République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur de la Recherche Scientifique
Université Mohamed Khider -Biskra-

N° d'ordre :........

Série :......

Faculté des Sciences Exactes et des Sciences de la Nature et de la vie
Département d'Informatique

THÈSE

présentée pour
obtenir le diplôme de doctorat en sciences

SPÉCIALITÉ : **INFORMATIQUE**

Pattern-based Approach for Quality Integration in Service-based Systems

(Une approche à base de patrons pour l'intégration de la qualité dans les systèmes à base de services)

par

Tarek ZERNADJI

Soutenue le: mars 2016 , Devant le jury composé de :

Version of April 3, 2016

Contents

Contents **iii**

Acknowledgement **vii**

Abstract **ix**

Résumé **xi**

1 Introduction **1**

1.1 Context 1
1.2 Problem statement 3
1.3 Contributions 5
1.4 Dissertation plan 7

2 State of the art **9**

2.1 Background 9
2.1.1 Service Oriented architecture (SOA) 10
2.1.2 Web services 11
2.1.3 Web Services composition 15
2.1.4 Web Services Business Process Execution Language (WS-BPEL) . 17
2.2 Literature review 23
2.2.1 Architecture decisions documentation 23
2.2.2 Software quality documentation 33
2.2.3 Quality achievement in service-based systems 45
2.2.4 Assistance to software evolution and impact analysis 50
2.2.5 Discussion 55
2.2.6 Summary 57

3 Pattern-based documentation model of architecture decisions 59

3.1 General Model . . . 59
3.2 Links between Architecture Design Decisions and Quality Attributes . . . 61
3.3 WS-BScript: Web Service BPEL Scripting language . . . 65
3.4 SOA Patterns Architecture Constraint Specification . . . 70
3.5 Summary . . . 75

4 SAQIM: Service-Oriented Architecture Quality Integration Method 77

4.1 The Method at a Glance . . . 77
4.2 Quality attribute integration intent specification . . . 80
4.3 Pattern Selection . . . 82
4.4 Pattern Application . . . 83
4.5 Quality Impact Analysis . . . 86
4.6 New Patterns Definition . . . 87
4.7 Pattern Cancellation . . . 87
4.8 Documentation of the New Architecture . . . 89
4.9 Summary . . . 90

5 Quality-oriented impact analysis process 93

5.1 A Micro-Process of Architecture Evolution . . . 93
5.2 Quality-Oriented Architecture Change Assistance . . . 95
5.2.1 Quality-Oriented Assistance Service . . . 101
5.2.2 Weighted Sum Model for Patterns ranking . . . 102
5.3 Summary . . . 107

6 Evaluation 109

6.1 SAQIM in Practice . . . 109
6.2 Experiment Process . . . 119
6.2.1 Methodology . . . 120
6.2.2 Data Collection . . . 120
6.2.3 Simulation . . . 124
6.2.4 Discussion . . . 127
6.2.5 Threats to validity . . . 129
6.3 Summary . . . 130

7 Conclusion and Future Work **131**

7.1 Conclusion 131

7.2 Future Work 134

List of Figures **137**

List of Tables **139**

List of Listings **140**

Bibliography **141**

Acknowledgement

First and foremost, praises and thanks to Allah.

I would like to thank my supervisors Dr. Foudil Cherif and Dr. Chouki Tibermacine who supported me throughout the writing of this dissertation, for their listening and relevant advices.

I also thank all the members of the jury Prof. Djeddi Noureddine, Dr. Salah Sadou, Dr. Abdelmalik Bachir, and Prof. Mohamed Benmohammed for the time they spend to review this work, I am grateful for the attention they paid to my work.

I also want to thank my family and friends for their unconditional support and their continued presence.

Finally I thank all those who helped me in some way for the realization of this work.

Abstract

Building distributed software by orchestrating existing Web services is a new paradigm, which has been proposed as a possible implementation for the service-oriented architecture (SOA) specification. The emergence of such technology and languages is recent. So, the engineering of these service-oriented applications is not yet mature and raises many challenging questions. Among these questions, we can mention the crucial issue of how to satisfy quality requirements in this kind of engineering processes.

In this thesis, we addressed the aforementioned problem by various contributions. We proposed a model of architectural design decisions documentation and two languages. This model explicits formally the links between SOA patterns as design decisions and qualitiy attributes. The first language is a scripting language called "WS-BScript". It is a lightweight DSL for specifying primitive changes making possible the reconfiguration of Web services orchestrations. The second, is a constraint language based on OCL coupled with BPEL (*Business Process Execution Language*) language meta-model. It allows to specify predicates that check whether an instance of a pattern exists in an architecture or not and therefore the quality it implements. We also proposed a method named "SAQIM" (Service-oriented Architecture Quality Integration Method) which aims to provide software architects of Web service orchestrations an on-demand assistance for the integration of quality requirements in their artifacts. This method is based on a SOA patterns catalog already documented using the model of the first contribution. It also makes use of our third contribution, a quality impact analysis process that support the reasoning about the quality consequences of an applied SOA pattern. An experimentation on using the proposed processes has been realized. This experimentation is considered as our last contribution in this thesis.

Keywords: Service Oriented Architecture (SOA), Design decision, Quality attribute, SOA pattern, BPEL.

Résumé

Construire des logiciels distribués en orchestrant des services Web existants est un nouveau paradigme, qui a été proposée comme une mise en œuvre possible de la spécification de l'architecture orientée services (SOA). L'émergence de telles technologie et langages est récente. Ainsi, l'ingénierie de ces applications orientées service n'est pas encore mature et soulève de nombreuses questions difficiles. Parmi ces questions, c'est de savoir comment satisfaire les exigences de qualité dans ce genre de processus d'ingénierie. Dans cette thèse, nous avons abordé le problème susmentionné par diverses contributions. Nous proposons donc un modèle de documentation de décisions architecturales ainsi que deux langages. Ce modèle explicite formellement les liens entre des patrons SOA comme étant des décisions de conception et les attributs qualités. Le premier langage est un langage de script appelé "WS-BScript". C'est un DSL léger qui permet de spécifier des changements primitifs rendant possible la reconfiguration des orchestrations de services Web. Le deuxième, est un langage de contrainte basé sur OCL couplé avec le méta-modèle de BPEL. Il permet de spécifier des prédicats qui vérifient si une instance d'un patron existe dans une architecture ou non et donc la qualité qu'il implémente. Nous proposons aussi une méthode nommée "SAQIM" qui vise à fournir aux architectes logiciels des orchestrations de services Web une assistance à la demande pour l'intégration des exigences de qualité dans leurs artefacts. Cette méthode s'appuie sur un catalogue de patrons SOA documenté en utilisant le modèle de la première contribution. Elle utilise notre troisième contribution, un processus d'analyse d'impact sur la qualité qui appuie le raisonnement sur les conséquences de l'application d'un patron SOA sur les qualités. Une expérimentation sur l'utilisation des processus proposés a été réalisé et est considérée comme notre dernière contribution.

Mots clés: SOA, Décision architecturale, Attribut Qualité, patron SOA, BPEL.

CHAPTER 1

Introduction

1.1 Context

In the last two decades, (Restful or SOAP-based) Web services have confirmed their status of one of the leading technologies for implementing components of service-oriented software architectures for desktop, Web and even mobile applications. The growing need for choosing such technology is related to: i) the integrability and portability (independence from programming languages, middleware or operating systems) provided by the published services, ii) the ease of use and efficiency of HTTP as a communication protocol with these services, iii) the security brought by the SSL/TLS layer included in HTTPS, among many other “ilities”.

When modeling applications that involve the invocation of Web services, we can build two kinds of compositions of Web services: orchestrations or choreographies. In choreographies, Web services are considered as peers that collaborate in order to implement the application’s business logic. One possible language that can be used for modeling choreographies is the OMG’s standard BPMN (Business Process Model and Notation [Groupe, 2011]). Orchestrations include a central workflow process that implements the main business logic of the modeled application, and which invokes operations of “partner” Web services. One of leading languages used for modeling (and even executing) orchestrations is the OASIS standard WS-BPEL[1] or BPEL (Business Process Execution Language [BPL, 2007]).

Building distributed software by orchestrating existing Web services is a new paradigm, which has been proposed as a possible implementation for the service-oriented architecture specification. It has been greatly influenced by the well-known business process engineering field, where processes can be designed as collaborations between a set of services published by some providers. New business logic can thus be implemented, as an extension of existing Web services, through these orchestrations. This helps development teams in capitalizing resources held by the providers of these services. Indeed, Web service providers, which hold some precious resources (like large databases of products to retail of Amazon, or weather forecast data of Meteo France), offer third party developers the opportunity (for free or not) to build new applications by extending their public services, and thus capitalize on these resources.

Nonetheless, these service-oriented software architectures, like any other software artifact, are subject to changes during their lifecycle, and thus can be affected by the consequences of an evolution phenomenon [Lehman et Ramil, 2002]. This evolution is a natural consequence of responses to the changing requirements, imposed by users as well as the environment with which the software system interacts or in which it runs.

A key aspect of a software evolution is the evolution of its architecture. The concept of software architecture - a high level abstraction of the system structure and behavior- is recognized as an effective means to deal with complex software systems design issues. A software architecture is one of the first artifacts of the design process. It represents the first decisions for designing a software system, and thus allows to analyze and evaluate the system early in the development process [Bass *et al.*, 2003]. A recent development in software architecture research is the notion of Architectural Knowledge (AK) [Kruchten *et al.*, 2006; de Boer *et al.*, 2007; de Boer et Farenhorst, 2008; Jansen, 2008]. AK encompasses all the knowledge acquired or formulated involved with software architectures. One of the most important form of AK is the notion of architectural design decisions (ADs) [Jansen et Bosch, 2005], and one of the most common design decisions at the architectural development process is the choice of a design pattern. Design patterns are sets of predefined design decisions with known functionality and behavior [Gamma *et al.*, 1995].

[1]In this thesis we will use the terms WS-BPEL and BPEL interchangeably

Evolution can target two aspects: functional and non-functional (qualitative). The first concerns the addition, removal or modification of functionalities, while the second focuses on the qualities that the software must reflect in its architecture. It is acknowledged that architectural design decisions are driven by the quality attributes required in the specification documents [Mylopoulos *et al.*, 1992 ; Bass *et al.*, 2003]. Indeed, it is not the software expected functionalities that mainly determine its architecture, but rather the way in which its functionalities will be provided that allows shaping and developing the architecture. So, when improving some software qualities such as maintainability, performance or portability by introducing new architectural design decisions, or when trying simply to add, remove or edit a functionality we may unintentionally affect other previously made decisions, and therefore certain qualities can be affected. It is argued that quality can be weakened after successive changes (Lehman's 7th law of software evolution [Lehman et Ramil, 2002]). These problems are often raised during the maintenance phase. This is mainly due to: i) the lack of information on the ADs that led to the software architecture, and an explicit definition of the links between the non-functional characteristics and ADs implementing them, and ii) the lack of tool support to supervise architecture changes. Architecture evolution is about making new design decisions or removing obsolete ones to satisfy changing requirements. The challenge is to do this in harmony with the existing design decisions [Jansen et Bosch, 2005].

This thesis deals with some problems encountered by architects during the design and/or the evolution of software architectures, more specifically those related to quality aspects in Web service oriented architectures. These problems are addressed in the following section.

1.2 Problem statement

The emergence of service-based systems related technologies and languages is recent. So, the engineering of these service-oriented applications is not yet mature and raises many challenging questions. Among these questions, we can mention the crucial issue of how to satisfy quality requirements in this kind of engineering processes.

In this thesis, we tackled the problem of integrating non-functional requirements

(*NFRs*) [2] in web service orchestrations. Addressing the problem of satisfying NFRs at the architectural design level of web service orchestrations, involves managing the related ADs that shape the service orchestration.

Dealing with the aforementioned problem raises three main underlying ones. The first is known as architectural knowledge vaporization about ADs. In this phenomenon most of ADs made during the architectural development process are lost and consequently the reasons (rationale) they led to them are also lost. This is mainly due to the fact that, ADs are often not explicitly documented or still as intentions in the mind of architects. Indeed, during the architectural development process architects tend to avoid documenting ADs. This may come from many reasons, we can mention among others, the fact that either they consider it as a difficult and/or time consuming task, or they do not perceive the real benefit from the dedicated effort. Thus, in the absence of an explicit documentation describing the choices that have been made by developers (ADs) and which shape the architecture, design conflicts could appear and eventually the loss of the system's quality properties.

Therefore, the first research question we addressed in this thesis is:

- *RQ1: How can we document architectural decisions in order to reduce AK vaporization?*

The second raised problem comes from the lack of assistance methods to architect in the existing design methods from the NFRs elicitation to their implementation through ADs application. Indeed, an architect's proposed solution (AD), like the choice of a design pattern for a quality attribute may present more than one competing design alternatives. In such a situation she/he may not be able to choose an alternative. Even when the architect knows what alternative to apply, she/he may not know how to apply it.

Thus, the second research question we addressed in this thesis is:

- *RQ2: How to assist architects in finding and applying the architecture design decisions that answer an integration of a quality attribute in their software architecture?*

[2] non-functional requirements and quality requirements are used interchangeably in this thesis

The third problem arises when a change is made to the software architecture. A change occurs by applying one or several design decisions which may affect qualities of the software architecture. Each decision may have different impact on the existing qualities. Existing methods still lack offering sufficient practices and guidance to architect that help him to better control change impact on quality requirements of the software architecture.

Hence, the third research question we addressed in this thesis is:

- *RQ3: How to assist architects in analyzing the impact of the integration of quality requirements on the overall software architecture qualities?*

Therefore, it is very useful to be able to control and master the integration of quality requirements in web service orchestrations by offering to the architects an efficient mean during its activity.

1.3 Contributions

Dealing with the aforementioned problems results in this thesis with three main contributions: i) an architecture design decision documentation model, ii) a service-oriented architecture quality integration method called *SAQIM* and iii) a quality-oriented impact analysis process.

- **Architecture design decision documentation model**: It defines in a formal way the links between architectural design decisions and quality attributes implemented by these decisions. It aims hence, at representing architectural design decisions as first class entities in a software architecture. A special kind of architectural design decisions is used in our model which are design patterns that target SOA architectures. The model includes the documentation of different facets of a pattern: its name, description, the guaranteed quality attribute, pattern instantiation scripts, and the constraints allowing the verification of its presence or absence in the architecture. Therefore, two languages are proposed to specify catalogs of SOA patterns. The first one that we propose in this thesis, is a scripting language that allows to create instances for a pattern in SOA architectures concretely defined with BPEL language. It is a DSL (*Domain Specific Language*) with a voluntarily simplified set of primitives to simplify the documentation of SOA patterns. The second language, is a constraint language based on OCL *Object Constraint Language*) coupled with BPEL language meta-model. This allows to specify predicates which verifies if a pattern instance exists in an architecture or not.

- **Service-oriented Architecture Quality Integration Method (SAQIM)**: It is a method which aims at providing to software architects of Web service orchestrations an on-demand assistance in integrating quality requirements in their artifacts. This method has been designed as a multi-step process and makes use of a previously documented SOA patterns catalog. It introduces a template for enabling architects to describe quality integration “intents”. It then analyzes these intents and helps the architect in satisfying the targeted quality attribute by suggesting some service-oriented patterns. After that, the method that we propose simulates the application of different alternative patterns that satisfy the targeted quality requirement. It helps the developer to select among several patterns the one that satisfies the best its preferences. It helps him also to instantiate the selected pattern on its architecture by executing its script. Thanks to this process, the developer has also the possibility to cancel the pattern instantiation using “cancellation” scripts obtained automatically from the first script.

- **Quality-oriented impact analysis process**: It is a second process which completes the first one. It aims to notify the developer about the impact of a software architecture modification on the other previously integrated qualities in the software architecture. A modification could be the instantiation of a pattern in the software architecture, hence the complementarity of this process with the previous one. This impact analysis is mainly based on OCL constraints evaluation of the already integrated patterns. This process is based on an architecture documentation that we proposed in the first contribution. Using some fine grained information defined in the documentation model, the process provides to the developer a detailed notification report that allows him to control the changes made to its architecture.

The first and the second process as well as the language interpreters have been implemented in prototypical tools and experimented on real-world BPEL orchestrations.

A number of simulations on using the proposed processes have been realized. The obtained results showed the benefit of using the proposed quality integration assistance. This experimentation is considered as our last contribution in this thesis.

1.4 Dissertation plan

Chapter 2 presents a State of the art that synthesize and analyze a set of research works in the literature dealing with the studied problem in this thesis. More particularly, works on architectural design decisions documentation, software quality documentation, as well as works on assistance to software evolution have been covered. Chapter 3 introduces the first contribution. The different concepts of the documentation model are presented and detailed namely, SOA patterns, quality attributes and their relationships, and links between the two concepts as well. Chapter 4 covers the presentation of SAQIM the second contribution. The different steps of the method are illustrated by detailed explanations and concrete examples. Chapter 5 presents through a detailed algorithm, the process used to analyze the impact of a quality integration on the other quality attributes of a web service orchestration. Chapter 6 details the experimental process to evaluate the proposed methods and the model they use. We show the setup, the evaluating process, the analysis and final results. Finally, in Chapter 7 we conclude the thesis with a summary of contributions and open perspectives.

Related publications

- Tarek Zernadji, Chouki Tibermacine, Foudil Cherif and Amina Zouioueche. Integrating Quality Requirements in Engineering Web Service Orchestrations. Accepted in the Journal of Systems and Software, November 2015. Elsevier.
- Tarek Zernadji, Chouki Tibermacine, et Foudil Cherif. Quality-driven design of web service business processes. In Proc. of WETICE/AROSA'14, Parme, Italie, Juin 2014. IEEE CS.
- Tarek Zernadji, Chouki Tibermacine, et Foudil Cherif. Processing the evolution of quality requirements of web service orchestrations: a pattern-based approach. In Proc. of WICSA'14, Sydney, Australia, April 2014. IEEE CS.
- Tarek Zernadji, Tibermacine Chouki, Fleurquin Régis, and Sadou Salah. Assistance à l'évolution du logiciel dirigée par la qualité. A book chapter In Évolu-

tion et maintenance des systèmes logiciels, edited by Abdelhak-Djamel Seriai and published by Hermes Sciences-Lavoisier 2014. ISBN 9782746245549, Paris, France.

- Chouki Tibermacine et Tarek Zernadji. Supervising the evolution of web service orchestrations using quality requirements. In Proc. of ECSA'11, pages 1–16, Essen, Germany, September 2011. Springer-Verlag.

CHAPTER 2

State of the art

In this chapter we present an overview of the state of the art. We begin with a brief overview on service-oriented software architecture (SOA) in general then, we present Web services orchestration as a kind of SOA. We emphasized on specific orchestrations which are implemented with WS-BPEL language. In the second part, we discuss works in four categories that are closely related to our work. In section 2.2.1 we present works that deal with architectural design decisions documentation. We show in section 2.2.2 some of the approaches on software quality documentation. Specific works dealing with quality requirements in the context of service-based systems are presented in section 2.2.3. Finally, we give a brief overview on works providing assistance to software evolution in section 2.2.4.

2.1 Background

Before discussing the literature about the studied problem in this thesis, we would like to clarify some background information on the service oriented architectures (SOA) paradigm (Section 2.1.1) and one of its implementation technologies the "Web services" (Section 2.1.2), as well as the notion of web service composition (Section 2.1.3).

At last, a detailed presentation of WS-BPEL language is given in section 2.1.4. We discuss these background knowledge in the sub-sections below.

2.1.1 Service Oriented architecture (SOA)

There are a lot of definitions of the term SOA "Service Oriented Architecture" but no exact definition has been defined to this day. We have listed some of them below.

