. This chapter is organized as follows: in section 5.2, we validate our service identification approach on four small and medium-sized case studies of Java OO applications. In section 5.3, we present the service packaging and deployment steps from service deployment until its interface generation. In section 5.4, we demonstrate the results of a concrete architecture and finally its transformation to BPEL in section 5.5.

5.2 Service Identification from Object-Oriented Classes

In order to demonstrate the applicability of our proposed approach, we evaluate it on the following Java OO applications as case studies:

- Java Calculator Suite [Fegler 2013], which is a small system with 17 classes, is an open-source calculator implemented in Java. It performs basic mathematical operations, has a graphic interface and supports Booleans, large numbers, machine numbers, and about 25 different operations.

- MobileMedia [Figueiredo 2007], which is a small sized system with 51 classes, is an open-source Java application used for managing media (photo, music, and video) on mobile devices.
- Galleon TiVo Media Server [Nicholls 2009] is a home media server. It is an open source project distributed under the GNU GPL.
- Log4j [Grobmeier 2015] is a popular logging package which is widely adopted and used in many applications. This audit framework is certified by the open source initiative.

Table 5.1 displays the case studies in terms of their used versions, number of classes and KLOC (thousands of lines of code).

Case Study	**Version**	**Number of classes**	**Code Size (KLOC)**
Java Calculator Suite	2013-04-10	17	2,360
MobileMedia	R7	51	3,016
Galleon TiVo Media Server	2.5.5	137	26
Log4j	1.2.17	220	21

Table 5.1: Case studies information

5.2.1 Service Identification Results

In this phase, we partition the source code of each case study into a set of clusters. Each cluster is composed of one or more classes. Each resulting cluster corresponds to one service. Table 5.2 shows the results in terms of number of obtained services for each case study and the corresponding average service quality value for each of the three characteristics: functionality, composability and self-containment. The distribution rate of classes to services is 17/7= 2,4 classes per service for Java Calculator Suite and 3,9 for MobileMedia. Even more, we notice that almost all classes of same service are grouped to offer single functionality. For example, in Java Calculator Suite case study, "Entries", "GuiCommandLine" and "ResultList" handle I/O issues.

5.2.2 Results and Validation

We validate the consistency of our proposed service identification approach either by comparing resulting services with the known architectural design or by analyzing the relevance of the identified services. We will give an example of the usage of each of these techniques on our studied use cases.

Case study	Number of services	Functionality	Composability	Self-containment
Java Calculator Suite	7	0,73	0,88	0,41
MobileMedia	13	0,60	0,79	0,59
Galleon TiVo Media Server	16	0,73	0,62	0,43
Log4j	23	0,69	0,56	0,39

Table 5.2: Service identification results

For example, in Java Calculator Suite use case, since no architecture design was available to compare it with our results, we have manually identified the architecture based on our knowledge about the system and its functionalities. Then, we compared services of this architecture with our automatic service identification results. We noticed that classes that provided similar functionalities were grouped in the same cluster. Figure 5.1 displays our obtained results of partitioning java classes into clusters by applying our service identification approach. For example, "Entries", "GuiCommandLine" and "ResultList" are three classes that were grouped in one cluster.

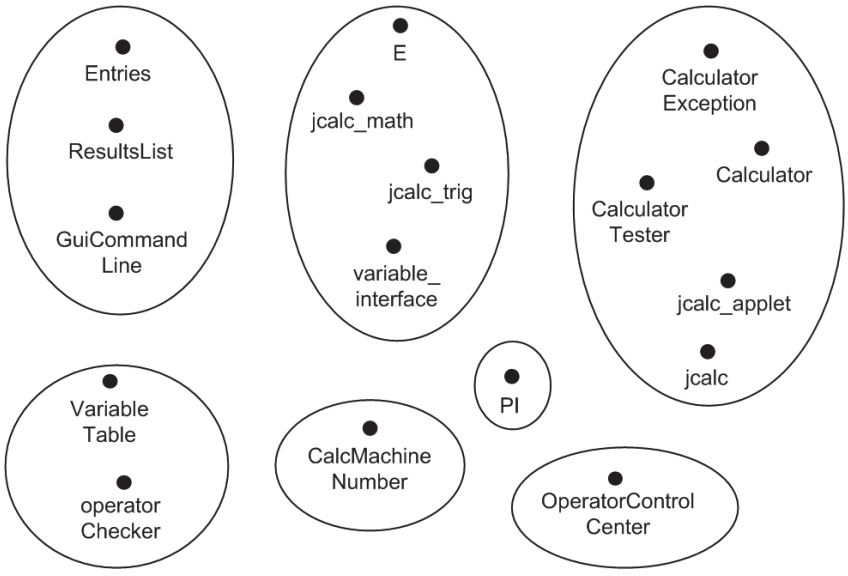


Figure 5.1: Java Calculator Suite service identification result

The case study of MobileMedia has a known architecture model, we therefore apply

the technique of comparing our identification results with the architecture design. In [Figueiredo 2008], the authors presented aspect oriented architecture for MobileMedia. We manually compare our extracted services with the modules of this design, after excluding aspect modules. We have found out that some services were directly mapped to one corresponding module in the architecture, such as the service that includes two classes "MediaListScreen" and "MediaData" was mapped to the module named "MediaListScreen". In total, 5 services were successfully mapped to 5 modules. Some other extracted services could be mapped to more than one module. This category can be divided to two types. The first type is one module with closely related functionalities such as the service named "Video Media Util Screen Play Capture Music" was mapped to three modules "PlayMediaScreen", "VideoAccessor" and "VideoAlbumData". These three modules are in fact functionally related and the resulted service was more coarse-grained than the architecture design. The second type is modules that are weakly related. For this case, we have found two services that each of them was mapped to respectively 3 and 4 modules of the architecture. Some services that are functionally closely related (in our case study, 4 services related to the functionality of transferring media via SMS) were mapped to many modules of the architecture (in our case study, 2 modules related to media transfer functionality). These extracted services were finer-grained than their corresponding modules. Finally, one service that groups exception classes is missing from the architectural design since in the architecture, non-functional modules are not represented.

The results show that 77% (10/13) of extracted services were successfully mapped in the architectural design.

5.3 Service Packaging and Deployment

After having presented, in section 5.2, the experimentation related to the service identification phase, we present in this section our experimentation related to the packaging and deployment of the identified services. As a reminder, the packaging phase includes transforming clusters to become interface-based through setting up a delegator class for each identified cluster (i.e. service), annotating this delegator class and building a service interface.

5.3.1 Preparing Service Creation

We have considered that our deployable units are Web services. Web service is a particular type of service, where its description is published in Web service Description Language (WSDL). We used Eclipse environment to create the corresponding WSDL file of each delegator class file and deploy it using Apache Axis 1.4 server. A WSDL interface can be created automatically only from one class which represents

the service implementation. Thus, we need to create for each cluster, a new class that exposes all provided methods of its composing classes.