The OASIS (Organization for the Advancement of Structured Information Standards) SOA reference model defines SOA as follows:"*Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains*" [MacKenzie *et al.*, 2006]. This definition presents SOA as a way of organizing different capabilities offered by different owners in the world of distributed computing.

The World Wide Web Consortium (W3C) defined SOA as: "*a set of components which can be invoked and whose interface descriptions can be published and discovered* "[1]. This definition gives a general context for implementing service-oriented architectures which is not limited to a specific kind of functionality, but imposes that the latter should be published and discovered.

Another definition was given by Thomas erl [Erl, 2009]: "*Service-oriented architecture represents an architectural model that aims to enhance the agility and cost-effectiveness of an enterprise while reducing the burden of IT on the overall organization. It accomplishes this by positioning services as the primary means through which solution logic is represented"*. Erl stresses the improved agility given by SOA which is represented as an architectural model. This definition uses the term of "services" rather than capabilities or components, as a means to implement SOA.

The open group gives the following definition: "*Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services*"[2]. This definition also emphasizes the use of services as units to build service-based systems.

All the definitions agree that SOA is a paradigm, a way of thinking that provides a

[1]http://www.w3.org/TR/ws-gloss/#component
[2]http://www.opengroup.org/soa/source-book/soa/soa.htm

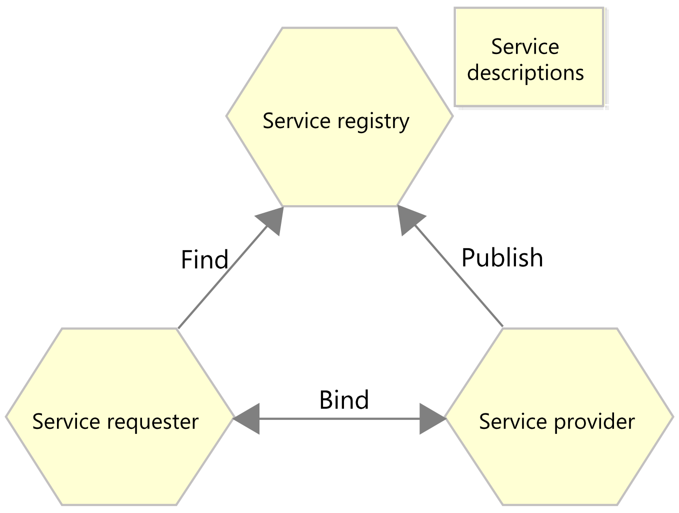


Figure 2.1 : SOA infrastructure

general architecture for building complex distributed systems based on services. Services are interface-based computing facilities which are described in a uniform, technology neutral manner. They allow loosely-coupled, message-based interaction and are transparent concerning their location [Lenhard, 2011]. Figure 2.1 shows the three types of partners required to build a SOA-based application: Service registry, Service Providers and Service Requesters. The course of action is as follows:

1. The service provider publishes the service description to the service registry
2. The service requester finds the desired service by querying the service registry
3. The service requester binds to the service and retrieves the desired function.

Not all service-oriented architecture is based on Web services but, the real momentum for SOA was created by Web services. Web services technology is the most promising choice to implement service oriented architecture and its strategic objectives [Sheng *et al.*, 2014].

2.1.2 Web services

A variety of definitions about Web services are given by different industry leaders, research groups, and Web service consortia. Web Services and surrounding technologies are promoted by the World Wide Web Consortium (W3C). In the following we give the definition which is adopted by the W3C: "*A Web service is 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*" [World Wide Web Consortium, 2004]. The aforementioned definition gives a high-level description of the major supporting technologies of Web services. Interoperation among machines is the major design goal of Web services. As the supporting standards, WSDL (Web Services Description Language) enables XML service description of Web services and SOAP (Simple object access protocol) defines a communication protocol for Web services [Yu *et al.*, 2008].

The Web services framework is divided into three areas: communication protocols, service descriptions, and service discovery, and specifications are being developed for each [Wang *et al.*, 2004].

- The simple object access protocol (SOAP) that enables communications among Web services;
- The Web Services Description Language (WSDL) that provides a formal, computer-readable description of Web services; and
- The Universal Description, Discovery and Integration (UDDI) directory that is a registry of Web services descriptions.

Simple object access protocol (SOAP)

SOAP [World Wide Web Consortium, 2004] is a Web service messaging standard that enables communication among Web services. It provides a lightweight messaging framework for exchanging XML-based messages. SOAP is independent of languages and platforms. While SOAP Version 1.2 doesn't define "SOAP" as an acronym anymore, there are two expansions of the term that reflect these different ways in which the technology can be interpreted:

- Service Oriented Architecture Protocol: In the general case, a SOAP message represents the information needed to invoke a service or reflect the results of a service invocation, and contains the information specified in the service interface definition.

- Simple Object Access Protocol: When using the optional SOAP RPC Representation, a SOAP message represents a method invocation on a remote object, and the serialization of in the argument list of that method that must be moved from the local environment to the remote environment.

At its core, a SOAP message has a very simple structure: an XML element with two children elements, one containing the header and the other the body. The header contents and body elements are also represented in XML. SOAP messages can be transported over HTTP for the runtime invocation, which helps achieve the synchronous communication. The HTTP protocol plays the bridging role for interactions between computer systems [Wang *et al.*, 2004].

Web Services Description Language (WSDL)

WSDL [World Wide Web Consortium, 2004] is a language for describing Web services. WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

A WSDL document (see the Hello service example in listing 2.1) describes programming interfaces and accessing formats of a Web service. It makes a clear separation of the abstract and concrete descriptions of a Web service [Yu *et al.*, 2008]. At the abstract level, the WSDL description includes three basic elements: *Types*, *Message*, and *PortType*. The types element encloses data type definitions that are relevant for the exchanged messages. Message represents an abstract definition of the transferred data. A message consists of one or more logical parts. WSDL message specifies what type of data the message must contain when an operation is invoked. PortType is a set of abstract operations provided by an endpoint of a Web service. At the concrete level the WSDL description provides information of binding a concrete service endpoint. It specifies three elements: *Binding*, *port*, and *service*. The binding specifies the communication protocols, and the data format of the operations and messages. The port describes a single address for binding a service endpoint. The service defines a collection of ports.

```
<definitions name="HelloService"
targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<message name="SayHelloRequest">
<part name="firstName" type="xsd:string"/>
</message>

<message name="SayHelloResponse">
<part name="greeting" type="xsd:string"/>
</message>

<portType name="Hello_PortType">
<operation name="sayHello">
<input message="tns:SayHelloRequest"/>
<output message="tns:SayHelloResponse"/>
</operation>
</portType>

<binding name="Hello_Binding" type="tns:Hello_PortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="sayHello">
    <soap:operation soapAction="sayHello"/>
    <input>
      <soap:body
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="urn:examples:helloservice" use="encoded"/>
    </input>
    <output>
    <soap:body
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:examples:helloservice" use="encoded"/>
    </output>
   </operation>
</binding>

<service name="Hello_Service">
  <documentation> WSDL File for HelloService</documentation>
  <port binding="tns:Hello_Binding" name="Hello_Port">
    <soap:address location="http://www.examples.com/SayHello/" />
  </port>
</service>
</definitions>
```

LISTING 2.1 : Hello service WSDL example

Universal Description, Discovery and Integration (UDDI)

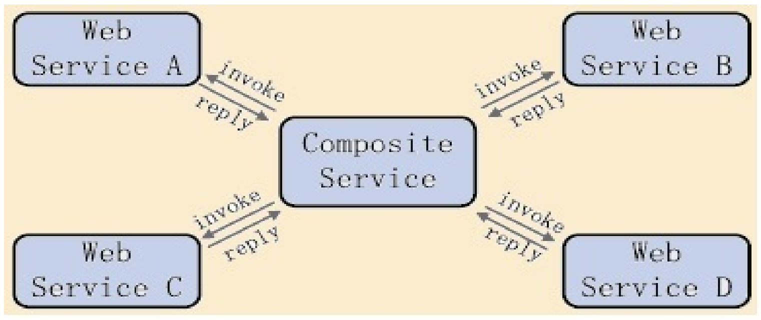
Universal Description, Discovery and Integration, or UDDI [OASIS, 2002], is the name of a group of web-based registries that expose information about a business or other entity and its technical interfaces (or API's). These registries are run by multiple Operator Sites, and can be used by anyone who wants to make information available about one or more businesses or entities, as well as anyone that wants to find that information. UDDI integrates Web service description and discovery to help service requesters locate their desirable services. It provides a set of search facilities for finding businesses, and their services. Services can be searched by specifying business name, service name or service category [Akkiraju *et al.*, 2003].

2.1.3 Web Services composition

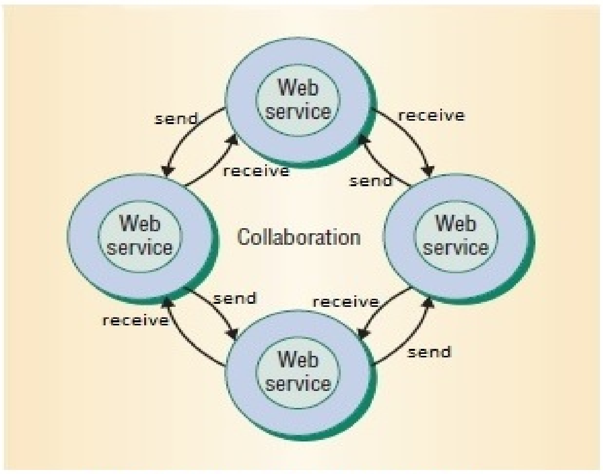
In service-oriented computing (SOC), developers use services as fundamental elements in their application development processes [Milanovic et Malek, 2004]. Developers and users can then solve complex problems by combining available basic services and ordering them to best suit their problem requirements. One key challenge for SOA and Web services technology is Web services composition [Sheng *et al.*, 2014]. Web service composition accelerates rapid application development, service reuse, and complex service consummation.

The terms orchestration and choreography describe two aspects of creating business processes from composite Web services. The former (figure 2.2 (a)) always represent control from one party's perspective. This differs from choreography (figure 2.2 (b)), which is more collaborative and allows each involved party to describe its part in the interaction [Peltz, 2003]. In this context, many languages and standards have been proposed for Web services composition over the years such as BPML (Business Process Modeling Language)[3], WS-BPEL [BPL, 2007], WSCL (Web Services Conversation Language) [WSC, 2002], WSCI (Web Services Choreography Interface) [WCI, 2002], WS-CDL (Web Services Choreography Description Language) [WCL, 2004], and BPMN (Business Process Model and Notation) [Groupe, 2011].

[3]http://www.ebpml.org/bpml.htm



(a) Service Orchestration



(b) Service Choreography

Figure 2.2 : Service orchestration and service choreography

Web services choreography

Choreography represents a global description of the observable behavior of each of the services participating in the interaction, which is defined by public exchange of messages, rules of interaction and agreements between two or more business process endpoints [Sheng *et al.*, 2014]. Choreography tracks the message sequences among multiple parties and sources (typically the public message exchanges that occur between Web services) rather than a specific business process that a single party executes [Peltz, 2003]. All Web services which take part in the choreography must be conscious of the business process, operations to execute, messages to exchange as well as the timing of message exchanges. The choreography mechanism is supported by the standard WS-CDL (Web Services Choreography Description Language).

Web services orchestration

Service orchestration represents a single executable business process that coordinates the interaction among the different services, by describing a flow from the perspective and under control of a single endpoint [Sheng *et al.*, 2014]. Orchestration can therefore be considered as a construct between an automated process and the individual services that enact the steps in the process. The interactions occur at the message level. They include business logic and task execution order, and they can span applications and organizations to define a long-lived, transactional, multi-step process model [Peltz, 2003]. Orchestration includes the management of the transactions between the individual services, including any necessary error handling, as well as describing the overall process. WS-BPEL (or BPEL in short), is the standard for Web services orchestration which is largely supported by the industry. WS-BPEL is one of leading languages for modeling and even executing Web service orchestrations.

In this thesis we deal with a special kind of SOA architectures which are Web service orchestrations concretely defined with WS-BPEL language.

2.1.4 Web Services Business Process Execution Language (WS-BPEL)

The Web Services Business Process Execution Language, commonly abbreviated as (*BPEL*), or (*WS-BPEL*) is an XML dialect for describing business processes based on Web services. BPEL is an orchestration language that was first conceived in July, 2002 with the release of the WS-BPEL 1.0 specification as a combination of XLANG [MIC, 2001] language by Microsoft and the Web Services Flow Language (WSFL) by IBM. Today, it is promoted by the Organization for the Advancement of Structured Information Standards (OASIS) and since April 2007 it is available in version 2.0 [BPL, 2007].

```
<process name=" SampleProcess ">
  <import  namespace="http://.../MyRole"
   location="MyRole.wsdl" importType="http://schemas.xmlsoap.org/wsdl/"/>
  <partnerLinks>
   <partnerLink name="MyRolePartnerLink" partnerLinkType="MyRolePartnerLinkType" myRole="
        myRole"/>
  </partnerLinks>
  <variables>
   <variable name="InputParameter" messageType="InputMessage"/>
  </variables>
```

```
10   <correlationSets>
11    <correlationSet name="CorrelationSet" properties="PropertyFromWSDL"/>
12   </correlationSets>
13   <sequence name="MainProcessFlow">
14    <receive name="StartProcess" createInstance="yes" variable="InputParameter"
        partnerLink="MyRolePartnerLink" operation="OperationFromWSDL" />
15         <!-- More basic and structured activities -->
16   </sequence>
17 </process>
```

LISTING 2.2 : A general structure of a BPEL process

BPEL allows to specify the behavior of a business process based on interaction between the process itself and its partners. The interaction with each partner occurs through a "`PartnerLink`" element which represent an external web service interface. This latter, is represented by a WSDL document that describe the offered service through a set of operations and handled messages. A BPEL process coordinates (orchestrates) the interactions between partners and specifies the necessary logic to achieve a business goal. BPEL provides two ways to describe processes: abstract processes and executable processes. Executable processes are fully specified processes that can be deployed and executed by an engine. Abstract processes are not executable and they are not completely specified. Abstract processes hide the implementation details of a process and serve mainly a descriptive role [BPL, 2007].

An XML representation is associated with the description of a BPEL process as well as a graphical (to design BPEL processes) one which are provided by most of the graphical editors[4,5].

A process is described in a BPEL process file and is purely Web Services-based [Lenhard, 2011]. The structure of a BPEL file is comprised of several elements as shown in listing 2.2. At least one WSDL file has to be imported into the process to serve as an interface for it (Line 3). The entire process is contained in a "`process`" element. It represents the root element (in the BPEL file) inside which the complete description of the process elements is embodied [BPL, 2007]:

`PartnerLinks`: PartnerLinks (Line 4) define the relationship between the process and external Web services. Each `partnerLink` (Line 5) relates to one partner who participates in the process. It references a `partnerLinkType` which is to be found in the

[4]https://eclipse.org/bpel/

[5]https://soa.netbeans.org/

WSDL file that describes the interface of the partner. Within a "partnerLink", the role of the business process itself is indicated by the attribute myRole and the role of the partner is indicated by the attribute partnerRole. At least one WSDL interface needs to be in place and one "partnerLink" needs to be defined for an executable process. This minimum "partnerLink" describes the role of the process itself (has the myRole attribute set), so that it can be invoked externally. If the BPEL process interacts with other Web services they should be added as a "partnerLink" and their WSDL description must be imported into the process.

`variables:` Variables (Line 7) provide the means for holding messages that constitute a part of the state of a business process. Messages might be those that have been received from partners or are to be sent to partners. A process may contain a set of variables. BPEL use three kinds of variables: Either their type is a "messageType" read from a WSDL file, or an XML Schema or Schema element type defined by an imported XSD (XML Schema Definition) file. During the execution of the process, variables can be referenced by several activities which may assign or read data to or from them.

`correlationSets:` BPEL supports message correlation using correlationSets (Line 10) which are used to route messages to the right process instance. correlationSets is made up of one or more WSDL properties that have a propertyAlias defined in the imported WSDL files. Correlation can be used on every messaging activity ("receive", "reply", "onMessage", "onEvent", and "invoke") [BPL, 2007]. Properties reference messageTypes that are XML simple types, defined in the types part of the Web service. The values of these types in incoming messages can then be used by the engine to direct the message to the matching process instance.

Finally, each BPEL process has one main activity that represent the main control flow (Line 13). BPEL define a set of activities which are divided into 2 classes: basic and structured. Basic activities are those which describe elemental steps of the process behavior. Structured activities encode control-flow logic, and therefore can contain other basic and/or structured activities recursively [BPL, 2007].

Figure 2.3 shows an excerpt from the BPEL language meta-model (upper part of the figure) and WSDL (lower part).

In the following subsections we are going to present the majority of the main BPEL language constructs and not all of them because of the large size of its specification.

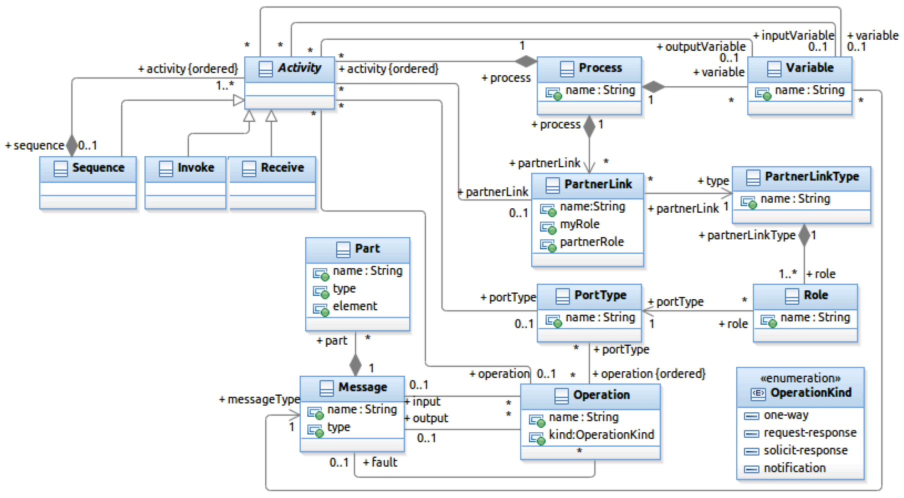


Figure 2.3 : An excerpt of the BPEL/WSDL metamodel

Basic activities [BPL, 2007]:

- `receive`: The `receive` activity allows the BPEL process to wait for a matching message to arrive and completes when the message arrives. It allows thus for the process to be invoked through its web service interface (its WSDL). Therefore, the partnerLink representing the BPEL process as well as the `operation` to invoke should be specified in the receive activity attributes. Other attributes might be specified such as the `variable` attribute which is used if needed to store input data.

- `reply`: The `reply` activity allows the BPEL process to send a message in reply to a previous inbound messaging activity, such as receive, `onMessage` or `onEvent` and thereby answers to a client waiting for this answer.

- `invoke`: The `invoke` activity is used to call Web services offered by service providers. The specification of the partnerLink to be invoked, as well as the operation to be performed are mandatory attribute. The invoke activity can enclose other activities, exception handling mechanisms using `catch`, `catchAll` or `compensationHandler` activities. correlations can also be defined in invoke activity.

- `assign`: The `assign` activity can be used to copy data from one variable to another, as well as to construct and insert new data using expressions. It can contain any number of elementary assignments, including `copy` elements which in turn contain `from` and `to` elements that specify source and target of the copy operation. BPEL supports XPath 1.0 language [Consortium, 1999] as an expression language by default.

- `throw:` used by a BPEL process to signal internal faults. A `faultHandler` element can use the data provided by the throw activity to handle the fault and to populate any fault messages that need to be sent to other services.

- `wait:` The `wait` activity is used to wait for a given time period or until a certain point in time has been reached. This can either be done by using a specific amount of time in the for element or a date that serves as deadline in the until element.

- `empty:` The `empty` activity can be used for doing nothing. This can be useful for instance, for synchronization of concurrent activities.

- `extensionActivity:` This activity can be used to extend BPEL by integrating new activities that are not part of the standard specification.

- `exit:` The `exit` activity is used to immediately end the business process instance. All currently running activities must be ended immediately without involving any termination handling, fault handling, or compensation behavior.

- `rethrow:` The `rethrow` activity is used in fault handlers to rethrow the fault they caught. It rethrows the specified fault, ignoring changes made to the original fault data by the a `faultHandler`.

Structured activities [BPL, 2007]:

Structured activities describe how a business process is created by composing the basic activities. BPEL define structured activities for various control-flow mechanisms namely sequential control (through `sequence`, `if`, `while`, `repeatUntil`, and the serial variant of `forEach` activities), concurrency and synchronization (through `flow` and `forEach` activities) as well as deferred choice (through `pick` activity).

- `sequence:` The `sequence` activity is used to define a collection of activities to be performed sequentially in the order in which they appear in the `sequence`.

- `if:` This activity provides a selection mechanism and allows the execution of one activity from a set of choices. It contains a condition which defines a boolean expression and an activity that is executed in case the condition evaluates to true. An optional number of `elseIf` elements can be defined each of which can define a condition and is executed in case its condition evaluates to true. An optional `else` element can be defined comprising an activity which is performed if no branch with a condition is taken in the whole `if` activity.

- `while:` It is a loop activity that allows a repeated execution of a child activity. The child activity is executed as long as the boolean condition evaluates to true. The condition is evaluated each time before executing the activity.

- `repeatUntil:` The `repeatUntil` is another loop activity. Its child activity is executed until the condition becomes true. It ensure thus at least one execution of the child activity.

- `forEach:` The `forEach` is another loop activity that gives the possibility to execute its contained activity a given number of times in two ways: in sequential order or in parallel (the parallel attribute set to “yes”). The contained activity must be a `scope` activity comprising a defined logic. The number of iteration is determined by “startCounterValue” and “finalCounterValue” attributes. A “completionCondition” may be used within the `forEach` to allow the `forEach` activity to complete without executing or finishing all the instances of the `scope`.

- `pick:` The `pick` activity is used to react to one of several possible received messages or for a time-out to occur. It must contain at least one `onMessage` (simalr to a `receive`) activity which executes its associated activity when a message arrives. Optional timer-based alarms may be defined in the `pick` activity with `onAlarm` activity.

- `flow:` The `flow` activity is used to perform parallel execution of one or more activities. A `flow` completes when all of the activities enclosed by the `flow` have completed. It enables the definition of synchronization relationships between its children activities through the `link` construct. Activities in the `flow` could be `sources` or `targets` of `Links`.

- `scope:` The `scope` activity provides the context which influences the execution behavior of its enclosed activities. It allows for the definition of its own `partnerLinks`, `variables`, `correlationSets` and handlers. Handlers may be of type `faultHandlers`, `compensationHandler`, `terminationHandler`, and `eventHandlers`.

2.2 Literature review

This section presents a State of the art that synthesize and analyze a set of research works in the literature dealing with the studied problem in this thesis. More particularly, works on architectural design decisions documentation, software quality documentation, quality documentation in service based-systems as well as works on assistance to software evolution have been covered.

2.2.1 Architecture decisions documentation

The concept of software architecture is recognized as an effective means to deal with complex software systems design issues. A software architecture is one of the first artifacts of the design process. It manifests the earliest design decisions of a software system [Bass *et al.*, 2003], and thus allows to analyze and evaluate the system early in the development process. The practice of software architectures in the last two decades has undergone significant evolutions on the representation and description aspects. The work of Kruchten et al [Kruchten *et al.*, 2009] presents an interesting historical view on how software architectures were addressed.

The concept of architecture design decision (AD) has been introduced the first time to the community of software architecture by Jansen et al. in [Jansen et Bosch, 2005]. According to them, an architectural design decision is defined as *"a description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture"*. This definition highlights the importance of this concept as one of the most important forms of *"Architectural knowledge"* (AK) in the software development process. The concept of "AK", is a new advanced research in the discipline of software architectures, which encompasses all acquired or formulated knowledge therein [Jansen, 2008]. Jansen et al. mentioned that this notion of "AK" is vital to the architecture construction process as it improves the quality of the process and the architecture itself.

Architectural design decisions (ADs) are decisions in the solution space[6] that directly influence the design of the software architecture [Jansen, 2008]. It could be for example, about choosing a particular architectural style or an architectural pattern. An architectural design decision (AD) has a rationale which defines the reasons behind it. Rationale describe why a change is made to the software architecture. It can include the basis for a decision, alternatives and trade-offs considered [ISO/IEC/(IEEE), 2011]. According to the standard ISO/IEC/IEEE 42010 [ISO/IEC/(IEEE), 2011], AD and rationale are considered as architecture description elements. Accordingly, documenting these two elements during the architectural development process is of great importance and has held a considerable attention of software architecture researchers community this last few years (see section 2.2.2 and section 2.2.1). Indeed, by keeping a traceability on ADs made during the development process and the reasons that led to these decisions, an architect can identify potential design conflicts and then avoid them. These latter, can happen when building the software architecture or during its evolution. Therefore, a software architecture documentation (of the architecture itself or, of ADs) is an efficient means to guide the architectural development process so that the involved decisions operate harmoniously.

Research conducted in the discipline of software architectures have shown important consequences due to the phenomenon of knowledge vaporization [Bosch, 2004 ; Jansen et Bosch, 2005]. This comes from the fact that, some details that a decision is based on, such as decision context, assumptions, decision drivers, consequences and considered alternatives, gets lost [Heesch et Avgeriou, 2009], or remains in heads of the designers [van der Ven *et al.*, 2006].

In this context, various approaches exist to describe and document this knowledge. One category of works focuses on the use of language constructs that allow to express architectural design decisions with respect to concepts that are defined at the "architecture descriptions level". Another category of works deals with the concept of architectural design decision as a first class entity explicitly defined, regardless of a particular architecture description. We will start first by presenting the first category of works represented by the architecture description languages, called ADL.

[6]The solution space is the domain containing all possible system solutions

Design decisions documentation in architectures description

ADLs *(Architecture Description Languages)* are languages that allow to specify software architecture descriptions. They provide a means to organize a software system in an assembly of components and connectors, a rather abstract view of the system. The components are units of computation or storage in the system and connectors are communication units between the components. Besides this ability, some ADLs allow the definition of architectural constraints that govern the allowed connection types by imposing restrictions on how the elements composing the system are arranged. Among ADLs that offer this possibility, we find the Wright [Allen, 1997] language. The latter integrates formal approaches to software architectures description and analysis and more particularly the formalization of connectors. Constraints in Wright are defined by predicates and cover any element of the architecture (Components, Connectors, Ports, Roles,etc). The following constraint stipulates that an architecture configuration must have a star topology:

$$\exists center : Components \bullet$$
$$\forall c : Connectors \bullet \exists r : Role; p : Port \mid ((center, p), (c, r)) \in Attachments$$
$$\wedge$$
$$\forall c : Components \bullet \exists cn : Connectors; r : Role; p : Port \mid ((c, p), (cn, r)) \in Attachments$$

The first predicate indicates that there is a component ("center") attached to all connectors of the description. The second predicate indicates that all components must be attached to a connector. Thus, this constraint ensures that every component is connected to the component representing the center of the star. Therefore, it gives a formal way to document the star architectural style as an architectural design decision.

Armani [Monroe, 2001] is another constraint language that extends and complements the Acme [Garlan *et al.*, 2000] ADL . It allows the description of software architectures, imposing constraints on the evolution of the elements composing these architectures, and especially the capture of expertise in the design of software architectures. It allows to describe three classes of architectural design expertise: design vocabulary, design rules, and architectural styles. Design vocabulary specifies the basic elements for system design. It describes the selection of components, connectors, ports, systems, properties, roles and representations that can be used in the system design. Design rules specify heuristics, invariants, and composition constraints to assist architects in the design and analysis of software architectures. The expression of constraints on a system architecture is in the form of first-order logic invariants. Armani allows the association of design rules with a complete architectural style, a collection of connected design elements, a design element type, or an instance of a component or connector. Armani predicates language provides various features, such as terms composition, the ability to define its own functions or use predefined functions. Among the predefined functions we find, the type functions (example: declaresType (e: Element, t: ElementType): boolean), graph connectivity functions (example, connected (c1, c2: Component): boolean), parent-child functions (example, parent (c: Connector): System), set functions (for example, sum (s: set{number}): number). The Armani predicate language includes also logical, arithmetic and comparison operators. It distinguishes two types of constraints: *"invariants"* and *"heuristics"*. The following example illustrates their use:

```
1 Invariant Forall c1,c2 : component in sys.Components |
2 Exists conn : connector in sys.Connectors |
3 Attached(c1,conn) and Attached(c2,conn);
4 Heuristic Size(Ports) <= 5;
```

The constraint expressed by the invariant (Lines 1, 2, and 3) imposes that components should be connected in pairs to form a complete graph. The heuristic (Line 4) states that the number of all ports must be less or equal to five.

The last class of expertise concerns architectural styles. Examples on specifying architectural styles using Armani can be found in [Monroe, 2001].

Undoubtedly, one of the most relevant solutions to promote reuse in software architectures is the use of architectural styles. An architectural style defines a family of systems by providing an architectural design vocabulary specific to a domain with restrictions on how parts can be grouped together [Kim et Garlan, 2010]. A specification of a style in Armani consists of a declaration of a design vocabulary that can be used to design styles, and a set of design rules that guide the composition and instantiation of the design vocabulary. In the category of ADLs, Wright provides the ability to define architectural styles.

In [Kim et Garlan, 2010], Kim and Garlan propose an approach for transforming architectural styles (a form of architectural design decision) formally expressed by the Acme ADL, to relational models expressed with the Alloy [Jackson, 2002] language. Alloy is a modeling language based on first-order logic (first order relational logic). The method, therefore, allows to describe architectural style specifications in the Alloy language constructs through a style translation scheme. The aim is to be able to verify properties on styles, namely, if a style satisfies some predicates set on its architectural structure, the consistency of style (the existence of at least one configuration that complies with style architectural constraints), or whether two styles are compatible for composition.

All the works that have just been addressed consider a high level abstract description (components, connectors, configurations etc.) of architectural design decisions (mainly as architectural styles) used to build an architecture. They are often provided with constraint languages and mechanisms allowing the maintain and verification of structural properties of those architectural design decisions.

Design decisions documentation related to architectures description

The other category of works on documenting architectural decisions treats architectural decisions as first-class entities and aims to represent them as well as their rationale (Design Rationale) explicitly in the software architecture documentation. This category is annex to architecture descriptions, which is specified using ADLs. The goal is to capture knowledge related to the architectural decisions made during the software development to reduce the effects of knowledge vaporization phenomenon. This idea comes from the fact that most decisions made during the software architecture construction remains implicit or as non-persistent intentions. Consequently, consider the architectural decisions as first-class elements and representing them explicitly in architectural documentation is one of the most interesting ways to improve the quality of software architectures as we will show later.

The work of Perry and Wolf [Perry et Wolf, 1992] was one of the first major contributions to software architecture description. They have introduced the following definition for software architecture:

Software Architecture = { Elements, Form, Rationale }

Architectural elements are of three different types: i) processing elements; ii) data elements; and iii) connecting elements; Processing elements are components that perform the transformation on the data elements. The data elements are those that contain the information that is used and transformed. The connecting elements make the connection between the first two elements (the glue that holds the different pieces of the architecture together).

The Form consists of a set of properties and relationships that define constraints on the architecture elements and how they interact. Properties are used to constrain the choice of architectural elements, whereas the relationships are used to constrain the placement of those elements. An architecture may have different forms while representing the same functionality.

The rationale capture the architect's motivations for some architectural choices (the choice of architectural style, the choice of elements, and the form). Perry et Wolf mentioned the use of "views" for building software architectures while respecting the concepts proposed in their model. These latter represent different aspects of the software architecture, which reflect the different concerns of its users.

The 4+1 view model [Kruchten, 1995] has been proposed in the same context. It marked a new era for software architectures description and documentation. This model organizes the description of a software architecture using four different views, namely: logical view, process view, development view, the physical view, and use a set of scenario (the use cases view; plus one) to check their correctness. All these views represent concerns of the different participants in the development of a software system. UML (Unified Modeling Language) is generally the language used to represent these views, but other notations and tools could also be used as well. Architectural decisions that appear in the software architecture can be elaborated on any of the views or captured by combining different views of the 4+1 model. The 4+1 approach has been adopted as a foundational part of the RUP (Rational Unified Process) approach [Clements *et al.*, 2003].

The notion of "view" has been taken into account some years later with the emergence of the IEEE 1471-2000 [IEEE, 2000] standard, but with more refinement on how views should express certain aspects of the software architecture. The standard defines the view to express a system architecture according to a particular "viewpoint". This concept determines the languages to be used to describe a view, the modeling methods or the associated analysis techniques that are applied to the representations of the view. The viewpoint address concerns (such as NFRs) of participants in the development of the software system. The conceptual model of the standard has been improved compared to the initial version, to incorporate as a first-class element, the rationale of architectural decisions. It includes, in addition, other elements namely the mission to be fulfilled by the system, the system environment, and a viewpoint library.

Clements et al [Clements *et al.*, 2003], proposed in their approach V&B (*Views and Beyond*) three different viewtypes for documenting a software architecture:

- The module viewtype, which responds to the way the software architecture should be structured as a set of implementation units;
- The component and connector viewtype (C&C for Component-and-Connector), allows to structure the architecture into a set of elements that have a runtime behavior and interactions;
- The allocation viewtype, answers how the architecture is linked to non-software structures of its environment.

The documentation approaches that we have illustrated [Kruchten, 1995 ; IEEE, 2000 ; Clements *et al.*, 2003], aim to define a set of views on a system elements and their relationships to describe a software architecture. We will now introduce some works that have been proposed in order to reduce knowledge vaporization, by making explicit the representation of architectural decisions.

In 2005, Jansen and Bosch [Jansen et Bosch, 2005] proposed a new way of perceiving a software architecture. They presented it as a set of architectural design decisions. In this work, the authors presented a new approach for software architectures development named “Archium”, which considers as stated above, that software architecture is a set of explicit architectural design decisions. The approach is based on an architectural decisions conceptual model that describes architectural decisions elements (Problem, Motivation, Cause, Solutions, Decision, trade-off, Architectural Modification and Context) and their relationships. The description of a software architecture is done through the conceptual model composed of the notions of: i) Deltas (part of the architectural model) that express a change in the behavior of a component and represent its functionalities; ii) design fragments; iii) and architectural decisions. The addition of an increment on the old system software architecture is obtained by the use of a composition model. This latter, provides the necessary elements for linking the architectural model (defines software architecture concepts similar to those used in architecture models namely: port, connector, interface,,etc.) elements with those of the design decision model (contains a design decision as a first class concept). It connects the changes made by the design decision model with elements of the architectural model. The final architecture is thus a set of architectural decisions.

Kruchten et al [Kruchten *et al.*, 2006] define architectural knowledge as follows:

AK = Design Decision + Design

This formula confirms the vision introduced by Jansen and Bosch. It considers the architectural decisions as an essential part of knowledge that contributes in the construction of software architectures. The authors mentioned three levels of knowledge that can be applied to categorize the architectural knowledge: i) Tacit: as intentions mostly in the head of people; ii) Documented: there is some trace of this knowledge; iii) Formalized: not only documented, but organized in a systematic way. The best way that seems obvious to us to preserve this knowledge, is to adopt the formalization level. This choice is motivated by the possibility of realizing an automatic decisions management.

The work of Kruchten [Kruchten, 2004] provides a taxonomy of architectural decisions. It could enable the construction of complex graphs of interrelated design decisions to support reasoning about them. He presents a model for describing architecture decisions, including `rationale`, `scope`, `state`, `history of changes`, `categories`, `cost`, `risk`, etc. He identifies in this ontology the different possible relationships between design decisions and links between design decisions and design artifacts.

In [Tyree et Akerman, 2005], Tyree and Akerman discuss the importance of documenting architecture decisions and their specification as first-class entities in an architecture description. They point out that a simple document describing the key architectural decisions can help significantly to clarify the systems architectures. To this end, they present a template specifically designed for architecture decision documentation, which embeds interesting information characterizing architecture design decisions on which a description and a documentation of a decision is elaborated (status, assumptions, implications, related artifacts, constraints, ...). The template is derived from two models *"REMAP"* [Balasubramaniam et Vasant, 1991] and *"DRL"* [Jintae, 1989]. The authors provide as mentioned in their work an alternative documentation form to that relying on a set of views. They also mentioned that, in their approach, they first identify what decisions are important. These decisions drive then architecture, and hence the views which is different from the V&B approach. In this latter, the architect uses "the view selection scheme" to decide which architecture view he/she wants to produce. Then, the view identifies a family of design decisions that the architects wants to resolve and be able to express. The proposed template has a rich vocabulary for describing architectural decisions, and may provide an eventual support for impact analysis for architecture evolution on software functionalities. This could be achieved based on the information documented in the field "Related artifacts". It specifies the elements that the decision impacts (functional changes could be isolated).

Lago and Van Vliet [Lago et van Vliet, 2005] talk about explicit documentation of the reasons for architectural decisions which they called *"assumptions"*. They proposed an approach to make these assumptions explicit considered as invariabilities in the system, and link them to software architectures documentation. This should enrich the documentation and provide better support for evolution and maintenance.

Bass et al. [Bass *et al.*, 2006] consider architectural design decisions and the Rationale, as the most important form of knowledge to capture. They defined the architectural decision by the notion of the architecture transformation from a state before applying the decision towards the state after its application. Based on this notion they suggested to document architectural decisions and their rationale by means of two graphs. The first named causal graph, is a directed acyclic graph that organizes decisions (represented by the nodes) in a temporal order. It considers design as a sequence of decisions (transformations) with which one can trace the genealogy of all made decisions. The second graph represents another aspect of the transformations, which is the selection of architectural patterns or tactics allowing the implementation of architectural decisions. This gives all architectural elements as they exist in the software architecture at a given time during development. The combination of the two graphs provides answers about the manner in which the software architecture takes a certain form, i.e. all architectural decisions in a chronological order of appearance on the architecture by means of the causal graph, as well as their impact on the architecture represented by a structural graph for a given level of the causal graph. Both graphs can provide valuable information about software architecture and thus improve the quality of the architectural design process. They provide the reasons for all made decisions, and unnecessary paths already chosen.

In [Capilla *et al.*, 2007], the authors proposed a way to characterize the architectural knowledge particularly architectural decisions in order to define a management process of these latter under an evolution context. The greater part of this knowledge remains implicit in the architect's intentions and tends to disappear over time carrying with it all the reasoning (alternative solutions, chosen paths) that is related to the current software architecture. This work, reinforces further the idea of documenting architectural decisions as first-class entities explicitly during the software architecture development process. To this end, the authors defined attributes to describe architectural decisions by separating according to their degree of importance, mandatory attributes and optional attributes. The first class introduces information associated with architectural decisions that should be defined throughout the system lifetime, including a `decision name` and `description`, the `constraints`, the `dependencies` (between decisions), the `status`, the `rationale`, the `design patterns`, the `architectural solution`, and the `requirements`. The second class provides additional information that can be chosen according to user preferences such as, the `alternative decisions`, `assumptions`, `pros and cons`, `category of decision`, or `quality attributes`. In addition to these attributes, they have defined attributes to support the evolution of architectural decisions, including the `date and version`, the `obsolete decision`, the `validity`, the `reuse times and rating`, and the `trace links`. Architectural decisions are described by the decision model which is part of the meta-model proposed by the authors for the construction and evolution of architectural knowledge. The other two parts of the meta-model are: the "Project model" that contains information for building software architecture, and the "Architecture" representing the software architecture described by one or more architectural views. The meta-model integrates two different types of architectural knowledge: the "Product" type which describes architectural decisions through the attributes shown above, and the "Process" type that expresses the decision-making activities undertaken by architects to store, manage, assess, document, communicate, discover and reuse architectural design decisions.