For each cluster, we created a new java class in form of a "delegator class" (see figure 5.2b). We placed in this class all provided methods' signatures of interface classes. As a reminder, an interface class is a class of which at least one method is invoked from other classes outside the cluster's boundaries. Methods of this delegator class do not implement the functionality, instead, they only delegate its execution to the same method of the appropriate class after instantiating an object of that class (see figure 5.2b). Listing 5.1 displays the implementation of "GuiResultsEntriesCommandList" delegator class in Java Calculator Suite case study. This cluster regroups three classes: "Entries", "GuiCommandLine" and "ResultList", as you can notice in table 5.3 (cluster number 7) and figure 5.2. In this delegator class, we first instantiated objects from each interface class, therefore we called the constructors of "Entries" and "ResultsList" classes. Then, we added the signatures of methods that are invoked from outside this cluster. In listing 5.1 we can note that six methods of "Entries" class were invoked from "Calculator" class, and one method "setCommandLine" from "ResultsList" was invoked by "jcalc" class. Some methods within "GuiCommandLine" class were invoked by "ResultsList" class, but since these two classes are part of the same cluster, we do not represent this method invocation in the delegator class. "GuiCommandLine" has no other connections, that is why it is considered as internal class and not as an interface class. In the implementation of each provided method of the delegator class, we invoke the same method of the interface class using an instance of that interface class, as you can notice in listing 5.1.

Figure 5.2a displays the dependencies between two clusters of Java Calculator Suite before setting up a delegator class, whereas in figure 5.2b we display the connections using the delegation class "GuiResultsEntriesCommandList". Dotted arrows between delegator class and delegatee classes in figure 5.2b represent the delegation of execution to concrete implementation classes.

Listing 5.1: GuiResultsEntriesCommandList delegator class implementation

```
public class GuiResultsEntriesCommandList {

        private Calculator c;

        // instantiations of interface classes
        Entries entries = new Entries();
        ResultsList results = new ResultsList(c);
```

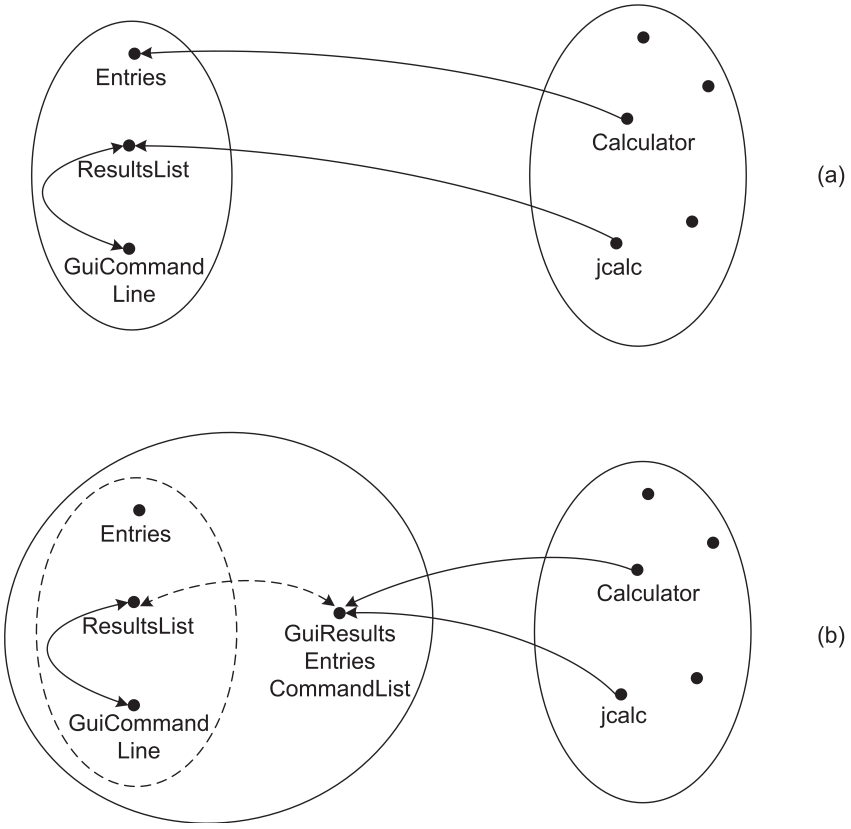


Figure 5.2: Java Calculator Suite partial call graph

```
        // provided methods of ResultsList class
        public void setCommandLine(GuiCommandLine cl){

        }
        // provided methods of Entries class
        public void clear(){
                entries.clear();
        }

        public void delete(int i) throws CalculatorException {
                entries.delete(i);
        }

        public String getAns(int i) throws CalculatorException {
                return entries.getAns(i);
        }

         public String getEntry(int i) throws CalculatorException {
                 return entries.getEntry(i);
         }

         public Vector getAllEntries(){
                 return entries.getAllEntries();
         }

         public void addEntry(String equation, String results){
                 entries.addEntry(equation, results);
         }

         // GuiCommandLine
                 // has no provided methods to external clusters

}
```

As to the classes that invoke those provided methods, we manually transformed the references from the interface class to the newly created delegator class. For example, in listing 5.2, we demonstrate a portion of "Calculator" class. This class was previously instantiating an object of "Entries". After setting up the "GuiResultsEntriesCommandList" delegator class, we changed the declaration of object instantiation to become an instantiation of "GuiResultsEntriesCommandList" class.

It is worth noting that we consider that services are stateless and not stateful,

this implies that for each request of an operation from a Web service client, a new instance is created.

Listing 5.2: Calculator class implementation

```
public class Calculator {
        ...
        public GuiResultsEntriesCommandList entries = new
            GuiResultsEntriesCommandList();

        // the rest of the implementation remains unchanged
        ...
}
```

5.3.2 Service Annotation

We document the resulting components by assigning a name based on the annotation algorithm with most frequent terms. In table 5.3, we display the classes' names of Java Calculator Suite case study distributed in clusters. We extracted the frequent terms and calculated the number of occurrence of each term (this number is displayed beside each term between parentheses). For example, in cluster 3, the term "Calculator" is repeated three times on the first position, this is why the cluster name will first be composed of that term. In the same cluster, "Exception" and "Tester" are programming language special words, that is why they were excluded during the terms extraction. We also notice that in cluster 7 that all terms occur once. We compose the cluster name with terms that are on the first position "Gui", "Results" and "Entries", followed by terms that are on the second position "Command" and "List". We fixed the maximum number of terms to compose to five, that is why the term "Line" was ignored within the cluster annotation.

5.3.3 Service Interface Generation

We automatically generated Web service interface in WSDL for each service using Eclipse Web service plug-ins. Listing 5.3 displays a simplified Web service interface generation in WSDL for "GuiResultsEntriesCommandList" service of Java Calculator Suite case study. All seven methods of delegator class are exposed as operations in WSDL such as `<wsdl:operation name="getAllEntries">` which represents `public Vector getAllEntries()` method of "GuiResultsEntriesCommandList" class.

Cluster Number	Composing Classes	Frequent terms	Cluster Name
1	CalcMachineNumber	Calc(1) Machine(1) Number(1)	CalcMachineNumber
2	OperatorControlCenter	Operator(1) Control(1) Center(1)	OperatorControlCenter
3	Calculator CalculatorException CalculatorTester jcalc jcalc_applet	Calculator(3) jcalc_applet(1) jcalc(1)	Calculatorjcalc_appletjcalc
4	E jcalc_math jcalc_trig variable_interface	E(1) variable_interface(1) jcalc_math(1) jcalc_trig(1)	Evariable_interface jcalc_mathjcalc_trig
5	VariableTable operatorChecker	Variable(1) operator(1) Checker(1) Table(1)	VariableoperatorCheckerTable
6	PI	PI(1)	PI
7	Entries GuiCommandLine ResultsList	Gui(1) Line(1) Results(1) Entries(1) Command(1) List(1)	GuiResultsEntries CommandList

Table 5.3: Java Calculator Suite services' identification results

Listing 5.3: A simplified WSDL description for GuiResultsEntriesCommandList Web service