Kruchten et al [Kruchten *et al.*, 2009], confirm the need to integrate architectural decisions and their rationale as first-class entities in software architectures documentation. They presented a historic evolution of the software architecture representation, which covers three periods. The first focuses on the use of architectural views, notably with the emergence of the famous 4+1 view model. The second is characterized by the appearance of new methods which complemented the description of views, such as IBM's RUP (Rational Unified Process) method, or SEI (Software Engineering Institute) methods of such as ATAM (Architecture Trade-off Analysis Method), ADD (Attribute Driven Design) and SAAM (Software Architecture Analysis Method). All these methods are used for software architectures analysis and evaluation. The authors mentioned that the common point of these methods is the use of architectural decisions, which brings us to the last period that highlights this notion. It is interested in investigating the representation, capture, management, and documentation of design decisions made during the construction of architectures. Kruchten reinforced this idea and included in the 4+1 view model, the "decision view" that incorporates design decisions. The authors provided guidelines to allow the capture and representation of architectural decisions, and help architects to document them.

In [Durdik, 2011 ; Durdik et Reussner, 2012 ; Ton That *et al.*, 2012], the authors proposed a process which is based on the use of a Pattern catalog to document patterns as identified architectural design decisions. Indeed, in [Durdik, 2011 ; Durdik et Reussner, 2012] the authors use questions to help architects in selecting and validating the most appropriate patterns.

2.2.2 Software quality documentation

The term "quality" comes from the Latin *"qualitas"*, derived from the word *"qualis"* meaning "what". It signifies the nature of an element. The quality concerns unquantifiable characteristics, which opposes it linguistically to quantity. The term "quality" has been associated with software since a long time. Since the appearance of the earliest methods of software development, we began to identify indicators that can help to give a global appreciation of a software.

The term "software quality" is a complex concept that exposes several facets. Among these facets, we find the quality of a software product, the quality of a software process, which covers various software development phases, or also the quality of software resources. We present in this section an overview on the different methods for documenting the quality of a software product. Two approaches are discussed, the quality models aiming at characterizing the quality of a finished software product, and the one that makes use of software architectures, which are considered as specific software products, to document quality properties.

Quality models

Research conducted in quality modeling have produced over the last thirty years, a multitude of quality models that have been applied at different degrees of success. Despite the diversity and heterogeneity of existing quality models, there is no clear definition of what is a quality model. This stems from the fact that, these apply in different contexts, and have rather distant targets ones of each other.

In [Deissenboeck *et al.*, 2009], a quality model is defined as a model that aims to describe, evaluate and/or predict software quality. This definition proposes the classification of quality models according to three different objectives namely the definition, assessment, and prediction of software quality, thereby separating these models into three categories: definition models, assessment models and prediction models. The work of [Kläs *et al.*, 2009] provides a broader classification scheme, of a wide range of quality models, the best known in the literature. Inspired by the GQM (Goal/Question/Metric) Template the authors provide five dimensions for classification which are: 1) *object*: specifies what is being examined by a quality model. The major classes of objects are products, processes, and resources; 2) *purpose*: specifies the intent/motivation of quality modeling (specify, measure, evaluate, monitor, predict, improve, manage, etc.); 3) *quality focus*: specifies the quality characteristic being modeled; 4) *viewpoint* (stakeholder): specifies the perspective from which the quality characteristic is modeled. The viewpoint may be, for example that of the developer or the user; 5) *context*: specifies the environment in which the quality modeling takes place.

By fixing the first two dimensions (object: the products and the purpose: specify, define, control, improve and manage), we will thus limit ourselves to present in the remainder of this section, the main approaches for the documentation of software product quality.

The most popular quality models are based on a decomposition approach commonly known by the name of FCM (Factor-Criteria-Metric) quality models. They are usually designed as a tree where the higher level of the hierarchy defines high-level abstract quality attributes and the lower level defines concrete quality criteria that can be measured by metrics.

One of the most known models is that of McCall [McCall *et al.*, 1977]. It includes eleven factors covering three perspectives to define and identify the quality of a software product. The first perspective represented by maintainability, flexibility and testability covers the software product revision (the ability to undergo changes). The second concerns the product transition that defines the software product adaptation to new environments. It is represented by the portability, reusability and interoperability. The last perspective is interested in product operation. It has the following attributes: Correctness, efficiency, reliability, integrity and usability.

So the model describes these factors in a hierarchy of twenty three quality criteria. The eleven factors describe the software external view, as perceived by users. The twenty three criteria describe the software internal view, as perceived by developers. The last level of this decomposition represents the metrics that are associated with criteria. They are used as the measurement method and are intended to capture some aspects of quality criteria.

The second of the founders models is that of Barry Boehm [Boehm *et al.*, 1976] proposed in 1976. It resembles that of McCall in that it uses the same decomposition method to characterize quality attributes. It is a hierarchical model structured on three levels. The top level addresses software users concerns:

- As-is utility : the extent to which the as-is software can be used, from three points of view: ease of use, reliability and efficiency
- Maintainability : to what extent should it be easy to maintain.
- Portability : ease of changing software to accommodate a new environment.

The intermediate level, represents seven quality attributes which together describe the expected qualities of the software system: portability, reliability, efficiency, usability, testability, understandability, flexibility. The last level in Boehm's model, represents the metrics associated with the characteristics of the previous level. This model is based on a wide range of characteristics compared to that of McCall, and is particularly focused on maintainability.

FURPS [Grady, 1992] is another model that was proposed later by Robert Grady at HP (Hewlett-Packard), less known than the previous two models. It was extended by IBM Rational Software to FURPS+. FURPS means: Functionality, Usability, Reliability, Performance and Supportability. The model decomposes characteristics into two different categories of requirements: i) functional: represented by "functionality" characteristic which identifies a set of features, and includes the "security" quality characteristic; and ii) non-functional (URPS) represented by the following features:

- Usability : may include human factors, aesthetics, user documentation, etc;
- Reliability : may include the frequency and severity of failure, recoverability, predictability , Mean Time Between Failures;
- Performance : include efficiency, response time, resource consumption, etc;
- Supportability : include maintainability, testability, compatibility, adaptability, etc.

In 1996, Geoff Dromey [Dromey, 1995 ; Dromey, 1996] introduces a new quality model associated with software products, which resembles its predecessors. Dromey's vision is that we can not build high-level quality attributes such as maintainability or reliability directly in a software. Instead, we can identify and construct a coherent set of internal tangible properties or characteristics (of low-level). These latter determine and exhibit external high level quality attributes. The author has identified three main elements for a generic quality model: i) the product properties which influences the quality; ii) high-level quality attributes; iii) and means for linking the product properties with the quality attributes. A product property in the model of Dromey is linked to a component of a product type in software development, starting from requirements specification, to implementation. A product is composed of multiple components. Some are simple, others are made of a set of simpler components. The model identifies four types of product properties. For each type of property it defines a number of quality attributes that are influenced by these latter :

- Correctness properties : define the properties to be respected, either directly by a component or a composition of components (a context) to function properly. For example, a variable in an implementation (product) is a component that may have as a quality-carrying property “assigned” or “precise”. If a variable does not carry any of these properties, the correctness may be affected. This type of properties affects the “Functionality” and “Reliability” quality attributes.

- Internal properties : specify the normal form of a component that defines its interior (structure) and that should not be violated whatever the context. For example, the body of a loop must always ensure progress toward termination. These properties measure if a component has been well deployed or composed. They affect “Efficiency”, “Maintainability”, “Reliability” quality attributes;

- Contextual properties : these properties are associated with the individual components, and address quality problems arising from a composition of a large number of components. They affect the “Reusability”, “Portability”, “Maintainability”, “Reliability” quality attributes;

- Descriptive properties : determine if a software product is easy to understand and use for its intended purpose. For example, giving a name to a variable that is not suggestive may affect descriptive properties. They affect the “Usability”, “Reusability”, “Portability”, “Maintainability” quality attributes;

Links between properties and quality attributes are not formally established, but are based on the classification of properties. For example, in order for a product to satisfy its functionalities, in a reliable way, correctness properties of all its components should be satisfied. Thus, the latter affects “Functionality” and “Reliability” quality attributes. Note that the links just mentioned for each property type, are specific to the implementation quality model. The model was primarily applied on implementation products, but it is generic and allows to build quality models for the requirements or design.

Kitchenham et al [Kitchenham *et al.*, 1997] proposed a quality model called SQUID (Software QUality In Development) with a hierarchical structure encapsulating and inspired by the first version of ISO/IEC 9126 model [ISO, 2001] and McCall model. It is a composite model reflecting the different aspects of a quality model as components representing the structure and content of the latter. It contains elements of both models (quality characteristics, quality sub-characteristics, internal software properties that affect sub-characteristics, and measurable properties). The authors state that

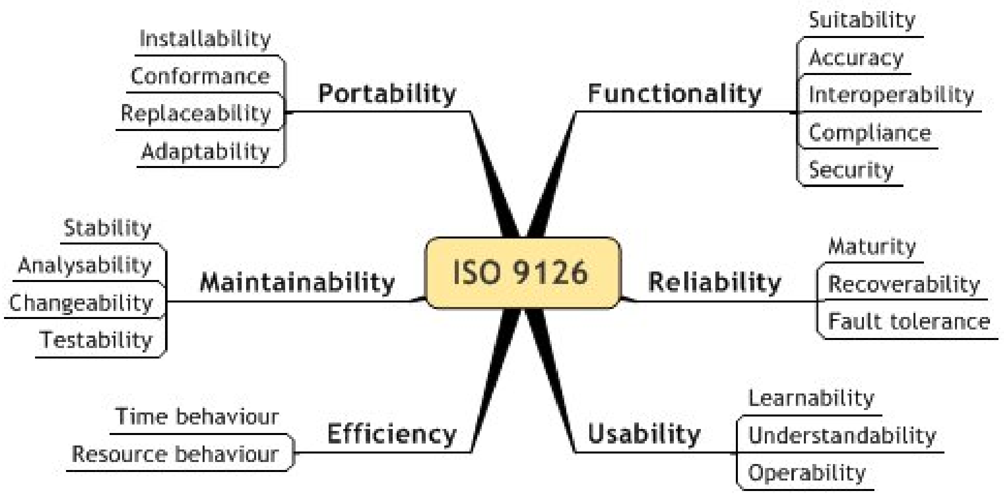


Figure 2.4 : An excerpt of the ISO/IEC 9126 quality model

SQUID's philosophy is that, you can not get a specification of quality requirements only by referring to a quality model, but from the specification of the desired behavior of a specific product. Therefore, a third component is necessary, in addition to the structure and content to meet quality requirements, which is that of a product quality model. This latter is the instantiation of a quality model of a specific product. In this model all elements are measured by metric assignment and values. The SQUID approach for software quality modeling is provided with a set of tools for specifying quality requirements.

ISO has provided in 2001 a new version of ISO/IEC 9126 [ISO, 2001] standard to evaluate software products: quality characteristics and guidelines for their use. The standard is based on the model of McCall and Boehm. Aside from being structured in the same way that these models it includes the *"functionality"* characteristic as well as the identification of internal and external quality characteristics of a software product. It consists of four parts: i) quality model; ii) external metrics; iii) internal metrics; and iv) quality in use metrics. The quality model (See figure 2.4) defines a hierarchy of quality characteristics (factors) on two levels, the quality characteristics (functionality, portability, maintainability, efficiency, usability, reliability) and their corresponding sub-characteristics.

The quality characteristics are defined as follows:

- functionality: The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.

- portability: The ability of a software to be moved from one environment to another and ease of integration, adaptation, installation and co-existence.
- maintainability: The ability of software to be easily analyzable, testable and modifiable.
- efficiency: The ability of software to provide its services effectively with respect to the execution time and system resources consumption.
- usability: The ability of software to be attractive, easily understandable and operable.
- reliability: The ability of software to provide its services under specific conditions and for a certain period

Despite this diversity of the proposed FCMs models and their popularity, they showed some limitations and have received several criticisms [Deissenboeck *et al.*, 2009 ; Kitchenham *et al.*, 1997 ; Deissenboeck *et al.*, 2007 ; Marinescu et Ratiu, 2004 ; Broy *et al.*, 2006]. Therefore, they failed to establish an acceptable basis for quality evaluation. The reason for this is the desire to condense quality attributes, as complex as maintainability into a single value and the fact that these models are usually limited to a fixed number of levels. Most of these models suffers from the lack of guidelines and decomposition criteria of complex quality concepts, which makes difficult their refinement and their localization in some large size quality models (eg. Usability can not be decomposed to measurable properties in only two steps according to the three levels of FCM models). It is also reproached, to this type of models to not be able to find the real causes of quality problems during analysis or evaluation of an object-oriented design. This is due to the significance of the metric values that reflect the presence of a design or implementation problem (the symptoms) and not the problem itself which makes treatment difficult [Marinescu et Ratiu, 2004].

Other works have been conducted at the SEI (Software Engineering Institute) which led in 2002 to a model of a different kind that offers quality attributes characterization of a software architecture, structured into three classes [Bass *et al.*, 2003]. The first class covers the system qualities and comprises the following attributes: availability, modifiability, performance, security, usability and testability. The authors state that other attributes can be found and added in the taxonomy of quality attributes, such as portability, captured as the modification of the platform. The second class is interested in business qualities and identifies some attributes which are: time to market, cost and benefits, projected lifetime of the system, targeted market, rollout schedule and integration with legacy systems. The third class covers the qualities directly related to architecture: conceptual integrity, correctness and completeness, and buildability. All these qualities represent goals for the software architect. The authors found that among the raised issues, that the definitions provided for these quality attributes are not operational and can overlap, thus not reflecting the context in which they are applied. For example, all systems are modifiable with respect to a set of changes and are not with respect to another. Or, to which quality should we classify an aspect, such as the failure of a system (availability, security, and usability). To remedy this problem they proposed a mechanism to characterize quality attributes through quality attribute scenarios which are comprised of six parts:

- Source of stimulus : it is the entity that generated the stimulus (a human, a software system, or other actuator);
- Stimulus : it is a condition that needs to be considered when it comes at a system;
- Environnement : represents the conditions under which the stimulus occurs;
- Artifact : represents stimulated parts in the system, or the entire system;
- Response : The response is the activity undertaken after the arrival of the stimulus;
- Response measure : the response should be measurable, in such a way the requirement can be tested.

More recently, in 2003, Georgiadou et al. have proposed GEQUAMO (Generic Quality MOdel [Georgiadou, 2003]). This model encapsulates the requirements of different stakeholders (developer, user, etc.) in a dynamic and flexible manner allowing each of them to build and customize its own model, which reflects the importance of each requirement, according to his/her point of view. It is a multilevel model that is built using a combination of two types of diagrams namely CFDs (Composite Features Diagramming) developed by the author and *"Kiviat"* diagrams. CFDs provide a qualitative way to describe the profile of an element under evaluation. They consist of a set of concentric circles which express increasingly lower details (sub-characteristics). The characteristics and sub-characteristics are built gradually with less detail in a tree structure. At each node and based on the number of sub-characteristics, we can build a polygon (triangle, rectangle, etc.) which constitute Kiviat diagrams. These latter, can represent quantitative information on requirements or characteristics of each level.

To meet the needs of the problematic discussed in chapter 1 of this thesis, this imposes the choice of a quality model for the characterization of the different quality attributes (concertized by architectural decisions) of a software product. However, the models which have been synthesized, offer this ability but in different ways. Indeed each model has its own vision and defines its own characteristics despite the fact that together these models share a variety of quality characteristics. Most of these models constitute the result of personal efforts, which explains this diversity in points of view and interpretations for their quality characteristics. In addition, these models are mainly applied to a software product in general, except for the Dromey model which mainly focused on implementation products (codes). Similarly to the SEI model that provides a classification of quality attributes for software architectures, but does not interest us for certain types of attributes like for example, business properties. So we chose to adopt the ISO 9126 standard to represent and characterize the quality properties. This one, constitutes the result of the international community consensus for the quality of a software product, and applies to all levels of the development process. Indeed this model seems to be the most appropriate for the purposes of this thesis because it is the most complete and representative of quality characteristics for service-based systems at design time. Indeed, we consider in this thesis static quality characteristics which are measurable at design time. Dynamic quality characteristics such as response time are not taken into account.

Quality attributes documentation in software architectures

Many works have been proposed on quality requirements capture and specification. These works try to anticipate quality assessment, and process the quality aspect by capturing and documenting quality requirements (commonly called NFRs for Non-

Functional Requirements) in software architectures. Considered as an important artifact in software development, software architecture appears to be an appropriate level to study the quality of software.

The conducted research on NFRs could be classified into two categories, product-oriented and process-oriented approaches. The first class covers the description of non-functional requirements in order to measure or observe them on a software product, while the second proposes the means that aims to identifying, modeling, and management of NFRs. These two approaches (product-oriented and process-oriented) are complementary and together enable to represent and use the non-functional requirements [Mylopoulos *et al.*, 1992].

One of the major works in the literature is that of Mylopoulos et al. [Mylopoulos *et al.*, 1992]. Following a process-oriented approach the authors propose a framework for the representation and use of Non-functional requirements during the development process. The framework includes five components allowing, following a goal-oriented process, to justify and argue design choices made to satisfy certain software quality requirements. These components are: i) a set of goals for representing NFRs, design decisions and arguments in support of or against other goals; ii) a set of link types for relating goals or goal relationships to other goals; iii) a set of generic methods for refining goals into other goals; iv) a collection of correlation rules for inferring potential interactions among goals; v) finally, a labeling procedure which determines the degree to which any given NFR is being addressed by a set of design decisions. The authors consider NFRs as `goals` to be achieved by validating the right design decisions and their rationale, considered in turn as goals. Thus, the system design is guided by NFRs, by building a graph containing the possible trade-offs between design decisions that implement them and their Rationale. This approach addresses qualitatively quality requirements satisfaction by anticipating the evaluation during the design process.

In Cyneirios et al. [Cysneiros et Sampaio do Prado Leite, 2004] propose an approach based on Mylopoulos's framework for capturing and representing NFRs and their interdependencies. Their approach shows the integration of NFRs in functional requirements models. The authors were interested in conceptual models expressed in UML by incorporating NFRs descriptions in class, sequence, and collaboration diagrams.

Other design methods in the literature allow the construction of software architectures that address Non-functional requirements. Bass et al. [Bass *et al.*, 2001], proposed

ADD method (*Attribute-Driven Design*) which is similar in the spirit to Mylopoulos's method. It follows an architectural design process guided by quality requirements. The idea behind is that design decisions are influenced by the quality requirements to meet. The authors proposed for this purpose the concept of attribute primitives (architectural patterns), which are collections of components and connectors collaborating to satisfy some quality attributes. These attributes are documented as general scenarios. Examples of attribute primitives are a "data router", a "cache and the components that access it", or "fixed priority scheduling". Each of these primitives targets and realizes a quality attribute. For the given examples, we have the "Maintainability" and "Performance". Indeed, the first primitive limits the knowledge that producers and consumers have on each other and thus affects Maintainability. Similarly, the second primitive keeps a copy of data accessible to components that use it providing thus better performance.