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://service7" xmlns:apachesoap="http://xml.
    apache.org/xml-soap" xmlns:impl="http://service7" xmlns:intf="http://service7"
    xmlns:tns1="http://cal" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3
    .org/2001/XMLSchema">
 <wsdl:types>
  <schema elementFormDefault="qualified" targetNamespace="http://service7" xmlns="
      http://www.w3.org/2001/XMLSchema">
   <import namespace="http://cal"/>
   <import namespace="http://xml.apache.org/xml-soap"/>
   <element name="clear">...</element>
   <element name="clearResponse">...</element>
   <element name="delete">...</element>
   <element name="deleteResponse">...</element>
   <element name="fault" type="tns1:CalculatorException"/>
   <element name="addEntry">...</element>
   <element name="addEntryResponse">...</element>
   <element name="getEntry">...</element>
   <element name="getEntryResponse">...</element>
   <element name="getAllEntries">...</element>
   <element name="getAllEntriesResponse">...</element>
   <element name="setCommandLine">...</element>
   <element name="setCommandLineResponse">...</element>
   <element name="getAns">...</element>
   <element name="getAnsResponse">...</element>
```

```
 </schema>
 <schema elementFormDefault="qualified" targetNamespace="http://cal" xmlns="http:
     //www.w3.org/2001/XMLSchema">
  <import namespace="http://xml.apache.org/xml-soap"/>
  <complexType name="CalculatorException">
   <sequence>
    <element name="location" type="xsd:int"/>
   </sequence>
  </complexType>
 </schema>
 ...
</wsdl:types>
<wsdl:message name="deleteRequest">
   <wsdl:part element="impl:delete" name="parameters"></wsdl:part>
</wsdl:message>

<wsdl:message name="clearResponse">...</wsdl:message>
<wsdl:message name="getEntryResponse">...</wsdl:message>
<wsdl:message name="getEntryRequest">...</wsdl:message>
<wsdl:message name="deleteResponse">...</wsdl:message>
<wsdl:message name="getAnsRequest">...</wsdl:message>
<wsdl:message name="setCommandLineRequest">...</wsdl:message>
<wsdl:message name="getAllEntriesResponse">...</wsdl:message>
<wsdl:message name="addEntryResponse">...</wsdl:message>
<wsdl:message name="addEntryRequest">...</wsdl:message>
<wsdl:message name="setCommandLineResponse">...</wsdl:message>
<wsdl:message name="getAllEntriesRequest">...</wsdl:message>
<wsdl:message name="getAnsResponse">...</wsdl:message>
<wsdl:message name="clearRequest">...</wsdl:message>
<wsdl:message name="CalculatorException">...</wsdl:message>

<wsdl:portType name="GuiResultsEntriesCommandList">
   <wsdl:operation name="clear">
      <wsdl:input message="impl:clearRequest" name="clearRequest"></wsdl:input>
      <wsdl:output message="impl:clearResponse" name="clearResponse"></wsdl:output>
   </wsdl:operation>
   <wsdl:operation name="delete">...</wsdl:operation>
   <wsdl:operation name="addEntry">...</wsdl:operation>
   <wsdl:operation name="getEntry">...</wsdl:input>...</wsdl:operation>
   <wsdl:operation name="getAllEntries">...</wsdl:operation>
   <wsdl:operation name="setCommandLine">...</wsdl:operation>
   <wsdl:operation name="getAns">...</wsdl:operation>
</wsdl:portType>

<wsdl:binding name="GuiResultsEntriesCommandListSoapBinding" type="
    impl:GuiResultsEntriesCommandList">
    <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/
        http"/>
    <wsdl:operation name="clear">
       <wsdlsoap:operation soapAction=""/>
       <wsdl:input name="clearRequest">
```

```
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="clearResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
      <wsdl:operation name="delete">...</wsdl:operation>
      <wsdl:operation name="addEntry">...</wsdl:operation>
      <wsdl:operation name="getEntry">...</wsdl:operation>
      <wsdl:operation name="getAllEntries">...</wsdl:operation>
      <wsdl:operation name="setCommandLine">...</wsdl:operation>
      <wsdl:operation name="getAns">...</wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="GuiResultsEntriesCommandListService">
    <wsdl:port binding="impl:GuiResultsEntriesCommandListSoapBinding" name="
        GuiResultsEntriesCommandList">
       <wsdlsoap:address location="http://localhost:7537/HelloTest/services/
           GuiResultsEntriesCommandList"/>
    </wsdl:port>
 </wsdl:service>

</wsdl:definitions>
```

5.4 Concrete Architecture Generation of DSOPL-ADL

We generated a concrete architecture of the on-line sales scenario example. The inputs of this phase were:

1. the reference architecture that comprises two possible partial reconfigurations of shipment, either shipment to home or shipment to a relay point. These partial configurations are based on a variation point description named `shipping_variation_point`, which is a service variation that has two alternative services, either `home_delivery_shipping_service` or `relay_point_delivery_shipping_service`.

2. a given context information which states that user requests a sales order with a relay point shipment.

Listing 5.4 demonstrates the concrete architecture that was generated from the configuration description that was described in listing 4.8 of chapter 4. We can notice that this concrete architecture does not contain any variability description or reconfiguration description anymore that is why it is called a concrete architecture. When a context information changes, the same reference architecture is reconfigured to produce another concrete architecture.

Listing 5.4: Concrete Architecture Generation of sales scenario

```
<concrete_architecture_description>
 <services>
  <deployable_service_instance service="customer_service" service_instance="
      customer_service_instance"/>
  <deployable_service_instance service="supply_chain_management_service" .../>
  <deployable_service_instance service="relay_point_delivery_shipping_service"
      service_instance="relay_point_delivery_shipping_service_instance" />
 </services>
 <bindings>
  <binding consumer_instance="customer_service_instance" consumer_interface="
      i_customer_order" provider_instance="supply_chain_management_service_instance
      " provider_interface="i_order_delegation" />
  <binding consumer_instance="supply_chain_management_service_instance"
      consumer_interface="i_shipment_ready_delegation" provider_instance="
      relay_point_delivery_shipping_service_instance" provider_interface="
      i_relay_point_delivery" />
 </bindings>
 <behavior>
  <receive name="receive_customer_request" consumer_instance="
      customer_service_instance" consumer_interface="i_customer_order"
      provider_instance="supply_chain_management_service_instance"
      provider_interface="i_order_delegation" operation="prepare_order"
      input_message="receive_order_items">
  </receive>

  <invoke name="invoke_relay_point_delivery_service" consumer_instance="
      supply_chain_management_service_instance" consumer_interface="
      i_shipment_ready_delegation" provider_instance="
      relay_point_delivery_shipping_service_instance" provider_interface="
      i_relay_point_delivery" operation="order_delivery_to_relay_point"
      input_message="in_ship-order" output_message="out_ship_order">
  </invoke>