The architectural design in ADD follows a decomposition and refinement process. At each decomposition step, attribute primitives are selected to satisfy a set of quality scenarios. Then functionalities are allocated to instantiate connectors and components provided by the primitives. Take the example of the attribute primitive "data router" as a solution for the quality attribute "Maintainability". This primitive defines three types of design elements: "Producer", "Consumer" and the "data router". According to a certain functional requirements specification, different functions are determined. A sensor function that produces data value, a guidance function as well as a diagnosis function consuming the data value. The element type "Producer" is instantiated into a "sensor". The element type "Consumer" is instantiated for functions "guidance" and "diagnosis". While the "data router" could be instantiated into a "blackboard".

In [Bass *et al.*, 2003], the authors proposed architectural tactics, in the same spirit as the primitive attributes to guarantee quality characteristics such as maintainability, performance and security in software architectural design.

The ABAS (Attribute-Based Architectural Style) were proposed by Klein et al [Klein *et al.*, 1999] as an improvement of architectural styles by associating to them reasoning frameworks based on quality attribute models. The ABAS can be used during software architecture analysis and design. They allow reasoning about architectural decisions guaranteeing certain quality attributes. The characterization of these latter is performed based on scenarios. The authors proposed several types of ABAS, such as: the synchronization ABAS for performance quality attribute, the layered ABAS for maintainability, and redundancy ABAS for availability.

In [Niemelä et Immonen, 2007] the authors proposed QRF method (Quality Requirements of a software Family), which focuses on the representation and transformation of quality requirements to architectural models for software product families. It also allows quality evaluation in the early stages of development. Quality requirements representation in a software architecture is done through a set of architectural views. This step which constitute the final step after a series of analysis steps (impact, quality, variability, and domain) includes two activities: selecting the styles and patterns supporting different qualities and describing specific qualitative constraints.

Kim et al. [Kim *et al.*, 2009] presented an approach for representing NFRs in software architecture using architectural tactics as reusable architectural building blocks. The latter and their relationships are represented as *Feature Models* and their semantics is defined with the RBML language (*Role-Based Meta-modeling Language*). Architectural tactics satisfying quality attributes are selected and composed into one tactic encompassing all the desired qualities. The resulting tactic is then instantiated to create a software architecture that incorporates NFRs for the system under development.

Marew et al [Marew *et al.*, 2009] proposed an approach inspired from the works of [Mylopoulos *et al.*, 1992 ; Chung et Nixon, 1995 ; Chung *et al.*, 1999]. It aims at integrating NFRs handling in analysis and design phases as with functional requirements to fill the gap between the elicitation and implementation of NFRs. The authors introduced in the phases prescribed by the object-oriented approach other phases relevant for modeling NFRs in the analysis and design phases. They provided a tactic's types classification scheme to model NFRs namely *Analysis Tactics* (AT) and *Design Tactics* (DT). The first category affects the analysis model while the second affects the design model. For the first category, they grouped tactics under the types for *Addition of Operations/Attributes, Addition of New Classes, Restructuring,* and *Using Specific Algorithm/ADT*. Under the design tactics, we find tactics for *Introducing New Behavioral Elements* and *Modifying Existing Elements.* They start first in their approach, by designing the *"Softgoals Interdependency Graph"*(SIG) [Mylopoulos *et al.*, 1992 ; Chung et Nixon, 1995 ; Chung *et al.*, 1999] and classify the tactics that realize NFRs according to the classification scheme. Then, ATs are modeled using "classpects" [Rajan et Sullivan, 2005] (combines the concept of *class* and the one of *aspect*), classes, new algorithms, etc after thorough understanding of the analysis model. Design tactics are also modeled using the design model that results from the design phase. The output of this step is an integrated design model that satisfies both FRs and NFRs of the user. After tactics modeling (ATs and DTs), tradeoff analysis is made to analyze the relationships among NFRs. The authors proposed Q-SIG, an improved quantified version of SIG coupled with prioritization on NFRs to arbiter between different competing requirements.

In [Al-naeem *et al.*, 2005], Alnaeem et al proposed “ArchDesigner”, a quality-driven approach for facilitating the architectural design of distributed software applications which use optimization techniques to determine optimal combination of design alternatives that best satisfy stakeholder’s quality goals and project constraints. Architectural design decisions in their work are high level architecture design decisions (the choice of Java EE, for example) to be applied in some ways, which are proposed to the architects in combination with other decisions as candidate (output) software architecture that satisfies the quality goals.

In [Choi *et al.*, 2006], the authors present an approach, called *“AQUA”*, to quality achievement at architectural level based on design-decision making. They used an evaluation contract (between users and software architects) for quality attributes identification, then a process to manually find high level architectural design decisions achieving these quality attributes. The authors of this paper used a decision graph transformation strategy to analyze the impact of applying a design decision alternative on the software architecture.

2.2.3 Quality achievement in service-based systems

A plethora of works have been proposed in the literature to integrate and satisfy quality requirements in service-based architectures such as, languages, middlewares, composition algorithms, models and processes. The focus of these works ranges from the specification to the maintenance software process phase. A more detailed very recent study on the different approaches dealing with quality requirements in service based-systems is proposed in [Neto *et al.*, 2016].

In the following we present some works dealing with the so called QoS-aware composition problem [Zeng *et al.*, 2003] which consists in, given a composition, finding the set of services that optimizes some QoS attributes under given QoS constraints.

In [Canfora *et al.*, 2008] the authors proposed a QoS (Quality of Service) aware composite service binding approach which is based on Genetic Algorithms (GAs). This approach dynamically satisfies and maintain quality goals of existing (already designed) composite Web services. It operates on designed composite services such as those written using WS-BPEL language. The composite service defines several abstract services each of which can be bound to a list of concrete services. The approach determines the optimal set of concretizations (a correspondence between abstract and concrete services) by binding at execution time for each abstract service the concrete service that meets the quality constraints imposed by the SLA (Service Level Agreement) contracts (between the provider and the consumer of a service). To do so, the approach estimates the QoS of the composite service using a formula that defines aggregation functions for each pair quality-attribute/composition language control statement such as Sequence, Switch, Loop or Flow. It also predicts Qos deviation (the QoS of the actual composite service becomes not compliant with the agreed SLA) at execution time by re-binding the composite service.

The work of [Klein *et al.*, 2011], similarly to the work mentioned above, uses a heuristic approach to find near-optimal solution that represents a service composition with the desired QoS. They first use an algorithm based on linear programming which has a low polynomial time complexity to compute an initial solution close to the optimal one of the composition problem. The aim is to influence positively the running time of the heuristic algorithm. The computed initial solution constitutes then the input for the heuristic algorithm based on Hill-Climbing (a type of genetic algorithms) that explores a reduced search space. The result is a reduced time complexity while still achieving near-optimal solutions. Their proposal was validated using simulation-based experiments.

Feng et al. [Feng *et al.*, 2013] proposed also a dynamic approach to service composition taking into account not only initial imposed quality criteria for each involved service but also their service-dependent QoS attributes. Their approach includes in the calculation of the service composition QoS optimal combination, the QoS of what they call “dependent services”. Their claim is that, for example a service provider may give a discount on the execution cost if a related service provided by the same service provider is invoked in the same workflow. This fact, may indeed in some situations led to a better result if dependent-services QoS is considered in the composition. The authors proposed graph-based composition algorithms to compute the optimal QoS of a service combination. They include in their QoS-aware service composition algorithm among others, support to topological and aggregated QoS constraints. The first ones concern the structure of the composition graph and allow the user to decide on the complexity of the generated composition graph. The second ones are useful if the user needs that the generated graph satisfy some QoS constraints in terms of other quality attributes.

The above mentioned approaches are based on "global optimization" that considers the overall QoS attributes and constraints for a composite service and selects the set of services that satisfy global constraints. Another category of approaches for service composition are based on "local selection" which identifies an optimal service candidate and guarantees QoS criteria for each task in a service composition.

In [Sun et Zhao, 2012] the authors propose a combination of the two approaches for QoS-aware service composition. They propose a decomposition-based approach for service composition, in which the utility of a composite service can be computed from the utilities of component services and the constraints of component services can be derived from the constraints of the composite service. The method selects component services independently to optimize the utility of the composite service while meeting the global constraints specified by users. It uses utility functions to evaluate a service by mapping all the QoS attributes into a single aggregated value. Similarly to other works like the one of [Canfora *et al.*, 2008], utility functions are proposed for different composition language control statements such as Sequence, Switch, Loop or Flow to compute the global utility for the composite service by the summation of the utilities of its component services. Their approach combines the advantages of local selection and global optimization, and enables the identification of a composite service that has a near-optimal utility and meets all the global constraints in dramatically reduced computation time.

A different kind of work (from those of the QoS-aware composition problem) is proposed in [Baligand *et al.*, 2006], the authors presented a language named *"QoSL4BP"* and a tool named *"ORQOS"* that enable the architect to specify QoS constraints and some QoS injection mechanisms in Web Service orchestrations. This approach offers a way to integrate quality requirements as usable information at a functional and runtime level, and not at the architectural level. This approach deals with quality requirements as extra information which are exploited at a post-deployment time.

In the field of Web service based business processes engineering methods, few works have been proposed to deal with quality aspects at the design (model) level.

In [Driss *et al.*, 2010] the authors presented an approach to Web service (WS) modeling, discovery and selection. They use an Intentional Service Model (ISM) which they enhance with quality aspects to configure the WS discovery and selection process. The selected services satisfy some quality requirements.

The work of [Azmeh *et al.*, 2011] proposed an approach to Web services composition that satisfy quality requirements. The result of the composition in their work is a sequence of invocations to services that satisfy dynamic quality attributes achieved at runtime (e.g., response time). They do not produce service orchestrations which embody more complex BPEL modeling elements (compared to sequences).

In [Rosenberg *et al.*, 2007] the authors deals with the integration of quality aspects at the modeling level similarly to what we target in our thesis. However, the input of their approach is a WS-CDL Web service choreography annotated with SLAs documents of each partner. The output is a set of BPEL processes and their corresponding WSDL descriptions (one for each partner) integrating the initial mapped QoS requirements of the service choreography. Based on a mapping between WS-CDL and BPEL a transformation is performed. To bring QoS aspects from the choreography to the orchestration layer they mapped SLAs documents to WS-QoS Policies (WS-QoS Policy is their extension to WS-Policy [World Wide Web Consortium, 2007]). WS-QoS Policies are integrated in the "PartnerLink" element of the BPEL process. However, their QoS aspects are requirements specified as policies of the BPEL process and not as its own QoS attributes that should be considered when designing such Web service based business processes.

The authors in [Mukherjee *et al.*, 2008], propose an approach and a tool that operate on already designed concrete BPEL orchestrations to compute their QoS in terms of three run-time measurable quality attributes namely *response time*, *cost*, and *reliability*. To do so, the approach makes use of QoS information of the BPEL orchestration partner services and certain control flow parameters. These includes among others, the probability of selecting branches/events in "if" and "pick" activities, the average number of iteration in loops, and for each “catch” or “catchAll” block the fraction of failures of its associated scope that it successfully intercepts. These attributes are determined from the execution log of the business process. The BPEL process is considered in this approach as an activity graph. A proposed algorithm computes for each node three parameters: i) the probability that a node completes successfully execution, ii) the time of completion of a node and iii) the cost. These three parameters for the root node of the activity graph give reliability, response time and cost respectively for the WS-BPEL composition. The approach allows also a designer to analyze the impact of using some fault tolerance technique on the QoS of the BPEL orchestration, thereby providing a way to achieve high reliability and performance of the designed orchestrations.

Many QoS calculation methods for a composite service like the above mentioned exist [Cardoso *et al.*, 2004 ; Jaeger *et al.*, 2004 ; Ardagna et Pernici, 2007 ; Yu *et al.*, 2008 ; Dumas *et al.*, 2010]. The work proposed in [Zheng *et al.*, 2013] showed some of their limitations and proposed a method to overcome them. The authors showed also that such works proved their usefulness in the selection of component services for composite services.

All the different approaches that we discussed in this section deal with quality aspects of service-based systems. The ultimate goal of these works is to build service-based systems that expose the highest possible quality. However they differ from our work in what follows:

- The majority of these works focus on the selection of services in an already designed abstract service composition or a simple composition in the form of a sequence of invocations. They do not address the actual design of a composition, i.e. how to arrange the various elements that constitute the composition;

- The quality constraints they deal with relate to dynamic quality attributes specific to a given service. These attributes must be measured beforehand (their values must be known, i.e. supplied by service providers). The constraints that we take into accounts (NFRS), relate to attributes that affect the system as a whole. They do not set threshold values for these attributes. They give indications to be considered upstream in the development cycle. Even though in some of these works, they deal with global constraints, these concern the system at runtime, and not its static architecture. Therefore, the nature of the constraints themselves (requirements) is not the same.

That is the context where the work we propose in this thesis comes to its utility. There exist no work in the context of BPEL Web service orchestrations that considers non-functional properties during the design of such service-based systems. Indeed, in our work we propose to deal with quality attributes such as performance, reliability of Web service orchestrations by integrating them at design time. The resulting service orchestrations could then be used in the design of other more complex service orchestrations (compositions) by using their QoS attributes such as response time as supplied by service providers. Web service composition QoS calculation methods [Mukherjee *et al.*, 2008] could then be used to evaluate and/or enhance the run-time QoS attributes of the designed web service orchestration. Such QoS attributes could be published in SLA (Service Level Agreement) documents and used by potential consumers of the service composition.

2.2.4 Assistance to software evolution and impact analysis

Despite the positive sens of the term “evolution”, it may be of harmful consequences when associating it to software. Indeed, software evolution will not lack undesirable consequences if it is not done in a controlled manner. Thus, when trying to meet new requirements by introducing them into the architecture of a software system, or trying to perform maintenance operations by making changes on the latter, we can easily affect other requirements (functional and non-functional) and deviate from the requirements specification initially planned. This evolution process, has as effects to bring up phenomena that have had different names by researchers in the literature, such as : “Software Aging” [Parnas, 1994], “Architectural Erosion” [Merkle, 2010], “Design Erosion”, “Code Decay” [Eick *et al.*, 2001], or “Architectural Degeneration” [Lindvall *et al.*, 2002].

Despite the diversity of these terms, these phenomena agree on the fact that software becomes, after a succession of changes, difficult to maintain and evolve. Therefore, this makes the software unusable. This situation shows the need to make use of methods and techniques to be able to monitor and control changes in the software.

Hochstein et Lindvall [Hochstein et Lindvall, 2005] presented in their work the above mentioned phenomena. They represented them by the term “degeneration” which encompasses them, despite the fact that researchers who have proposed these names were referring to different levels of abstraction namely the design level (Architectural Erosion and Design Erosion) and the implementation level (software aging and Code Decay). The authors discussed various ways to handle the control of software evolution. They introduced various approaches starting from the diagnosis, treatment, research, to prevention following the medical model. Degeneration diagnosis consists in identifying, when the software degenerates. In this phase, we find in the literature architecture recovery solutions that consist in extracting from the source code and other software artifacts, concrete software architecture. These methods are placed in the field of reverse engineering. The idea is to compare the current architecture with the initially planned and see if violations at the architecture level were detected. Most of the mentioned methods are focused on identifying architectural styles and design patterns. The diagnosis can be used as a pretreatment phase to correct the detected problems. Techniques for dealing with degeneration are those of refactoring which consist in restructuring code without changing the system behavior. The degeneration research aim at understanding the evolution to find out how systems degenerate. Among the mentioned techniques, we find those of architectural changes visualization through versions. These approaches apply after real evolution of software, that is, after the application of the desired change has taken place on the software and the architecture of the latter has degenerated. Even though these methods ensure that the changes are reflected correctly on the targeted parts they involve extra costs due to correction operations they bring. It is therefore better to predict changes to be able to avoid them (prevention is better than cure). This, is about taking precautions and prepare the software to potential changes that may occur in the future.

Other authors [Burge et Brown, 2006] have emphasized the importance of *Decision Rationale* (DR) taken during the development process and showed their usefulness in the software system evolution. The authors proposed the SEURAT (Software Engineering Using rationale) system as a maintenance support that allows to exploit DR by giving the possibility of representing, capturing and inferring on the latter in order to detect eventual inconsistencies and indicate design problems. The representation of DR is made by RATSpeak, which is based on the DRL language (Decision Representation Language). RATSpeak is a structuring that includes a set of elements to express DR. Elements defining the DR proposed by the authors are: Requirements, Decision problems, Questions, Alternatives, Arguments, Claims, Assumptions, Argument ontology and Background knowledge. The capture of these elements is done by SEURAT system that offers a tightly integrated tool with Eclipse IDE, making the Rationale documentation process as an integrated part for developers and not separate from the development process. This increases therefore the chances for this very important form of knowledge to be easily saved. Developers can associate, among others, DR with code and specify parts of the code that are implementing them via SEURAT functionalities. The system allows also to infer the knowledge embedded in the code and assist the developer during certain maintenance operations. It offers a number of type inference allowing the control of the knowledge structure (lack of information) and to assess the consistency of a design which resulted from a decisions sequence. SEURAT assists the system users while making maintenance operations, by informing them about the impact of a modification on the choices that have been made. It allows eventually to assist the maintainer of the system during improvement operations, which is to bring new system functionalities through new decisions, by checking that they are consistent with the system's earlier implemented decisions.

In the context of object-oriented software systems evolution management, Steyaert and al. [Steyaert *et al.*, 1996] proposed the concept of reuse contract in a perspective of a good reuse of software artifacts. It is about the control and manage of propagation of changes undertaken on models by modifying the object classes (reusable artifacts) that compose them, by the use of reuse contracts. The latter documents the intentions of both parties, the developer of the artifact and the one that will reuse the artifact. A reuse contract is an interface that contains the specification of a set of methods. Each method is identified by a name with a clause of optional specialization listing only the methods required for the design of the method, and can be of abstract or concrete type. The utility of reuse contracts was shown on abstract classes as reusable artifacts, using inheritance as reuse mechanism. The approach has been exploited on conflicts problems arising from changing super-classes of a class hierarchy of an object model, particularly on the exchange of parent classes by other classes. Reuse contracts are usually implemented by abstract classes and encapsulate some design information on dependencies among methods within them. Therefore, they constitute a source of information to detect inconsistencies in the class hierarchy of an object model. The authors distinguished three basic operators that are applied to reuse contracts: concretization to implement the abstract methods, refinement allowing methods overloading, and extension that allows the addition of new methods. Three additional operators apply on reuse contracts: abstraction, coarsening, and cancellation. They enable to apply the inverse operators of the first ones and derive from the basic contracts associated with parent classes, contracts associated with subclasses. These operators consolidate reuse contracts with additional information on how the classes will be reused.

Tom Mens and Theo D'hondt [Mens et D'Hondt, 2000] proposed later a generalization of reuse contract formalism by integrating it in the UML metamodel. They introduce the concept of evolution contract for design conflicts detection in an object model. The contract specifies the clauses of the provider and the software artifact modifier. The former describe the artifact properties, and the second specifies how the artifact has to be altered, therefore, evolve. Generally, the idea is to add evolution contracts between the elements of an object model representing a given phase. The contract must specify evolution actions that the developer must follow during the modifications. These constitute the basis of the conflict detection process after the model has evolved. The authors used the concept of contract types to restrict operations governing the work of the modifier. Four types have been proposed: Add, Delete, Connection, and Disconnection. An evolution contract is defined as an extension to the UML metamodel. The semantic of a modification (contract type) is specified by OCL rules. The introduction of evolution contracts in the UML meta-modeling level allowed attacking evolution problems on different levels of abstraction, starting from the requirement specification to implementation.

Other works focused on component-based software systems. Also based on the notion of contract, Andreas Rausch [Rausch, 2000] proposed the concept of "Requirements/Assurances" contracts as support to manage the evolution of a component-based system. They were used as part of a development methodology that takes into account the evolution. The establishment of the requirements/assurances contract is done using functions that determine the obligations of each participant (component). They allow at one hand (the REQUIRES function) to calculate from a set of documents, the set of properties (defined by predicates) required by a component (requires from its environment). On the other hand (the ASSURES function) computes the set of properties provided by the component to its environment. Once all the properties specified for each component, the designer explicitly defines the behavioral dependencies between components, by specifying for each component assurances that guarantee other components needs. The formulation of these dependencies constitutes a requirements/assurances contract. The next step is to check at each evolution step (defined as a change in the development documents in a period of time) if the needs of a given component are satisfied by assurances of another component which has undergone an evolution. The formal structures proposed in this work allow describing a component-based or object-based system. The evolution of these descriptions involves changes that might be undesirable in some cases. These latter are detected through contracts requirements/assurances that capture the dependencies between the components of the system and assist the developer during the modifications. However, the provided assistance by these contracts treats the functional aspect (business) and do not care about the impact of the changes on the systems qualities.

Madhavji and Tassé [Madhavji et Tassé, 2003] proposed an evolution policy-driven approach to preserve the qualities and requirements imposed on the software during its evolution. The approach introduces two concepts. The first is a mechanism which enables verification of the violation of certain evolution policy. The second is a contextual framework that constitutes a support for activities which enable the evolution of the software system. The evolution policy is formulated as a constraint in the first order logic, such as the requirement stating that the estimated sum of the number of lines of code added to all system components must not exceed the growth average plus an error percentage. The verification mechanism, which must collect information from a product or process model to improve, assists the developer of the system by ensuring the validity of the constraint and notifying him by the result. The result is analyzed and provided feedback information is presented to the developer. The latter uses this information to decide what action to take and make improvements to the model.

In [Mosser et Blay-Fornarino, 2013] the authors proposed an activity meta-model "ADORE" largely inspired by the BPEL language grammar, that supports business process evolution. The meta-model is based on first-order logic, where process activities are represented as nodes, and relations between activities as edges. The meta-model handles concepts dedicated to support a compositional approach of business process design, with the definition of fragments of processes. Such fragments define additional activities that aim to be integrated into other processes and adequately support their evolution. A weave algorithm was proposed to this end that manage a set of defined actions by the meta-model. However, the proposed approach emphasizes on behavioral evolution of business orchestrations at the model level and do not deal with quality aspect when evolving such business orchestrations.

There are many works proposed in this field that agree on the need to make use of methods and techniques to be able to monitor and control changes in the software, and more particularly those related in our case to quality requirements.

2.2.5 Discussion

In the second part of the state of the art we covered the main areas that are close to our work. The works using ADLs as a means for documenting architectural decisions, work on abstract representations of architecture. These representations employ the notions of components, connectors or configurations to describe architectural decisions. This does not match the types of architecture that we handled in this thesis, which represent service architectures implemented by a language that allows to build concrete architectures (BPEL Web Service Orchestration). Moreover, most of the ADLs are not adapted to document such specific architectures. Indeed, despite the fact that most of them are provided with constraint languages and mechanisms allowing the specification of architectural decisions structural aspects, they are not suited to our purpose which is to describe architectural patterns targeting BPEL's language constructs.

In the third part of the state of the art we discussed the importance of considering architectural design decisions as first-class elements during software architecture design. This category of works is part of the architectural knowledge management approaches that make use of various information sources to capture architectural knowledge, which is comprised of architecture design, design decisions, assumptions, context, and other factors that together shape a software architecture [Breivold *et al.*, 2012]. All the works agree with the fact that architectural decisions and their rationale are the most important form of knowledge to capture during the architectural development process. An explicit representation of this knowledge is necessary for evolving systems and assessing future evolutionary capabilities of a system [Kruchten *et al.*, 2006]. Most of the proposed documentation models provide textual or semi-formal descriptions of architecture design decisions and few works proposed their formalization (more specifically their structural aspects). Even though efforts was made to describe links between design decisions they still lack an explicit formal description of the links between design decisions and quality attributes they affect. Accordingly, in our work we proposed an architecture decision documentation model based on patterns as a kind of architectural design decisions and we explicitly defined in a formal way the links between patterns and their implemented quality attributes. These latter are the rationale behind the choice of a pattern. Additionally, we proposed to formalize not only the structural aspects of a pattern as a design decision but also the way it can be applied to a software architecture. We do not claim the presentation of a new means to document architecture design decisions. Otherwise we believe that our proposal enriches the previous works with fine grained useful information that might be of great interest (as we are going to show it in this thesis) during the design and the evolution of software architectures. Our model is thus complementary to previous efforts in the literature.

In the fourth part of the state of the art we showed a variety of works that support quality considerations during software architecture design. However, the discussed works still have some limitations. They do not sufficiently support reasoning (impact analysis) about the quality consequences of an applied design decision. Besides, they do not offer, or lack sufficient support to explicitly make trade-off analysis between competing design decisions with respect to quality attribute. Some of these works are quite similar to our work in the sense that they use reusable design decisions (attribute primitives and architectural tactics, we use SOA patterns) to address issues pertaining to quality attributes. However, they differ from our work in that they focus on the design stage, while we focus on the design and evolution stages. In addition, we give support to the architect to choose among several possible competing alternatives of a design decision the one that satisfies the best a given quality goal. Besides this, we help the architect in applying the selected design decision (the choice of SOA Patterns) in a semi-automatic fashion, and we give her/him assistance to make impact analysis. To do so, we use a Multi Criteria Decision method (MCDM) in a complementary way with a simulation-based quality-related impact analysis. Note that, our work is a mix from those who exploit the architectural knowledge and those who consider NFRs during software architecture design. Indeed, we make use of an architecture decision documentation model to fulfill NFRs during the design of software architectures.

The fifth part of the state of the art is about achieving quality requirements in service-based systems. In general, in service composition approaches services are selected by considering their QoS attributes. However, these quality attributes are non-functional properties which are mostly measurable at the execution of the service-based system (in our case they are BPEL orchestrations). Indeed, in the service composition problem, individual services are considered as black boxes offering specific functionality and exhibiting QoS attributes. In most cases, these services are differentiated based on those qualities, the one that offers a high quality level is the most desirable. But these qualities, are the consequence of how these services have been designed. Indeed, considering NFRs during the design of services (in our work we talk about BPEL Web services orchestrations) leads to services with eventually hight QoS attributes which are observable/measurable at execution time. This is were the work we propose in this thesis comes to its utility. Indeed, methods that take into account non-functional requirements in engineering Web service orchestrations are required. We considered qualities as static quality attributes during the design of such service architectures. Once integrated in a service orchestration, these qualities are exposed then in future service compositions as dynamic QoS attributes (response time, availability etc.) that could be measured then used (as a differentiator between functionally equivalent services) using different techniques.

The last category gives a preview about assistance and impact analysis approaches applicable during the evolution of software systems. The work we propose in this thesis provides an on-demand assistance method and tools that help architects of BPEL Web service orchestrations in integrating quality requirements when engineering such a kind of architectures.

2.2.6 Summary

This chapter sums up existing works in state of the art in four main categories: AD documentation, software quality documentation, quality achievement in service-based systems and assistance to software evolution. In the first category, we showed different approaches to document design decisions and separated between two categories. Those in architecture descriptions using ADLs and those related to architecture descriptions. The last one highlighted the focus on AD as first-class status in the software development process. The documentation of AD has been proved to bring many benefits. Of these benefits, one most important is a clear vision about the rationale of the AD, which conveys certain quality properties of the architecture.

In the second category we illustrated two ways to document quality. The first one is by using quality models. We have seen that, in general, quality models serve to capture quality characteristics and/or sub-characteristics and then asses/evaluate them on a final software product. The second way, is to anticipate quality assessment, and process the quality aspect by capturing and documenting quality requirements in software architecture. Most of the related research works use architectural styles, architectural patterns, or architectural tactics to satisfy quality requirements in software architectures.

The third category discuss how non-functional properties in service-based systems and more specifically in Web service orchestrations was addressed. Many works was proposed such as languages, composition algorithms, middleware among others to fulfill quality requirement in such systems. Most of the works as the SOA principles impose reuse services with their exhibited QoS attributes to compose more complex service orchestrations with high quality level of QoS attributes that meet the user's quality constraints. We showed the need of engineering methods that consider quality attributes at design level of such service orchestrations that will be used as individual services exposing their own QoS attributes.

The last category of state of the art is about assistance to software evolution. we emphasized the need to make use of methods and techniques to be able to monitor and control changes in the software, and more particularly those related in our case to quality requirements.

The following chapter aims at presenting the first contribution of this thesis, in which we have proposed an architecture decision documentation model. This latter considers SOA patterns as a special kind of design decisions, and represents them as first-class citizens in a software architecture. We base our proposal on the postulate stating that, quality can be implemented through patterns [Zernadji *et al.*, 2014a ; Zernadji *et al.*, 2014b], which can be specified with checkable/processable languages.

CHAPTER 3

Pattern-based documentation model of architecture decisions

In this chapter we present our first contribution which consists in a model to architecture decision documentation based on patterns and two languages that formalizes patterns. We present first an overview on the model in section 3.1 then a detailed presentation in section 3.2 in which we discuss the different concepts of the model. Section 3.3 and 3.4 show the languages we used as a means to formalize patterns.

3.1 General Model

The concept of architecture decision documentation has been firstly introduced in [Tibermacine *et al.*, 2005]. In this thesis, we present an improvement to the old version of this documentation [Tibermacine et Zernadji, 2011]. It defines in a formal way the links between architecture decisions and quality attributes implemented by these decisions. We consider thus architecture decisions, which are entities that can be formalized, as a way to indirectly check automatically quality requirements, which are properties that cannot generally be formalized directly (or are very difficult to formalize[1]).

During the software architectural design process architecture design decisions could be about the use of architectural styles, architectural tactics or patterns, among others, to cope with design problems that the architect may face. Design patterns are an efficient widely used means to communicate and document proven reusable design solutions to recurring problems, as well as the most common used artifact in the software systems development community. Design patterns are sets of predefined design decisions with known functionality and behavior [Jansen et Bosch, 2005]. They provide a common vocabulary and understanding for design principles and a means of documenting software architecture [Gamma *et al.*, 1995]. We consider in this thesis design patterns as the kind of architecture design decisions that achieve quality attributes. The latter are considered hence as the rationale behind these decisions. Design patterns are basically represented by textual descriptions with graphical descriptions accompanying the text. Other approaches use formal descriptions to facilitate analysis and tool support when applying pattern solutions during systems design [Gross et Yu, 2000], and this is why we adopted in our documentation model the formal description. Design patterns find their origins and extensive use in the object-oriented development [Gamma *et al.*, 1995] and gain in popularity and attractiveness even with the emergence of the component-based and service-based development. Therefore, a variety of pattern catalogs (such as [Gamma *et al.*, 1995], [Buschmann *et al.*, 1996], or [Erl, 2009] which is more specific to SOA) have been proposed in the literature to deal with recurrent design problems. Since our work targets service-based systems we were focused on the formalization and use of SOA patterns.

An architecture decision documentation abstracts the links between a given quality attribute and an architecture decision (a pattern) associated to this attribute. Figure 3.1 shows how these links are organized.

[1]By "formalization", we simply mean here the specification of a given artifact in an unambiguous and structured or semi-structured way using a language that can be processed by tools (not using the natural language).

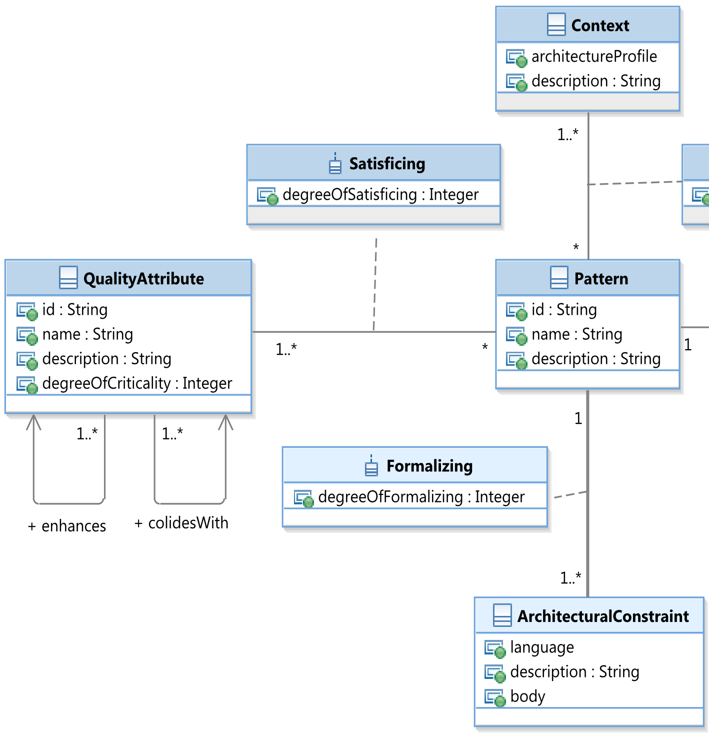


Figure 3.1 : Links between Architecture Decisions and Quality Attributes

3.2 Links between Architecture Design Decisions and Quality Attributes

We associate to a link a degree of satisfaction. A pattern in collaboration with other patterns contribute to the satisfaction of a given quality attribute. Each degree of satisfaction represents a percentage. In the ideal situation (where the developers are confident in the pertinence of their design decisions), the sum of all degrees associated to the same quality attribute (within the same architectural element) would be equal to 100%. For example, a portability quality attribute can be concretized by three different architecture decisions: the choice of the facade service pattern [Erl, 2009][2], the choice of the MVC pattern [Buschmann *et al.*, 1996] and the use of an API. If the developers consider that the two first decisions contribute more, in the concretization of the portability quality attribute, than the third one, because they are critical, they can associate to them high scores (for example 40 % to each pattern) and the last pattern a

[2]This pattern is originally inspired from [Gamma *et al.*, 1995].

lower score (20 % for example). This is done in the same manner as in software requirements engineering where the project manager assigns values like high, medium or low for the technical difficulty of the realization of each requirement or for their functional priority. In our case, we chose to give them numerical values voluntarily because of the complementarity which exists between patterns to reach a quality goal, as illustrated in the example above.

We voluntarily simplify, in this documentation, the specification of patterns as the kind of architecture design decisions. A pattern is thus formalized by two elements, an architectural constraint and an architectural script (See figure 4.2). For the former element, here again, a formalization degree is a percentage associated to the link between a pattern and an architectural constraint. This score represents the extent to which the constraint formalizes the pattern. If we consider that several constraints formalize the same pattern, it is possible for the developer to state how the different constraints share the formalization of the pattern. In some cases, a given constraint may have a degree of formalization more important than others. In the ideal situation (where the developers are sure of the completeness of their formalization), the sum of all degrees associated to the same pattern would be equal to 100%. The constraints written in a given documentation are defined with a predefined constraint language. Architectural constraints are a formal specification of the structural conditions imposed by the pattern and allow the checking of its presence or absence in a service orchestration, therefore, the satisfaction of a quality attribute or not.

A pattern has an application context which is defined by the business or quality constraint as well as the architect preferences. To a pattern we associate a context-suitability degree (a percentage) which is specified and documented at quality integration time (for more details see section 5.2.2) because it depends on the pattern's suitability to a given situation and to the service orchestration. One advantage of the context-suitability degree is to distinguish between pattern variants suitability for a specific situation and a specific service orchestration. Even if the same pattern variant is applied again on the same orchestration it would not have the same impact because the context is frequently not the same. The architect could have for example a preference for a pattern variant over another if it is a matter of price of the delivered service.

The second element is an architectural script that serves as a mean to apply a pattern and its embodiment into a service orchestration. It provides the way it should be applied in an orchestration. A script is composed of basic architecture changes which are a set of parameterized actions that aim to reconfigure the structure of the Web service orchestration. Actions are specified using a scripting language for Web service orchestration reconfiguration called *"WS-BScript"* which is detailed in the next section 3.3.

A quality attribute in this documentation is a non-functional property representing an ISO 9126[3] characteristic or sub-characteristic (Reliability, Maintainability, Portability, ...). It has a degree of criticality (inspired from Kazman's quality attribute scores and Clements' quality attribute priorities [Clements *et al.*, 2002]) which is specified by developers (when expressing their preferences over quality attributes in a service-oriented system project quality plan) and represents the importance of this quality attribute in the architecture. A degree of criticality is represented by a percentage. The sum of all degrees associated to all the quality attributes should be equal to 100%. The technique used to derive the criticality degree values is detailed in section 5.2.2.

Associated to a given architecture decision, a quality attribute can enhance (affect positively) other quality attributes. For example, the choice of the pipeline architecture style targets the maintainability quality attribute, which enhances in this case the performance attribute of the system. Contrarily, a given quality attribute can collide with (affect negatively) other quality attributes. For example, the security quality attribute collides generally with the efficiency attribute. This depends of course on the documented architecture decision and the application context. It is on the responsibility of developers, fully aware of the application's context and the architecture decisions they made, to document these optional parts (the other quality attributes that collide-with or enhance the documented quality attribute) of the pattern-based architecture decision documentation.

A given quality attribute can be tightly- or weakly-coupled to another one. In the first case, if a quality attribute A affects positively another attribute B, if we enhance A, B will B enhanced; and if A is weakened, B will be weakened too. In the second case (weakly-coupled attributes), if A affects positively another attribute B, if we enhance A, B will be enhanced; and if A is weakened, B will not be affected. Inversely, the same thing can be considered, if A affects negatively B. This is illustrated in Figure 3.2.

[3]Software engineering – Product quality – Part 1: Quality model. The International Organization for Standardization Website: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749

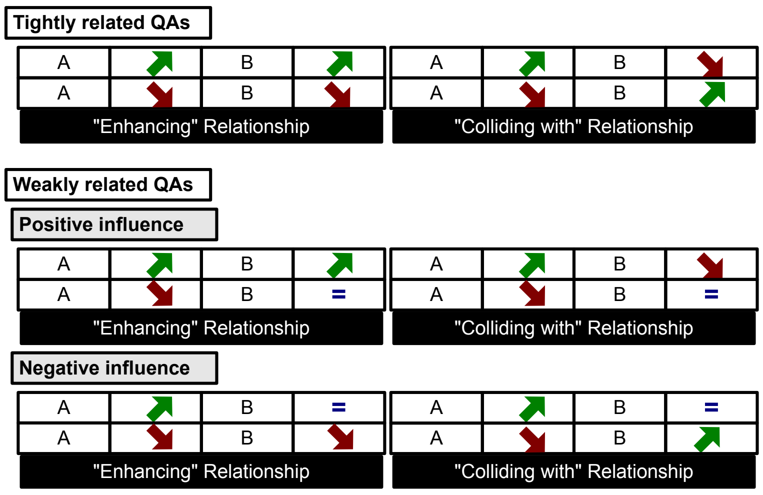


Figure 3.2 : Relationships between Quality Attributes

For example, in a service orchestration, adding an invocation to an encryption service before transmitting information to a remote server is a simple architecture decision taken to enhance the security quality attribute. This makes less efficient the whole orchestration (affects negatively the performance attribute). If we decide in another context, to remove a binding to an authentication service which is invoked before a given business service, this will obviously affect positively the performance quality attribute (there is less time to execute the business service). We conclude here that the two quality attributes, in the two contexts, are tightly coupled.

In another illustrative example, designing a system using the facade service design pattern aiming to enhance its portability affects negatively the reliability quality characteristic (more precisely, the availability sub-characteristic). Indeed, in the presence of a single service providing the business service to clients, if this service crashes, the provided functionality will not be anymore available. Let us suppose now that a given service is provided by a component within a web application in order to abstract details of the different Internet browsers in which the application is executed at the client side (portability purpose). The removal of such a service will not affect in any way the reliability attribute. This is an example of two quality attributes which are weakly coupled.