  <respond name="send_order_details" consumer_instance="customer_service_instance"
      consumer_interface="i_customer_order" provider_instance="
      supply_chain_management_service_instance" provider_interface="
      i_order_delegation" operation="prepare_order" output_message="
      out_order_details">
  </respond>
 </behavior>
</concrete_architecture_description>
```

5.5 Transformation to Executable Language

In order to transform a concrete configuration specified in DSOPL-ADL to an executable language (e.g. BPEL), we apply the concepts' mapping between DSOPL-ADL and BPEL paradigms. Figure 5.3 displays the sales order concrete architecture implementation result in BPEL process. The skeleton of BPEL code is demon-

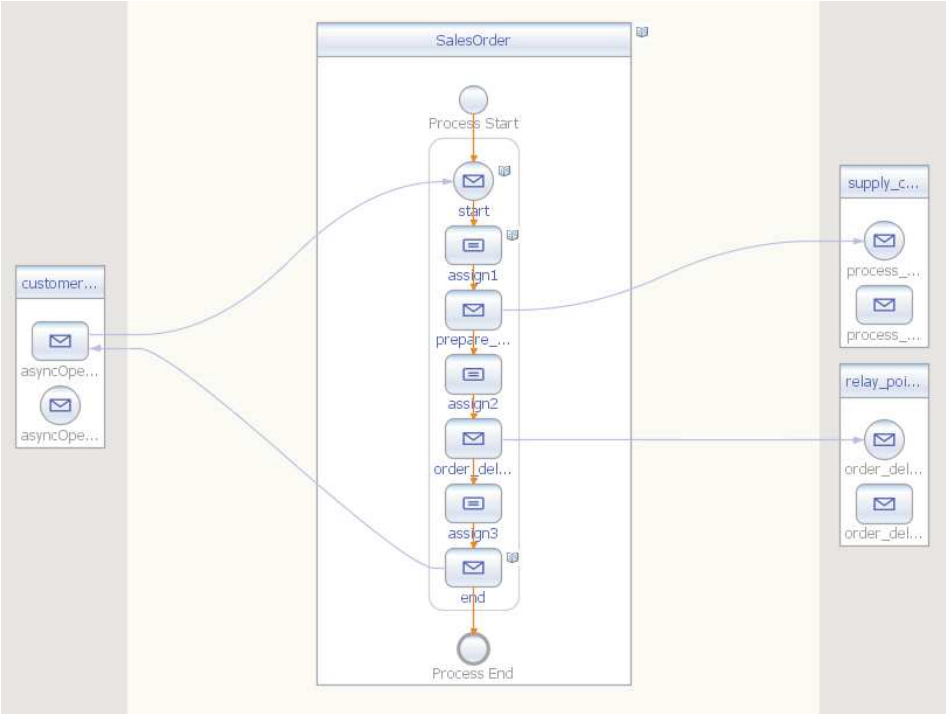


Figure 5.3: Sales order implementation in BPEL

strated in listing 5.5. We can notice that <services> elements of listing 5.4 are transformed to <partnerLinks>. <variables> section defines input and output variable for each partnerLink. For example, Order _delivery_to_relay_pointOut and Order_delivery_to_relay_pointIn are variables to communicate with relay _point_delivery_shipping_service partnerLink. Finally, in <sequence>, we specify the workflow between partnerLinks and the process itself.

Listing 5.5: BPEL skeleton of sales order example

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="SalesOrder" ...>
    <documentation>...</documentation>
    ...
        <!-- import all partner links -->
    <import namespace="http://.../supply_chain_management_WSDL" location="
        supply_chain_management_WSDL.wsdl" importType="http://schemas.xmlsoap.org/
        wsdl/"/>

        <partnerLinks>
        ...
        <partnerLink name="supply_chain_management_service" ... partnerRole="
            order_request"/>
        <partnerLink name="relay_point_delivery_shipping_service" ... partnerRole="
            order_delivery_request"/>
        <partnerLink name="customer_service" ... myRole="request_order" partnerRole=
            "prepare_order"/>
    </partnerLinks>

    <!-- define all variables -->
    <variables>
        <variable name="Order_delivery_to_relay_pointOut" ... />
        <variable name="Order_delivery_to_relay_pointIn" ... />
        <variable name="Process_orderOut" ... />
        <variable name="Process_orderIn" ... />
        <variable name="inputVar" ... />
        <variable name="outputVar" ... />
    </variables>

    <sequence>
        ...
    </sequence>
</process>
```

As to the sequence section, the transformation result of <receive> activity is demonstrated in listing 5.6. It is composed of four activities: receive, assign, invoke and assign.

Listing 5.6: Transformation result of receive activity

```
<sequence>
   <receive name="start" partnerLink="customer_service" ... operation="
       asyncOperation" variable="inputVar" createInstance="yes">
   </receive>
   <assign name="assign1">
      <copy>
         <from variable="inputVar" ... />
         <to variable="Process_orderIn" ...></to>
      </copy>
   </assign>
   <invoke name="prepare_order" partnerLink="supply_chain_management_service"
       operation="prepare_order" ... inputVariable="Process_orderIn" outputVariable
       ="Process_orderOut"/>
   <assign name="assign2">
      <copy>
         <from variable="Process_orderOut" ... />
         <to variable="Order_delivery_to_relay_pointIn" ... />
      </copy>
   </assign>
</sequence>
```

Chapter 6

Conclusion and Future Perspectives

Contents

6.1 Outline

In this chapter, we draw up an overview of the contributions that we have proposed within the frame of this thesis. Additionally, we outline future research directions that may arise from this work.

6.2 Contributions

This work supports describing highly reusable dynamically self-reconfigurable and variant-rich service based software architecture. In this sense, we applied a migration strategy that is divided into two main processes: reverse engineering and forward engineering. As a big picture, we first used reverse engineering techniques to analyze existing legacy system. Once the legacy architecture model identified, we then used forward engineering techniques to describe a new architecture and proceed to obtain executable business processes, which we consider as service implementation. More concretely, we proposed the following contributions:

- **Service identification from legacy source code:** We first identified services embedded in an existing legacy object-oriented system using reverse engineering techniques. On the basis that a service represents one or a group

of classes, we defined a mapping model between object and service concepts in object-oriented and SOA paradigms respectively. Then, in order to identify the best group of disjoint classes that provides a coherent functionality and that is at the same time loosely coupled to other groups of classes, we introduced a measurement function based on service characteristics. Those characteristics were refined to quality properties and then to measurable metrics in order to measure the quality of those identified candidate services. Finally, both clustering and depth first search algorithms were applied respectively to partition legacy classes in services by respecting the fact that classes with a maximized fitness function value were grouped in the same cluster (i.e service). The output of this step was a number of identified services who regrouped each a number of object-oriented classes. It is worth noting that service implementation remained implemented in legacy code. This contribution was experimented on four case studies and obtained results were satisfying in terms of relevancy of identified services to architectural designs.

- **Service packaging:** In order to prepare identified services to become deployable, we adapted clusters to become interface-based. This means that dependencies between classes of different services were lifted to become dependencies between services by setting up for each cluster a new class which acted as an interface and exposed cluster's provided functionalities. We also annotated each cluster and generated its interface. This step was experimented using a case study implemented in Java and Web service interfaces were generated and published in WSDL.

- **Dynamic service-oriented product lines architecture:** We identified a new language grammar of a software architecture that described service-based and variant-rich software artifacts at architecture abstraction level. We called the language "Dynamic Service-Oriented Product Lines Architecture Description Language" (DSOPL-ADL). The goal behind modeling variability description as a first-class element in a service-based architecture was to enable a variability-based reconfiguration of architecture.

- **Dynamic architecture self-reconfiguration based on variability:** Within the process of forward engineering, we composed and configured services using DSOPL-ADL. Building the right composition of services at runtime was subject to environment changes. That is why we also integrated context information to our ADL. This rendered our ADL context-aware. We also described at design time a common architecture in addition to a series of possible service configurations based on variable artifacts in what called a "reference architecture". During system's execution at runtime, a concrete architecture was generated by integrating an appropriate partial configuration to the common architecture part. This concrete architecture satisfied context related con-

straints. Finally, in order to generate a service composition implementation, we mapped DSOPL concrete architecture's concepts to BPEL concepts and transformed architecture to business process language. As to the experimentation, we have generated a concrete architecture of sales order example and then transformed this architecture to a BPEL skeleton.