Between weakly coupled quality attributes, we identified two kinds of relationships. There can be a positive or a negative influence. In the first case (positive influence), it is the enhancement of the first attribute which has an influence on the second one; however in the other case, it is its weakness which produces an effect on the second attribute. This is shown on Figure 3.2

In the current implementation of architecture decision documentation, architecture constraints are specified using the OMG's OCL [OMG, 2010] language. An architecture constraint in this language navigates in a meta-model of BPEL Web service orchestrations, but apply to only one instance of that meta-model (a model which represents a BPEL process). The evaluation of a given constraint tells the developer whether the architecture description conforms to the constraint or not.

In addition to architecture decision documentation, we propose (as an optional feature) to build a catalog of quality attribute relationships. Designing such a catalog consists in :

1. Identifying the quality attributes defined in the quality model of the company
2. Identifying the attributes defined in the quality plan of the software project
3. Building a bi-dimensional table with all the quality attributes (one per line and one per column)
4. Completing progressively the correlation between the quality attributes (on the basis of information gathered from previous projects and the experience of developers)
5. Each time, adapting the table to the service-oriented architecture context

As aforementioned, a pattern is formalized by two languages: i) OCL language coupled with BPEL language meta-model and ii) a scripting language called WS-BScript which is presented in the following section.

3.3 WS-BScript: Web Service BPEL Scripting language

As we mentioned in the previous section, a given quality attribute can be implemented using several patterns. Once the architect choses the pattern that she/he wants to use to implement the desired quality attribute, she/he should apply it inside the Web service orchestration. It could be difficult to the architect to know how exactly each of the existing patterns she/he may choose, has to be applied on the service orchestration. This is especially true when the architect (a novice one) does not know the existing patterns for a given quality attribute. Even if the architect knows the way patterns has to be applied there exist no existing tools/languages that could assist her/him to specify the changes made by the pattern application in the context of BPEL orchestrations. Therefore, it is very useful to have a language/tool which allows to specify these changes then apply them in a semi-automatic way on a web service BPEL orchestrations.

To the best of our knowledge, there is no scripting language which allows the specification of set of actions that reconfigure WS-BPEL web service orchestrations. Therefore, we have developed a voluntarily simplified language called "*WS-BScript*" (for Web Service-BPEL Scripting). WS-BScript is a lightweight DSL that enables the architect to specify primitive changes making possible the reconfiguration of Web service orchestrations. The idea behind WS-BScript is to formalize some SOA patterns in order to apply them as much automatically as possible in the form of reusable design decisions. This language allows the definition of parameterized "scripts". A script is composed of a set of actions like add, wire, and remove, among others. The basic structure of a script is the following:

script apply<PatternName> (<listOfParameters>)
{ <setOfActions> }

A script declares a set of parameters, which represent the scope of the architectural actions. This set identifies BPEL orchestration elements involved in the changes brought by the elementary actions when applying a pattern. They form a super-set for the elements indicated in the architectural area of the quality integration intents, because generally more elements are needed to apply a pattern (these are requested from the architect). These actions are simple statements. We enumerate them in the following listing:

```
(01) add (BpelElement element, BpelElement AttachedParentelement,
int  elementPosition)
(02) add (PartnerLinkElement element, String wsdlFileName)
(03) getPosition (String BpelElementName)
(04) create (BpelElement.Kind)
(05) remove (BpelElement element)
(06) wire (BpelElement element,PartnerLinkElement
element,String PartnerLinkOperationName)
```

```
(07) unwire (BpelElement element,String PartnerLinkElement,
String PartnerLinkOperationName)
(08) ask (String message)
(09) let variableName
(10) variableName = <expression>
(11) for(variableName : OrderedListVar) <actions>
(12) if (<condition>) <action1 or blocOfActions1>
[else <action2 or blocOfActions2>]
(13) query (String OCLExpression)
(14) scriptCall (String scriptName([parameters])
(15) return (BpelElement element)
```

For instance, the first action adds a BPEL element to an orchestration, namely, Invoke, Assign, Receive, and other BPEL process elements (except `PartnerLink` BPEL element which does not require a position in an orchestration). An element is added in a specific position inside a parent element (AttachedParentelement argument) in the orchestration. The second action adds specifically a BPEL `PartnerLink` (`PartnerLinkElement` argument) and links it with a given Web service specified by the `wsdlFileName` parameter. The *"getPosition"* action (Line 03) returns the position of a BPEL element in the orchestration specified by the `BpelElementName` parameter. It is used to identify precisely at what level we should apply a change in the orchestration. The architect may provide the name of the BPEL element, or a qualified name which indicates the path to the element if there are several elements with the same name in the orchestration. Line 04 indicates the *"create"* action which creates a BPEL element instance with some kind (the BPEL element that should be added to an orchestration like, Invoke, Assign, Sequence, Flow, Scope, etc. as defined in the BPEL specification). Line 05 shows the *"remove"* action which eliminates a BPEL process element (except in this case a PartnerLink) from an orchestration such as, a Sequence, an Assign or an Invoke, among others. The *"wire"* action binds a BPEL element to an operation `PartnerLinkOperationName` in a PartnerLink element. The opposite action of *"wire"* is *"unwire"* (Line 07). The *"ask"* action (Line 08) interrupts the execution of the script and waits for some customization values from the architect. It is commonly used in case of complex patterns application, which needs additional parameters that are not fully specified in the quality integration intent. Declaring variables is possible using the *"let"* action (Line 09). A variable can be initialized with an `expression` (Line 10). This expression can be a simple variable, a returned action's value like "getPosition" action, or even a value obtained after evaluation of an arithmetic expression. Variables can be of type integer, string, BPELElement, or Collection type. The *"for"* loop executes repeatedly a given block of actions (Line 11) which should be enclosed between braces. In addition to the *"for"* loop, it is possible to specify *"if-else"* statements (Line12). A condition in an if-else statement is a simple, or a composed boolean expression where we can use conjunction (&&), disjunction (||) and negation (!). The *"query"* action (Line 13) allows to navigate the BPEL meta-model through parameterized OCL [OMG, 2010] expressions and returns the expected result (BPEL elements usually). OCL is used as a navigation language in a complementary way with WS-BScript actions to get BPEL elements but without making any change to the orchestration. Composing patterns is possible through the *"scriptCall"* action (Line 14). It allows calling another pattern script by providing, as an argument, the name of the pattern script we want to call and its arguments. The last action (Line 15) can be used inside a script if the architect wants to return a given BPEL element that can be used by the caller script. Calling the return action terminates the script execution.

The listing below shows a script example of the *"Brokered Authentication Pattern"* [Erl, 2009] which implements the "Access Security" quality attribute. It adds an authentication broker service on top of the invocation sequence in the service orchestration and takes the responsibility for authenticating the client of the service, then issuing a token that the client can use to access the other needed services without the need for the client to have a direct relationship with them. Before executing the script the architect is asked first to indicate its arguments. She/he has to look first in the orchestration for the WSDL file (the `wsdlFileName` parameter) which represents the service. Second, she/he should look for a specific operation (the `partnerLinkOperationName` parameter) in the WSDL file representing the service as well as the process operation name (the `processOperationName` parameter) that should be called to return a response. Finally, she/he looks for the `firstAssign` parameter representing the `Assign` activity after which a call to the authentication broker service has to be made.

```
script applyBrokeredAuthenticationPattern (String firstAssign,
String wsdlFileName, String partnerLinkOperationName,
String processOperationName) {
let aPartnerLink = create (BpelElement.PartnerLink);
add (aPartnerLink, wsdlFileName);
let position = getPosition (firstAssign);
let ocl = "self->closure(eContents().oclAsType(EObject))->select(a|
a.oclIsKindOf(model::BpelType) and a.oclAsType(model::BpelType).name=
'firstAssign')->collect(a:EObject| a.eContainer())->asSet()";
let elem = query (ocl);
let aSequence1 = create (BpelElement.Sequence);
add (aSequence1, elem, position+1);
let aSequence = create (BpelElement.Sequence);
add (aSequence, aSequence1, 0);
let aInvoke= create (BpelElement.Invoke);
add (aInvoke, aSequence, -1);
wire (aInvoke,aPartnerLink,partnerLinkOperationName) ;
let aIf = create (BpelElement.If);
add (aIf, aSequence, -1);
let aCondition = create (BpelElement.Condition);
add (aCondition, aIf, 0);
ask(aCondition);
let aAssign= create (BpelElement.Assign);
add (aAssign, aIf, 0);
let aElse = create(BpelElement.Else);
add (aElse, aIf, -1);
let aSequence2 = create (BpelElement.Sequence);
add (aSequence2, aElse, 0);
let aAssign1= create (BpelElement.Assign);
add (aAssign1, aSequence2, 0);
let aReply= create (BpelElement.Reply);
add (aReply, aSequence2);
wire (aReply,ProcesspartnerLink,processOperationName);
}
```

LISTING 3.1 : Brokered Authentication Pattern application script

Another example is presented in the listing below that shows the architectural script example of the "*Service facade Pattern*" defined using "*WS-BScript*". The application of this pattern as a design decision brings a level of abstraction into the architecture to accommodate potential changes that could occur in a service business logic. Hence, the quality attribute ensured by this architectural pattern is the portability.

```
script applyFacadePattern (List AssignList,
List OperationList,String wsdlFileName) {
let aPartnerLink = create (BpelElement.PartnerLink);
add (aPartnerLink, wsdlFileName);
for (p: OperationList && a: AssignList) {
  let position=getPosition(a);
  let ocl = "self->closure(eContents().oclAsType(EObject))->select(a|
  a.oclIsKindOf(model::BpelType) and a.oclAsType(model::BpelType).name=
  'BpelElementName')->collect(a:EObject| a.eContainer())->asSet()";
  let elem = query (ocl);
  let aInvoke= create (BpelElement.Invoke);
  add (aInvoke, elem, position+1);
13    wire (aInvoke, aPartnerLink, p);
14    let aAssign= create (BpelElement.Assign);
15    add (aAssign, elem,  position+2);
16   }
17  }
```

LISTING 3.2 : Service Facade Pattern application script

This script gives a general way to insert a facade service into a specific position in the Web service orchestration (regardless of the invoked operations number in the service). It adds a partner link element (`PartnerLink` instance) which represents the Web service encompassing the facade (Lines 03 and 04). This one is specified by the `wsdlFileName` parameter. It then repeatedly, through the *"for"* loop (Line 05), looks for the position through the *"getPosition"* action (Line 06) where the architect wants to invoke the service (an operation in the PartnerLink) referenced by an `Assign` activity List[4] element *"a"* (Line 06). The *"getPosition"* action returns the position relatively to a BPEL activity's container. This is why we have to get the container BPEL activity of the `"a"` element so it could be possible to insert a BPEL activity just after it. To do so, in Lines 07-09 through a parameterized OCL expression with a generic format the script gets the container element of the `"a"` activity. The OCL expression accepts two parameters, the name of the `a` activity and the type (`BpelType`) of the activity (namely, Receive, Reply, Invoke, Assign, Sequence, etc. as defined in the BPEL specification). This latter is automatically deduced by the *"WS-BScript"* toolset and injected in the OCL expression. The OCL expression is executed in Line 10 through the *"query"* action and the result is saved. After that, the script adds an `Invoke` activity (the `aInvoke` instance) and binds it to the specified operation List[5] element *"p"* in the previously inserted `PartnerLink` (lines 11, 12, 13). Lines 14 and 15 introduce an `Assign` activity (*"aAssign"*) after the last inserted `Invoke` activity to set variables values.

3.4 SOA Patterns Architecture Constraint Specification

As we have mentioned in the previous section, we used OCL language to specify architectural constraints for SOA patterns. OCL has been chosen because of its simplicity [Briand *et al.*, 2005] and the existence of a good tool support (OCL Toolkit [Dresden.,

[4]The list AssignList represents the Assign activities after which the service operations has to be invoked.

[5]The lists OperationList and AssignList must be ordered lists.

2009], Eclipse MDT/OCL [Foundation, 2009]). OCL (Object Constraint Language) language [OMG, 2010] is the OMG (Object Management Group) standard for expressing constraints on UML models. The goal of this language is to provide developers with a means of specifying conditions for refining the semantics of their models. This constraint language was initially suggested for specifying conditions on functional, not architectural, aspects [Tibermacine, 2014]. However, the OCL language could be used not only at model level but also at the meta-model level which allows the expression of architectural constraints.

After applying a pattern on a service orchestration we have to be sure that its structural aspect is respected when making changes by imposing some architectural constraints. These latter are part of the pattern specification (see Figure 4.2) and serve to verify if an architecture conforms to the pattern or not. Since the pattern implements a quality attribute in the service orchestration, the non-conformance of its structure to the specified constraints implies an altered quality attribute.

In order for this architectural constraints to be reusable artifacts, we build the SOA pattern catalog with parameterized constraints that can be configured then checked when applying a pattern into a service orchestration. The SOA patterns architectural constraints was specified in the context of BPEL web service orchestrations. Therefore, OCL expressions operate on the BPEL meta-model elements. We have to note that not all SOA patterns found in the literature are applicable on the architectural level. About eighty-five patterns for service-based systems that have been described in [Erl, 2009] and the SOA Patterns website[6], about thirty of them [Ton That *et al.*, 2012], each having several variants, can be applied at an architectural level.

Generally speaking, we performed the specification of the architectural constraints in the following steps:

- Reflexion: understanding the role of the pattern, to which problem it responds, and whether it is applicable at an architectural level or not;

- Analysis and specification: identifying the conditions inside a service orchestration that allow to characterize the pattern, then writing the constraints;

[6]http://www.soapatterns.org

- Test and correction: apply the constraints on a specific implementation of a service orchestration and correct the eventual errors.

We give in the following an architectural constraint of the *"Passive Replication Pattern"* brought from our implemented catalog of SOA patterns. We have to note that constraints have been tested[7] on an *"Ecore-specific"* implementation of WS-BPEL meta-model, and that is why *"Ecore"* related details was removed for constraints clarity.

Listing 3.3 shows the architectural constraints of the *"Passive Replication Pattern"* which is one of the three variants of the *"Replication Pattern"* that we have specified in the SOA patterns catalog. This pattern serves the *"Reliability"* quality attribute. Its design solution organizes the service invocations in a hierarchical way, a call to another replicated service is planned only if the first does not answer. The identified structural conditions characterizing this pattern are listed below:

i) The service to be replicated should be wrapped by a `Scope` BPEL activity, this should guarantee to isolate the service that could eventually fail and allow to handle (through a `faultHandlers` BPEL activity) its failing in a `Catch` BPEL activity. This latter is defined inside a `faultHandlers` BPEL activity.

ii) In all the `Catch` activities attached to the `Scope` it should exist only one `Reply` BPEL activity. This latter represents the fault response case of all the replicated services and should be in the last `Catch`.

iii) The number of `Invoke` BPEL activities (representing the calls to the replicated services) where each one is contained in a `Catch`, equals the one of `Catch` activities minus one. The last `Catch` intercepts the failure case of the last replicated service.

iv) The service invocations are organized in a hierarchical way.

```
Context TRS: Process inv:
let scp :Set(Activity)=
self->closure(oclAsType(Activity))->select(a:Activity|a.oclAsType(Scope).name='aScope')
    in
--The service to be replicated should be wrapped by a 'Scope' activity
scp.oclAsType(Scope).activity->exists(b:Activity| b.oclAsType(Invoke).name='
    serviceTobeReplicated')
and
let cth :OrderedSet(Activity)=
```

[7] Tests were held on an enriched version of the NetBeans travel agency application.

```
8  scp->closure(oclAsType(Activity))->select(c:Activity|c.oclIsKindOf(Catch))->asOrderedSet
       () in
9  let rep: Set(Activity)=
10 cth->closure(oclAsType(Activity))->select(c:Activity|c.oclIsKindOf(Reply)) in
11 --In all the 'Catch' elements attached to the 'Scope' it should exist only one 'Reply'.
       This latter represents the fault response case (if any) of all the replicated
       services and should be in the last 'Catch' element
12 rep.oclAsType(Reply)->size()=1 and cth->last()->exists(c:Activity|c.oclIsKindOf(model::
       Reply))
13 and
14 let ink: Set(Activity)=
15 cth->closure(oclAsType(Activity))->select(c:Activity|c.oclIsKindOf(Invoke)) in
16 --The number of 'Invoke' activities equals the one of 'Catch' activities minus one. The
       last 'Catch' intercepts the failure case of the last replicated service if any.
17 ink.oclAsType(Invoke)->size()>=1 and ink.oclAsType(Invoke)->size()= cth.oclAsType(Catch)
       ->size()-1
18 and
19 let fhandlers :OrderedSet(Activity)=
20 scp->closure(oclAsType(Activity))->select(c:Activity| c.oclIsKindOf(FaultHandler))->
       asOrderedSet() in
21 --The service invocations are organized in a hierarchical way
22 if fhandlers->size() > 1 then
23 fhandlers->excluding(fhandlers->last())->forAll(aa,bb:Activity|aa.oclIsKindOf(
       FaultHandler) and
24 aa->exists(bb.oclIsKindOf(FaultHandler))) else false endif
```

LISTING 3.3 : Passive Replication Pattern Architectural constraint

Firstly, this constraint checks that the service invocation to be replicated should be wrapped by a `Scope` which the name is given as a parameter in the constraint (aScope in Line 3). This allows to establish a recovery after failure system by attaching to the `Scope`, a `faultHandlers` element offering the possibility to handle the failure of the service (serviceTobeReplicated in Line 5) in a `Catch` elements. In the second part of the constraint (Line 12), we check that there is only one `Reply` that should be placed at the end of the `Catch` elements hierarchy. The third part of the constraint (see Line 17) ensures that there is `Catch` element which does not encompass a service invocation. Additionally, it ensures the existence of at least one invocation to a replicated service. Finally, the last part checks that the invocations to the replicated services are hierarchically structured since each `faultHandlers` element encompasses another and, each one encompasses a `Catch` (Lines 22- 24). The fact that `Invoke` activities are encompassed by `Catch` activities ensures that a service is called only if its predecessor has failed. The failing of a service throws an exception which is intercepted by a `Catch`, this way we ensure passively the execution of one service at a time.

In listing 3.4 we illustrate the architectural constraint of the *"Exception Shielding Pattern"* [Erl, 2009]. It is a service security pattern which aims to prevent the service environment from malicious attackers by sanitizing unfiltered exception data output before making it available to the consumer. To implement this pattern, we use the "faultHandlers" and the "Scope" BPEL elements which allow to isolate the process part responsible of the exception. This pattern is part of our specified SOA patterns catalog and can be used to implement the *"security"* quality attribute in a service orchestration. Hereafter, the structural conditions characterizing this pattern:

i) The `Invoke` BPEL activity, which invokes the service causing exceptions delivering data to be sanitized should be encompassed in a `Scope` activity.

ii) It should exist only one `Catch` BPEL element (defined in `faultHandlers`) which is attached to the `Scope` activity to intercept exceptions

iii) The `Catch` should encompass the invocation (`Invoke` activity) to the sanitizer service as well as the response message transmitted by the `Reply` activity. The `Invoke` should obviously be made before the `Reply`.

```
let scp :Set(Activity)=
self->closure(oclAsType(Activity))->select(a: Activity|a.oclAsType(Scope).name='Scop') in

--The service that causes exceptions to be sanitized should be wrapped by a 'Scope'
    activity
scp.oclAsType(Scope).activity->exists(b: Activity| b.oclAsType(Invoke).name='
    serviceTobesanitized') and
let cth: Set(Activity)=
scp->closure(oclAsType(Activity))->select(a:Activity|a.oclIsKindOf(Catch)) in
--only one 'Catch' activity attached to the 'Scope' should exist to intercept exceptions
cth.oclAsType(Catch)->size()=1
and
let ch: Set(Activity)=
cth->closure(oclAsType(Activity)) in
ch->exists(a:Activity|a.oclIsKindOf(Sequence) and a.oclAsType(Sequence).name='Seque' and
let sq: Set(Activity)=
a->closure(oclAsType(Activity)) in
--The 'Catch' should contain a call to the sanitizer service (op1) and the sanitized
    response message throught a 'Reply' activity (op2)
sq->exists(b,c: Activity| b.oclAsType(Invoke).name='op1' and c.oclAsType(Reply).name='op2
    ' and
sq->asOrderedSet()->indexOf(b) < sq->asOrderedSet()->indexOf(c)) )
```

LISTING 3.4 : Exception Shielding Pattern Architectural constraint

In the first sub-constraint, we check that the sanitizer service invocation is encompassed by a `Scope` activity. The names of the `Scope` as well as the `Invoke` activities are provided as parameters for the constraint (see Lines 2- 4). The second sub-constraint checks in Line 8 the existence of only one `Catch` element which is sufficient to catch a thrown exception. The unfiltered exception data output processing is performed inside the `Catch` element. This is done through an invocation to the sanitizer service that filters the data, then transmitting the safe data to the service consumer through the `Reply` activity (the operation names of the invoked service and the one to which the reply is intended should be provided to the constraint). These two activities should be inside the `Catch` which is checked in the last sub-constraint (see Line 16). It verifies also that the invocation is performed before the reply (see Line 17).