6.3 Future Perspectives

As a part of future work, our proposed migration process can be either improved or extended in several directions. We classify the future perspectives according to their importance into short-term and long-term perspectives.

6.3.1 Short-term Perspectives

Current work improvements:

- **Considering non-functional characteristics for service identification:** So far, during the service identification phase from legacy source code, only functional characteristics were taken into account for measuring candidate service's quality. We would like to take into account additional service characteristics, in particular related to non-functional service characteristics such as reliability and maintainability.

- **Evaluating our approach with more complex case studies:** Our service identification approach was evaluated on four case studies from which the largest one initially contained 220 classes. We plan to evaluate our approach using real industrial complex case studies. In addition, all case studies that we have used were implemented in Java. We plan to generalize our approach on other case studies implemented in other programming languages.

- **Automating service packaging process:** Currently, in service packaging phase of legacy software migration, preparing interface-based clusters was carried out manually, whereas service interface generation step was automated.We were able to manually package identified services in a delegate class and it was out of our direct interests to automate the transformation of code to make inter-service communications pass through the delegate class of each service. Manual service packaging would have become definitely more complex in terms of time and effort if we had applied it on sophisticated and large legacy systems. Therefore, as a short-term perspective, we would like to automate the transformation steps in order to reduce cost and errors.

- **Automating transformation to BPEL:** The transformation from architecture level to implementation level was so far achieved manually. BPEL skeleton was later manually completed by concepts for which no mapping was

found in DSOPL-ADL. We would like to automate the transformation procedure as well as the auto-generation of missing code segments.

current work extension:

- **Extract context and variability information from legacy source code using dynamic analysis:** So far, our service identification technique from legacy source code was based on the static analysis of object-oriented classes. This enabled us to extract services and their configuration. Whereas variability and context information was injected by the architect at architecture level. In order to extract variability and context information directly from source and avoid their manual injection, we would like to perform a dynamic analysis on the legacy source code objects during system's execution in order to capture variability and context information. Every service in the legacy system which is present during all snapshots is a mandatory service, while every service which appears in certain snapshots is represented as a variant in the DSOPL architecture. Its conditions of taking part in system's architecture is also captured, analyzed and documented as context and constraint information. This renders the migration process more automatic without architect's intervention.

6.3.2 Long-term Perspectives

- **Transform DSOPL architecture into cloud-based architecture:** Cloud computing has recently gained significant importance for the fact that it moved the platform power from local devices into the cloud. The software as a service is from now on hosted on the cloud and runs in user's browser smoothly without any installation overhead or infrastructure cost. Consequently, availability of resources within the cloud is subject to continuous change. In order to deliver solutions as high-quality cloud-services, service providers should dispose complex software architectures to adapt their dynamic reconfiguration to suit the availability of current resources. As a long-term perspective, we would like to draw attention on the aspects that are required to change in current service-oriented architectures in order to port them on cloud.

List of Publications

International and European Conferences:

- Seza Adjoyan, Abdelhak-Djamel Seriai, *"Reconfigurable Service-Based Architecture Based on Variability Description"*. 10th European Conference on Software Architecture - Track on Woman in Software Architecture ECSA 2016, Istanbul, Turkey, 5-9 September, 2016. *(submitted)*

- Seza Adjoyan, Abdelhak-Djamel Seriai, *"An Architecture Description Language for Dynamic Service-Oriented Product Lines"*. 27th International Conference on Software Engineering and Knowledge Engineering SEKE 2015, Pittsburgh, USA, 6-8 July, 2015.

- Seza Adjoyan, Abdelhak-Djamel Seriai, Anas Shatnawi, *"Service Identification Based on Quality Metrics - Object-Oriented Legacy System Migration Towards SOA"*. 26th International Conference on Software Engineering and Knowledge Engineering SEKE 2014, Vancouver, BC, Canada, 1-3 July, 2014.

National Conference:

- Seza Adjoyan, Abdelhak-Djamel Seriai, Anas Shatnawi, *Service Identification Based on Quality Metrics - Object-Oriented Legacy System Migration Towards SOA*. accepted article at SEKE 2014. 3ème Conférence en IngénieriE du Logiciel CIEL et 8ème Conférence francophone sur l'Architecture Logicielle CAL, Paris, 10-12 June, 2014.

Bibliography

[Abu-Matar 2011] M. Abu-Matar and H. Gomaa. *Feature Based Variability for Service Oriented Architectures.* In Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on, pages 302–309, June 2011. (Cited on pages 2, 11, 26, 31 and 33.)

[Akkiraju 2005] Rama Akkiraju and Farrell Joel. *Web Service Semantics - WSDL-S.* `https://www.w3.org/Submission/WSDL-S/`, 2005. [Online; accessed 06-May-2016]. (Cited on page 23.)

[Albreshne 2009] Abdaladhem Albreshne, Patrik Fuhrer and Jacques Pasquier. *Web Services Orchestration and Composition*, 2009. (Cited on page 83.)

[Aldris 2013] A. Aldris, A. Nugroho, P. Lago and J. Visser. *Measuring the Degree of Service Orientation in Proprietary SOA Systems.* In Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on, pages 233–244, March 2013. (Cited on page 39.)

[Allen 1998] Robert Allen, Remi Douence and David Garlan. *Specifying and analyzing dynamic software architectures.* In Fundamental Approaches to Software Engineering, pages 21–37. Springer, 1998. (Cited on pages 26, 28 and 29.)

[Almonaies 2010] Asil A Almonaies, James R Cordy and Thomas R Dean. *Legacy System Evolution towards Service-Oriented Architecture.* In International Workshop on SOA Migration and Evolution, IEEE, pages 53–62. Citeseer, 2010. (Cited on page 10.)

[Alshara 2015] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde Lilia Bouziane, Christophe Dony and Anas Shatnawi. *Migrating Large Object-oriented Applications into Component-based Ones: Instantiation and Inheritance Transformation.* In Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, pages 55–64, New York, NY, USA, 2015. ACM. (Cited on page 51.)

[Apel 2013] Sven Apel, Don Batory, Christian Kästner and Gunter Saake. Feature-oriented software product lines: concepts and implementation. Springer Science & Business Media, 2013. (Cited on page 13.)

[Arsanjani 2007] Ali Arsanjani, Liang-Jie Zhang, Michael Ellis, Abdul Allam and Kishore Channabasavaiah. *S3: A Service-Oriented Reference Architecture.* IT Professional, vol. 9, no. 3, pages 10–17, 2007. (Cited on pages xv, 24 and 25.)

[Audrey 2008] Occello Audrey. *Introduction à l'Architecture Orientée Service- module SAR O3 SI3 MIAGE.* http://www.academia.edu/5995272/cours_architecture_orientée_services_SOA, 2008. [Online; accessed 27-August-2015]. (Cited on pages 8 and 15.)

[Bachmann 2001] Felix Bachmann and Len Bass. *Managing Variability in Software Architectures.* SIGSOFT Softw. Eng. Notes, vol. 26, no. 3, pages 126–132, May 2001. (Cited on page 13.)

[Bachmann 2005] Felix Bachmann and Paul Clements. *Variability in Software Product Lines.* Rapport technique CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. (Cited on page 2.)

[Barbosa 2011] Eiji Adachi Barbosa, Thais Batista, Alessandro Garcia and Eduardo Silva. *PL-AspectualACME: An Aspect-oriented Architectural Description Language for Software Product Lines.* In Proceedings of the 5th European Conference on Software Architecture, ECSA'11, pages 139–146, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on pages 25, 26, 31 and 32.)

[Baresi 2012] Luciano Baresi, Sam Guinea and Liliana Pasquale. *Service-Oriented Dynamic Software Product Lines.* Computer, vol. 45, no. 10, pages 42–48, 2012. (Cited on page 32.)

[Bieman 1995] James M Bieman and Byung-Kyoo Kang. *Cohesion and reuse in an object-oriented system.* In ACM SIGSOFT Software Engineering Notes, volume 20, pages 259–262. ACM, 1995. (Cited on page 44.)

[Bisbal 1999] Jesús Bisbal, Deirdre Lawless, Bing Wu and Jane Grimson. *Legacy Information Systems: Issues and Directions.* IEEE Softw., vol. 16, no. 5, pages 103–111, September 1999. (Cited on page 10.)

[Booth 2004] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris and David Orchard. *Web Services Architecture, W3C Working Group.* http://www.w3.org/TR/ws-arch/, 11 February 2004. [Online; accessed 27-August-2015]. (Cited on pages 40 and 48.)

[BPE 2007] *Web Services Business Process Execution Language Version 2.0*, 2007. (Cited on pages 26, 27 and 31.)

[Bradbury 2004] Jeremy S Bradbury. *Organizing definitions and formalisms for dynamic software architectures.* In In Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems Newport. Citeseer, 2004. (Cited on page 27.)

[Brown 2000] Alan W Brown. Large-scale, component-based development, volume 1 of *Object and component technology series*. Prentice Hall PTR Englewood Cliffs, 2000. (Cited on page 9.)

[Brown 2002] Alan Brown, Simon Johnston and Kevin Kelly. *Using service-oriented architecture and component-based development to build web service applications*. 2002. (Cited on pages 2, 12, 24, 40, 41 and 48.)

[Capilla 2014] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés and Mike Hinchey. *An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry*. Journal of Systems and Software, vol. 91, pages 3–23, May 2014. (Cited on pages 2, 31 and 32.)

[Cavalcante 2015] E. Cavalcante, T. Batista and F. Oquendo. *Supporting Dynamic Software Architectures: From Architectural Description to Implementation*. In Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, pages 31–40, May 2015. (Cited on page 26.)

[Cetin 2007] S. Cetin, N. Ilker Altintas, H. Oguztuzun, A.H. Dogru, O. Tufekci and S. Suloglu. *Legacy Migration to Service-Oriented Computing with Mashups*. In Software Engineering Advances, 2007. ICSEA 2007. International Conference on, pages 21–21, Aug 2007. (Cited on pages 8, 15, 18, 19, 20, 22 and 23.)

[Cetina 2008] Carlos Cetina, Vicente Pelechano and Pablo Trinidad. *An Architectural Discussion on DSPL*. In In 2nd International Workshop on Software Product Lines, pages 59–68, 2008. (Cited on page 32.)

[Channabasavaiah 2004] Kishore Channabasavaiah, Kerrie Holley and Edward Tuggle. *Migrating to a service-oriented architecture*. IBM DeveloperWorks, vol. 16, 2004. (Cited on pages 18 and 43.)

[Chen 2005] Feng Chen, Shaoyun Li, Hongji Yang, Ching-Huey Wang and William Cheng-Chung Chu. *Feature analysis for service-oriented reengineering*. In Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific, pages 8–pp. IEEE, 2005. (Cited on pages 2, 18, 21, 22 and 23.)

[Chen 2009] Feng Chen, Zhuopeng Zhang, Jianzhi Li, Jian Kang and Hongji Yang. *Service identification via ontology mapping*. In Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International, volume 1, pages 486–491. IEEE, 2009. (Cited on pages 2, 18, 20, 21 and 22.)

[Clark 2001] James Clark and Makoto Murata. *RELAX NG Specification*. `http://relaxng.org/spec-20011203.html`, 2001. [Online; accessed 06-May-2016]. (Cited on page 76.)

[Classen 2008] Andreas Classen, Arnaud Hubaux, Franciscus Sanen, Eddy Truyen, Jorge Vallejos, Pascal Costanza, Wolfgang De Meuter, Patrick Heymans and Wouter Joosen. *Modelling variability in self-adaptive systems: Towards a research agenda.* In Proceedings of international workshop on modularization, composition and generative techniques for product-line engineering, pages 19–26, 2008. (Cited on pages 1 and 2.)

[Clements 2001] Paul Clements and Linda Northrop. Software product lines: Practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. (Cited on page 13.)

[Clements 2010] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord and Judith Stafford. Documenting Software Architectures: Views and Beyond (2nd Edition). Addison-Wesley Professional, 2 édition, 2010. (Cited on pages 10, 11, 12 and 27.)

[Corporation 2008] Oracle Corporation. *Measuring the Degree of Service Orientation in Proprietary SOA Systems.* In Business process management, service-oriented architecture, and web 2.0: Business transformation or train wreck? Oracle Corporation, 2008. (Cited on page 2.)

[Cox 1986] Brad J Cox. *Object-oriented programming: an evolutionary approach.* 1986. (Cited on page 8.)

[Czarnecki 2006] Krzysztof Czarnecki, Chang Hwan, Peter Kim and KT Kalleberg. *Feature models are views on ontologies.* In Software Product Line Conference, 2006 10th International, pages 41–51. IEEE, 2006. (Cited on page 14.)

[Dashofy 2002] Eric M. Dashofy, André van der Hoek and Richard N. Taylor. *An Infrastructure for the Rapid Development of XML-based Architecture Description Languages.* In Proceedings of the 24th International Conference on Software Engineering, ICSE '02, pages 266–276, New York, NY, USA, 2002. ACM. (Cited on pages 26, 31, 32 and 35.)

[Deiters 2011] Constanze Deiters and Andreas Rausch. *A Constructive Approach to Compositional Architecture Design.* In Ivica Crnkovic, Volker Gruhn and Matthias Book, editeurs, Software Architecture, volume 6903 of *Lecture Notes in Computer Science*, pages 75–82. Springer Berlin Heidelberg, 2011. (Cited on page 11.)

[Endrei 2004] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo and Tony Newling. Patterns: Service-Oriented Architecture and Web Services. IBM Redbooks, 1 édition, April 2004. (Cited on page 2.)

[Farrell 2007] Joel Farrell and Holger Lausen. *Semantic Annotations for WSDL and XML Schema*. https://www.w3.org/TR/sawsdl/, 2007. [Online; accessed 06-May-2016]. (Cited on page 23.)

[Fegler 2013] Brian Fegler. *Java Calculator Suite*. https://sourceforge.net/projects/bfegler/, 2013. [Online; accessed 04-May-2016]. (Cited on page 87.)

[Fiadeiro 2013] José Luiz Fiadeiro and Antónia Lopes. *A model for dynamic reconfiguration in service-oriented architectures*. Software & Systems Modeling, vol. 12, no. 2, pages 349–367, 2013. (Cited on page 1.)

[Figueiredo 2007] Eduardo Figueiredo. *Mobile Media*. http://homepages.dcc.ufmg.br/~figueiredo/spl/icse08/, 2007. [Online; accessed 04-May-2016]. (Cited on page 88.)

[Figueiredo 2008] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho and Francisco Dantas. *Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability*. In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, pages 261–270, New York, NY, USA, 2008. ACM. (Cited on page 90.)

[Footen 2012] J. Footen and J. Faust. The service-oriented media enterprise: Soa, bpm, and web services in professional media systems. Taylor & Francis, 2012. (Cited on page 12.)

[Fuhr 2013] Andreas Fuhr, Tassilo Horn, Volker Riediger and Andreas Winter. *Model-driven software migration into service-oriented architectures*. Computer Science - Research and Development, vol. 28, no. 1, pages 65–84, 2013. (Cited on pages 18, 19, 20 and 22.)

[Galster 2010] Matthias Galster. *Describing Variability in Service-oriented Software Product Lines*. In Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ECSA '10, pages 344–350, New York, NY, USA, 2010. ACM. (Cited on page 67.)

[Galster 2011] Matthias Galster, Paris Avgeriou, Danny Weyns and Tomi Männistö. *Variability in Software Architecture: Current Practice and Challenges*. SIGSOFT Softw. Eng. Notes, vol. 36, no. 5, pages 30–32, 2011. (Cited on pages 13 and 31.)

[Griffiths 2010] Nathan Griffiths and Kuo-Ming Chao. Agent-based service-oriented computing. Springer Publishing Company, Incorporated, 1st édition, 2010. (Cited on pages 2, 12 and 82.)

[Grobmeier 2015] Christian Grobmeier. *Log4j*. https://logging.apache.org/log4j/1.2/download.html, 2015. [Online; accessed 04-May-2016]. (Cited on page 88.)

[Hilliard 1999] Rich Hilliard. *Using the UML for Architectural Description*. In Robert France and Bernhard Rumpe, editeurs, «UML»99 — The Unified Modeling Language, volume 1723 of *Lecture Notes in Computer Science*, pages 32–48. Springer Berlin Heidelberg, 1999. (Cited on page 10.)

[Iribarne 2004] Luis Iribarne. *Web Components: A Comparison between Web Services and Software Components*. Revista Colombiana de Computación, vol. 5, no. 1, 2004. (Cited on page 24.)

[ISO 2011] *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Rapport technique, ISO/IEC 25010:2011, 2011. (Cited on page 39.)

[Jaggernauth 2015] Camille Jaggernauth, Bozena Kaminska and Douglas Gubbe. *Context-Aware Model for Dynamic Adaptability of Software for Embedded Systems*. International Journal of Computer (IJC), vol. 19, no. 1, pages 91–113, 2015. (Cited on page 1.)

[Jia 2007] Xiangyang Jia, Shi Ying, Honghua Cao and D. Xie. *A New Architecture Description Language for Service-Oriented Architec*. In Grid and Cooperative Computing, 2007. GCC 2007. Sixth International Conference on, pages 96–103, Aug 2007. (Cited on pages 11, 25, 26, 27, 29 and 30.)

[Joolia 2005] A. Joolia, T. Batista, G. Coulson and A.T.A. Gomes. *Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform*. In Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on, pages 131–140, 2005. (Cited on pages 26, 28 and 29.)

[Juric 2007] Matjaz B. Juric. *A Hands-on Introduction to BPEL*. http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html, 2007. [Online; accessed 06-May-2016]. (Cited on page 83.)

[Kang 1990] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak and A Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Rapport technique, DTIC Document, 1990. (Cited on page 13.)

[Karastoyanova 2003] Dimka Karastoyanova and Alejandro P Buchmann. *Components, Middleware and Web Services*. In ICWI, pages 967–970, 2003. (Cited on page 24.)

[Khadka 2011] R. Khadka, G. Reijnders, A. Saeidi, S. Jansen and J. Hage. *A method engineering based legacy to SOA migration method*. In Software Maintenance

(ICSM), 2011 27th IEEE International Conference on, pages 163–172, Sept 2011. (Cited on pages 18, 21 and 22.)

[Khadka 2013a] R. Khadka, A. Saeidi, S. Jansen and J. Hage. *A structured legacy to SOA migration process and its evaluation in practice.* In Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2013 IEEE 7th International Symposium on the, pages 2–11, Sept 2013. (Cited on pages 10, 18 and 20.)

[Khadka 2013b] Ravi Khadka, Amir Saeidi, Andrei Idu, Jurriaan Hage and Slinger Jansen. *Legacy to SOA evolution: A systematic literature review.* In A. D. Ionita, M. Litoiu, G. Lewis (Eds.) Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments: Challenges in Service Oriented Architecture and Cloud Computing Environments, page 40, 2013. (Cited on pages 10, 18, 21 and 23.)

[Koning 2009] Michiel Koning, Chang-ai Sun, Marco Sinnema and Paris Avgeriou. *VxBPEL: Supporting variability for Web services in BPEL.* Information and Software Technology, vol. 51, no. 2, pages 258–269, 2009. (Cited on pages 26 and 33.)

[Kruchten 1995] Philippe B Kruchten. *The 4+ 1 view model of architecture.* Software, IEEE, vol. 12, no. 6, pages 42–50, 1995. (Cited on page 11.)

[Kuba 2007] Martin Kuba and Ondrej Krajícek. *Literature search on SOA, Web Services, OGSA and WSRF.* Institute of Computer Science, Masaryk Univerity, 2007. (Cited on page 9.)

[Lausen 2005] Holger Lausen, Axel Polleres and Dumitru Roman. *Web Service Modeling Ontology (WSMO).* `https://www.w3.org/Submission/WSMO/`, 2005. [Online; accessed 06-May-2016]. (Cited on page 23.)

[Lee 2012] Jaejoon Lee, G. Kotonya and D. Robinson. *Engineering Service-Based Dynamic Software Product Lines.* Computer, vol. 45, no. 10, pages 49–55, Oct 2012. (Cited on page 13.)

[Lewis 2005] Grace Lewis, Edwin Morris, Liam O'Brien, Dennis Smith and Lutz Wrage. *SMART: The Service-Oriented Migration and Reuse Technique.* Rapport technique CMU/SEI-2005-TN-029, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. (Cited on pages 2, 18, 20, 21, 22 and 41.)

[Lindsey 1978] CH Lindsey and HJ Boom. *A modules and separate compilation facility for Algol 68:(preprint).* Stichting Mathematisch Centrum. Informatica, no. IW 105/78, pages 1–42, 1978. (Cited on page 8.)

[Luckham 1995] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan and Walter Mann. *Specification and analysis of system architecture using Rapide*. IEEE Transactions on Software Engineering, vol. 21, pages 336–355, 1995. (Cited on pages 26, 27 and 28.)

[Magee 1995] Jeff Magee, Naranker Dulay, Susan Eisenbach and Jeff Kramer. *Specifying Distributed Software Architectures*. In Proceedings of the 5th European Software Engineering Conference, pages 137–153, London, UK, UK, 1995. Springer-Verlag. (Cited on pages 25, 26, 28, 29 and 32.)

[Marchetto 2008] Alessandro Marchetto and Filippo Ricca. *Transforming a Java application in an equivalent Web-services based application: Toward a tool supported stepwise approach*. In Web Site Evolution, 2008. WSE 2008. 10th International Symposium on, pages 27–36. IEEE, 2008. (Cited on page 18.)

[Martin 2004] David Martin. *OWL-S: Semantic Markup for Web Services*. `https://www.w3.org/Submission/OWL-S/`, 2004. [Online; accessed 06-May-2016]. (Cited on page 23.)

[Matos 2009] Carlos Matos and Reiko Heckel. *Migrating legacy systems to service-oriented architectures*. Electronic Communications of the EASST, vol. 16, 2009. (Cited on pages 18, 20, 21 and 22.)

[Medvidovic 1996] Nenad Medvidovic. *ADLs and Dynamic Architecture Changes*. In Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops, ISAW '96, pages 24–27, New York, NY, USA, 1996. ACM. (Cited on pages 11, 25, 26, 28 and 29.)

[Medvidovic 2000] N. Medvidovic and R.N. Taylor. *A classification and comparison framework for software architecture description languages*. Software Engineering, IEEE Transactions on, vol. 26, no. 1, pages 70–93, Jan 2000. (Cited on page 11.)

[Minora 2012] Leonardo Minora, Jérémy Buisson, Flávio Oquendo and Thaís Vasconcelos Batista. *Issues of Architectural Description Languages for Handling Dynamic Reconfiguration*. CoRR, vol. abs/1205.4699, 2012. (Cited on page 28.)

[Nakagawa 2012] Elisa Yumi Nakagawa. *Reference Architectures and Variability: Current Status and Future Perspectives*. In Proceedings of the WICSA/ECSA 2012 Companion Volume, WICSA/ECSA '12, pages 159–162, New York, NY, USA, 2012. ACM. (Cited on page 31.)

[Nakamura 2009] Masahide Nakamura, Hiroshi Igaki, Takahiro Kimura and Kenichi Matsumoto. *Extracting service candidates from procedural programs based on process dependency analysis.* In Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific, pages 484–491. IEEE, Dec 2009. (Cited on pages 2, 12, 15, 18, 20, 21, 22, 40 and 43.)

[Nicholls 2009] Leon Nicholls and John Kohl. *Galleon TiVo Media Server.* `https://sourceforge.net/projects/galleon/`, 2009. [Online; accessed 04-May-2016]. (Cited on page 88.)

[Niiyama 2008] Craig Niiyama, Sam Chung, Donald Chinn and Sergio Davolos. *Service-oriented software reengineering methodology for composite services.* TCSS 702 Design Project in Computing and Software Systems, 2008. (Cited on page 11.)

[O'Brien 2005] L. O'Brien, D. Smith and G. Lewis. *Supporting Migration to Services using Software Architecture Reconstruction.* In Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on, pages 81–91, 2005. (Cited on pages 18, 20, 21 and 22.)

[Oquendo 2004] Flavio Oquendo. *π-ADL: an Architecture Description Language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures.* SIGSOFT Softw. Eng. Notes, pages 1–14, 2004. (Cited on pages 25, 26, 28, 29 and 30.)

[Oquendo 2008] Flavio Oquendo. *π-ADL for WS-Composition: A Service-Oriented Architecture Description Language for the Formal Development of Dynamic Web Service Compositions.* In Second Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2008), pages 1–14, Porto Alegre, Brazil, August 2008. (Cited on pages 10, 11, 26, 27, 29, 30 and 31.)

[Papazoglou 2007] Mike P. Papazoglou and Willem-Jan Heuvel. *Service Oriented Architectures: Approaches, Technologies and Research Issues.* The VLDB Journal, vol. 16, no. 3, pages 389–415, July 2007. (Cited on pages 2 and 12.)

[Papazoglou 2008] Michael Papazoglou. Web services: Principles and technology. Pearson Education. Pearson Prentice Hall, 2008. (Cited on pages 2, 12 and 41.)

[Patidar 2013] Mr Kailash Patidar, R Gupta and Gajendra Singh Chandel. *Coupling and cohesion measures in object oriented programming.* International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, no. 3, 2013. (Cited on page 44.)

[Pohl 2005] Klaus Pohl, Günter Böckle and Frank J van Der Linden. Software product line engineering: foundations, principles and techniques. Springer Science & Business Media, 2005. (Cited on page 13.)

[Salehie 2009] Mazeiar Salehie and Ladan Tahvildari. *Self-adaptive software: Landscape and research challenges.* ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 4, no. 2, page 14, 2009. (Cited on page 1.)

[Sneed 2006] Harry M Sneed. *Integrating legacy software into a service oriented architecture.* In Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, pages 11–pp. IEEE, 2006. (Cited on pages 7, 10, 18, 20, 21, 22 and 23.)

[Stehle 2008] Edward Stehle, Brian Piles, Jonathan Max-Sohmer and Kevin Lynch. *Migration of Legacy Software to Service Oriented Architecture.* Department of Computer Science Drexel University Philadelphia, PA, vol. 19104, pages 2–5, 2008. (Cited on pages 7, 10, 12 and 18.)

[Svahnberg 2005] Mikael Svahnberg, Jilles van Gurp and Jan Bosch. *A Taxonomy of Variability Realization Techniques: Research Articles.* Softw. Pract. Exper., vol. 35, no. 8, pages 705–754, July 2005. (Cited on page 13.)

[Szyperski 2002] Clemens Szyperski. Component software: Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2002. (Cited on page 9.)

[Tosic 2003] Vladimir Tosic, David Mennie and Bernard Pagurek. *Software Configuration Management Related to the Management of Distributed Systems and Service-Oriented Architectures.* In Bernhard Westfechtel and André van der Hoek, editeurs, Software Configuration Management, volume 2649 of *Lecture Notes in Computer Science*, pages 54–69. Springer Berlin Heidelberg, 2003. (Cited on page 24.)

[van Ommering 2000] R. van Ommering, F. van der Linden, J. Kramer and J. Magee. *The Koala component model for consumer electronics software.* Computer, vol. 33, no. 3, pages 78–85, Mar 2000. (Cited on pages 25, 26, 28, 30, 31, 32 and 35.)

[Vestal 1993] Steve Vestal. *A cursory overview and comparison of four architecture description languages.* Rapport technique, Citeseer, 1993. (Cited on page 11.)

[Voelter 2007] Markus Voelter and Iris Groher. *Product line implementation using aspect-oriented and model-driven software development.* In Software Product Line Conference, 2007. SPLC 2007. 11th International, pages 233–242. IEEE, 2007. (Cited on page 13.)

[Wahler 2015] M. Wahler, R. Eidenbenz, C. Franke and Y. A. Pignolet. *Migrating legacy control software to multi-core hardware.* In Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on, pages 458–466, Sept 2015. (Cited on page 3.)

[Weiss 1999] David M. Weiss and Chi Tau Robert Lai. Software product-line engineering: A family-based software development process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. (Cited on page 13.)

[Zhang 2004] Zhuopeng Zhang and H. Yang. *Incubating services in legacy systems for architectural migration.* In Software Engineering Conference, 2004. 11th Asia-Pacific, pages 196–203, Nov 2004. (Cited on pages 12, 18 and 23.)

[Zhang 2005] Zhuopeng Zhang, Ruimin Liu and Hongji Yang. *Service identification and packaging in service oriented reengineering.* In In Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering (SEKE, pages 241–249, 2005. (Cited on pages 2, 21, 22, 23 and 40.)

[Zhang 2006] Zhuopeng Zhang, H. Yang and W.C. Chu. *Extracting Reusable Object-Oriented Legacy Code Segments with Combined Formal Concept Analysis and Slicing Techniques for Service Integration.* In Quality Software, 2006. QSIC 2006. Sixth International Conference on, pages 385–392, Oct 2006. (Cited on page 23.)

[Zhu 2011] Jiayi Zhu, Xin Peng, Stan Jarzabek, Zhenchang Xing, Yinxing Xue and Wenyun Zhao. *Improving Product Line Architecture Design and Customization by Raising the Level of Variability Modeling.* In Klaus Schmid, editeur, Top Productivity through Software Reuse, volume 6727 of *Lecture Notes in Computer Science*, pages 151–166. Springer Berlin Heidelberg, 2011. (Cited on pages 31 and 33.)