3.5 Summary

In this chapter we presented a pattern-based documentation model of architecture decisions and two languages for specifying SOA patterns catalogs. The model includes a documentation of the different facets of a pattern: the name, description, guaranteed quality attribute, the pattern instantiation script, and the constraints necessary for the verification of the pattern presence in the service architecture. It explicits formally the existing links between AD (SOA patterns) and the quality it implements. The first language WS-BScript allows to specify the scripts necessary to create a new instance of a given pattern in a SOA architecture defined with BPEL. It is a scripting language with voluntarily simplified primitives to facilitate SOA patterns documentation. The second language is a constraint language based on OCL (Object Constraint Language), a standard OMG (Object Mangagement Group) coupled with BPEL language meta-model. This language allows to specify predicates to verify whether an instance of a SOA pattern exists in an architecture or not and hence the quality it implements.

Through the pattern-based documentation of architecture decisions the architect could build an architecture documentation as a set of design decisions during the architectural design process. Each time a design decision is made in the form of a SOA pattern, it is documented with its rationale which is the quality attribute it implements. In this way, we keep traceability on all design decisions involved in the architectural design process thus, reducing the vaporization phenomenon.

We will show in the next chapter how the pattern-based documentation model of architecture decisions and the SOA patterns catalog are exploited in the quality integration assistance process. We will present SAQIM (Service-oriented Architecture Quality Integration Method). A multi-step process that we have proposed in this thesis to deal with quality requirements integration.

CHAPTER 4

SAQIM: Service-Oriented Architecture Quality Integration Method

This chapter introduces SAQIM, a quality-driven method for assisting architects of BPEL Web service orchestration in achieving their quality goals. We give first during the presentation of our approach a quick overview on the method in section 4.1. Then we detail each of the method steps in the sections that follow.

4.1 The Method at a Glance

A given quality attribute can be implemented using several patterns inside a software architecture. For example, the portability quality attribute can be concretized by three different design patterns: the choice of the *Facade* service pattern, the choice of the *MVC* pattern and the use of abstract APIs. Since the quality attributes that we have to deal with in a service-based system can be listed in an exhaustive way (many quality models exist), their corresponding implementation solutions (SOA patterns) can also be exhaustively listed to some extent (by considering catalogs, such as [Gamma *et al.*, 1995], [Buschmann *et al.*, 1996], or [Erl, 2009] which is more specific to SOA). These quality implementations represent recurrent solutions and seem generic enough to be "formally" specified then processed in a semi-automatic way to be used in different quality integration scenarios.

Even if these implementations could be reused, finding a solution (the well suited to be implemented) among several ones for a given NFR is not a trivial task for the architect. There is about eighty-five patterns for service-based systems that have been described in [Erl, 2009] and the SOA Patterns website[1], about thirty of them [Ton That *et al.*, 2012], each having several variants, can be applied at an architectural level. This makes difficult the decision making for the architect. The reason for that is related to the way each solution concretizes a quality attribute, and what impact it could have on the software architecture. This is especially true, when the architect (a novice one) does not know the existing patterns for a targeted quality attribute or she/he is newly assigned to the software project. For example, the reliability quality attribute can be concretized by the *"Replication pattern"* in different possible ways with different variants namely, *"Naive Replication"*, *"Smart Replication"*, and the *"Passive Replication"*. What is the best possible choice between the three offered solutions? Each of them is suitable for the reliability but one is better than the other depending on the context in which the pattern will be applied. For example, the architect may not be able to figure out the application of the *"Facade Pattern"* for the portability quality attribute, or the use of the *"Exception Shielding Pattern"* to secure her/his orchestration. Besides, even if the architect is assisted by a collection of reusable patterns, it is difficult for her/him to know the way each of the patterns has to be applied on the software architecture. For example, for satisfying the access security quality attribute an architect may not know how to exactly apply the *"Trusted Subsystem Pattern"* in her/his Web service orchestration.

The process we propose in the following sections aims to address the aforementioned problems and helps the architects to: i) find one or several SOA patterns to answer an integration of a quality attribute in their orchestration, ii) choose among several ones the most suitable one, and iii) apply a pattern in their architecture (or cancel an existing pattern if the integration consists in weakening or removing an existing quality attribute).

Figure 4.1 shows the multi-step process that we propose in our work to deal with

[1]http://www.soapatterns.org

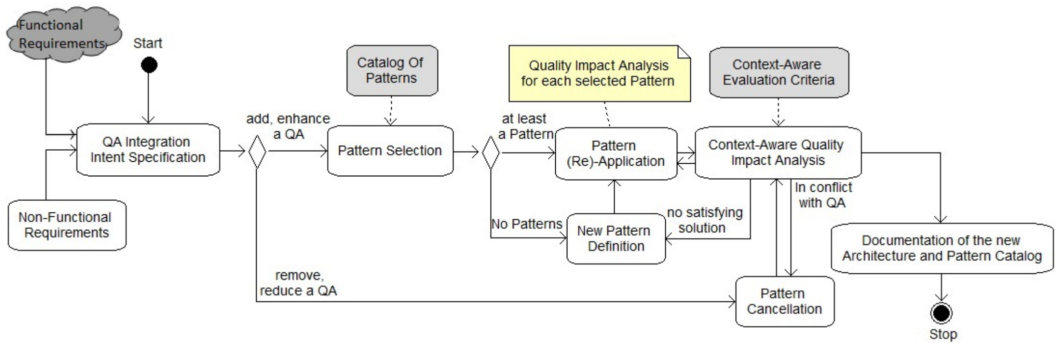


Figure 4.1 : A process for integrating quality requirements in engineering Web service business processes

quality requirements integration. During the process execution, its steps are handled automatically or in a semi-automatic fashion and thus need the architect's involvement. The process that we propose is quality-driven. The architect starts first by formulating an intent (QA intent specification step in Figure 4.1) for achieving a given quality attribute in the Web service orchestration. A template is proposed for this purpose that she/he may complete with some information. Depending on what she/he wants to do against a quality attribute (QA) two options are possible. If the intent is about adding or enhancing a QA the architect is assisted with a collection of proposed SOA patterns (pattern selection step) implementing the targeted QA. Then, the process assists the architect in choosing the pattern that satisfies the best its preferences, by applying first in a semi-automatic fashion the selected patterns (pattern application step) then by reasoning (automatically) on the quality effects of each pattern on the other eventually added qualities (Context-aware quality impact analysis step). If the architect's intent is to remove or weaken a QA, a different type of assistance is provided. The architect is assisted in a semi-automatic way to cancel (pattern cancellation step) the targeted pattern implementing the QA. For the two aforementioned options, at the end of a validated architecture change the architect is invited to document its design decision in a semi-automatic way.

The following sections detail the steps of the method.

4.2 Quality attribute integration intent specification

The architect begins the design usually with functional requirements. We believe that at the design phase some quality attributes are correlated with functional requirements, hence, they have to be processed at the same time with them. For example, in order to integrate the reliability quality attribute, the architect may replicate some service partners. Thus, she/he should look for similar service partners that satisfy the same functional requirement. Consequently, those service partners together allow to achieve the reliability quality attribute.

The architect should first gather the needed information that may help her/him to take decisions correctly while going through the different steps of the process. This information is specified according to a template described in Table 4.1. The architect provides in this template the quality attribute targeted by this integration activity from the quality requirements specification (*i.e.* the architect wants to implement in the service orchestration). We adopt at the top level of our specification the ISO 9126[2] quality model to represent quality attributes as quality characteristics and sub-characteristics. We consider in our work the ISO 9126 quality characteristics mainly as "abstract" quality attributes and sub-characteristics as "concrete" quality attributes which are specializations of the first ones. Some ISO 9126 quality sub-characteristics like "security" are however still considered as "abstract" quality attributes for service-based systems. These sub-characteristics may have several specializations as "concrete" attributes like "Data security" and "Access security". Additionally, the architect should specify where in the orchestration the changes have to be made. Hence, she/he should identify the architectural area that shows the scope of the change. It represents the architectural elements (or sets of these elements) in the BPEL process concerned by the changes. She/he does not specify an exhaustive list of all these elements, but only the main ones. For example, the architect can identify the `Assign` activities in the BPEL process after which `Invoke` activities should be added to integrate `Authentication`. Besides this, the architect has to indicate in her/his intent specification the integration kind by indicating if she/he wants to add (a new), enhance (an existing), weaken, or withdraw (an existing) quality attribute. This will determine the assistance type to provide in the following process steps. Additional information should be specified if the architect

[2]Software engineering – Product quality – Part 1: Quality model. The International Organization for Standardization Website:http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_ detail.htm?csnumber=22749

Table 4.1 : Template for Quality Integration Intent Description

Element	Scope	Description
Quality Attribute Integration	What?	State the quality attribute targeted by the integration activity.
Integration Kind	How?	State if the integration targets to add a new quality attribute, enhance, weaken or withdraw the quality attribute.
Related Quality Attribute	Ultimately what?	If the integration kind is withdrawing or weakening the quality attribute, state here the quality attribute which will be ultimately enhanced or added (left empty otherwise).
Architectural Area	Where?	Indicate where in the orchestration changes will occur.

wants to withdraw or reduce a quality attribute. This is stated in the “Related Quality Attribute” section. Indeed, we argue that each time the architect wants to remove or weaken an existing quality attribute, she/he wants *in fine* to enhance or add another attribute, which is considered here as the “related quality attribute”. For example, when the architect tries to remove “Authentication” for affecting (weakening or removing) “Security”, there is a final goal of enhancing “Performance”. In the other integration kinds (add or enhance), this section is left empty.

The integration intent specification is analyzed, and depending on the integration kind two cases are distinguished. These are detailed in the following subsections.

Quality Integration by Adding or Replacing a Pattern

In this case, the architect wants to enhance (replace the existing pattern implementing the quality attribute by applying one or several other patterns) or add a new quality attribute (apply a new pattern) to the orchestration. Therefore, a collection of patterns is suggested to the architect.

Quality Integration by Removing a Pattern

In other situations the architect may have to remove a quality attribute. For example, she/he may weaken or remove the security quality attribute (authentication). There are no proposed patterns from the catalog here since there is no pattern to apply to the architecture. Rather, a cancellation of the pattern implementing the quality attribute is performed. This cancellation is automatically obtained from the scripts for a pattern application. The pattern application and cancellation will be detailed in sections 4.4 and 4.7.

4.3 Pattern Selection

We consider in this work the existence of an "SOA Pattern Catalog" that we have build, whose structure is detailed later. This pattern catalog is automatically analyzed using the "WS-BScript" toolset and this may result with a collection of patterns related to the targeted quality[3] which are proposed to the architect. The suggested patterns are then applied (Pattern Application step) on the orchestration by the architect in a semi-automatic way by configuring then executing their scripts (using WS-BScript toolset), to evaluate then automatically their impact on the existing qualities (using the WS-BScript toolset). The analysis may also result with no patterns. In this case, the architect is invited to define a new pattern (New Pattern Definition step). The proposed process is based on an "SOA Pattern Catalog", where each pattern is specified according to the model shown in Figure 4.2.

The pattern's specification includes a "name" with a textual description of its role. It includes also the "quality attribute" (The ISO 9126 quality characteristic or sub-characteristic considered as concrete quality attribute) that the pattern implements. Additionally, the pattern contains in its specification an "architectural script" which describes the way it should be applied in the orchestration. This script is composed of basic architecture changes which are a set of parameterized actions that aim to reconfigure the structure of the Web service orchestration. Actions are specified using a scripting language for Web service orchestration reconfiguration called *"WS-BScript"*. The last section in the description of a pattern contains the "architectural constraints", which are a formal specification of the structural conditions imposed by the pattern and allow the checking of its presence or absence in the orchestration.

Existing SOA patterns are usually presented in the literature following a functional organization (patterns for reliable messaging, patterns for atomic distributed service transactions, etc.). This does not answer our needs in this work where we would like to

[3]As stated previously, a quality attribute may be implemented by applying several patterns in different ways.

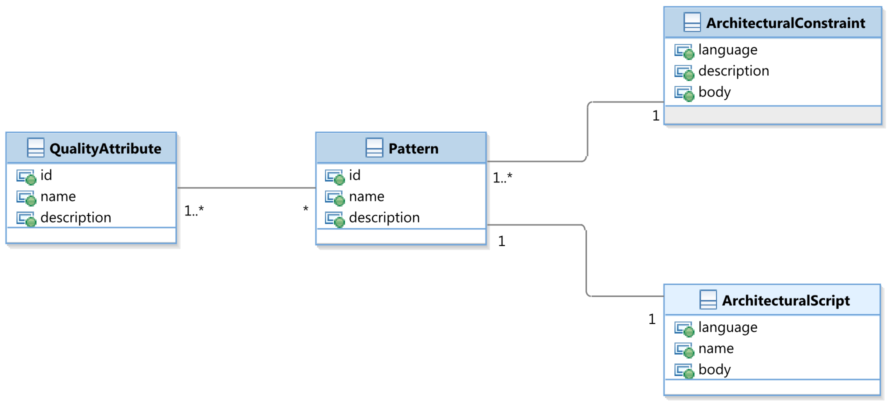


Figure 4.2 : Pattern Specification

propose a pattern that concretizes a given quality attribute. Consequently, we organize the patterns catalog based on the qualities they implement.

The “SOA Pattern Catalog” is an important artifact in SAQIM. It is partially built before any use of SAQIM. It is then enriched, according to the model presented previously, each time a new pattern is used in the engineering of a given service orchestration using SAQIM. There are two roles associated to this catalog: i) a catalog administrator, whose responsibility is to feed the catalog with new pattern specifications (scripts, constraints, ...), and ii) a catalog user (an architect of a given orchestration), who will not directly manage the catalog, but will just see SAQIM suggesting the application of patterns retrieved from the catalog (or executing cancellation scripts processed from the catalog). As indicated previously, in some cases, the architect has the possibility to feed the catalog with new pattern specifications. In this case, the architect will play temporarily the role of an administrator. It is true that the responsibility of the architect is to design the system, but the fact that she/he is able to enrich the catalog will enable future instantiations of the same pattern, either in the same orchestration or in other orchestrations by benefiting from the automated support provided by SAQIM.

4.4 Pattern Application

This is an important step in the process where the selected SOA patterns are applied on a targeted Web service orchestration by means of some scripts, which specify simple architectural changes expressed with a Web service orchestration scripting language (WS-BScript).

In this step of the process, the architect will apply one or several predefined[4] scripts (issued from the catalog of patterns) on her/his orchestration. For this end, the architect has to configure the scripts she/he wants to apply by initializing their parameters first and then by customizing them on the fly (through `ask` actions).

In the current implementation, the selected patterns are instantiated (from the pattern catalog which contains the description of patterns) and then applied on the Web service orchestration. It produces at last a new Web service orchestration. The architect is informed about the script application progress by displaying information on the embodied elements composing a pattern instance.

A registry of patterns is created in this step which references all the instances[5] used to build a service orchestration, each of which has a unique identifier. The registry exists during the quality integration assistance process; it is destroyed at the end of the process. Compared to the architecture documentation, it contains all the patterns that have been proposed to the architect for selection while the documentation contains only those chosen and applied on the service orchestration. At evolution time, where future changes may occur on the service orchestration, the registry could be restored from the architecture documentation to assist architects.

Listing 4.1 below shows a script example of the *Trusted Subsystem Pattern* [Erl, 2009] which implements the "Access Security" quality attribute. It prevents from unauthorized access to the resources of a service by malicious attackers. It adds an authentication service on top of the invocation sequence in the orchestration to secure the service from direct access to the databases.

Before executing the script the architect is asked first to indicate its arguments. She/he has to give first the WSDL file (the `wsdlFileName` parameter) which represents the service. Second, she/he should indicate a specific operation (the `partnerLinkOperationName` parameter) in the WSDL file representing the service. Then, she/he should state an operation name in the BPEL process (the `ProcessOperationName`) to which a reply is prerformed in case of an authentication failure. Finally, the `BpelElementName` parameter representing the BPEL activity after which a call to the authentication service has to be made is provided by the architect.

```
script applyTrustedSubsystemPattern (String BpelElementName,
```

[4]The patterns scripts are already specified in the patterns catalog, the architect has just to apply them.

[5]Several instances of the same pattern may exist in an orchestration.

```
2  String wsdlFileName, String partnerLinkOperationName,
3  String ProcessOperationName){
4  let position = getPosition (BpelElementName);
5  let ocl = "self->closure(eContents().oclAsType(EObject))->select(a|
6  a.oclIsKindOf(model::BpelType) and a.oclAsType(model::BpelType).name=
7  'BpelElementName')->collect(a:EObject| a.eContainer())->asSet()";
8  let elem = query (ocl);
9  let aAssign = create (BpelElement.Assign);
10 add (aAssign, elem, position+1);
11 let aSequence = create (BpelElement.Sequence);
12 add (aSequence, elem, position+2);
13 let aPartnerLink = create (BpelElement.PartnerLink);
14 add (aPartnerLink, wsdlFileName);
15 let aInvoke = create (BpelElement.Invoke);
16 add (aInvoke, aSequence, 0);
17 wire (aInvoke, aPartnerLink, partnerLinkOperationName) ;
18 let aIf = create (BpelElement.If);
19 add (aIf, aSequence, -1);
20 let aCondition = create (BpelElement.Condition);
21 add (aCondition, aIf, 0);
22 ask(aCondition);
23 let aAssign1 = create (BpelElement.Assign);
24 add (aAssign1, aIf, 0);
25 let aElse = create(BpelElement.Else);
26 add (aElse, aIf, -1);
27 let aSequence1 = create (BpelElement.Sequence);
28 add (aSequence1, aElse, 0);
29 let aAssign2 = create (BpelElement.Assign);
30 add (aAssign2, aSequence1, 0);
31 let aReply = create (BpelElement.Reply);
32 add (aReply, aSequence1, -1);
33 wire (aReply, ProcesspartnerLink, ProcessOperationName);
34 }
```

LISTING 4.1 : Trusted Subsystem Pattern application script

The script starts first by looking through the *"getPosition"* action (Line 04) for the position of the BPEL activity (`BpelElementName` parameter) representing the architectural area after which the architect would like to apply the change. The *"getPosition"* action returns the position relatively to a BPEL activity's container. This is why we have to get the container BPEL activity of the `BpelElementName` activity so it could be possible to insert a BPEL activity just after it. To do so, in Lines 05-07 through a parameterized OCL expression with a generic format the script gets the container element of the `BpelElementName` activity. The OCL expression accepts two parameters, the name of the `BpelElementName` activity and the type (`BpelType`) of the activity (namely, Receive, Reply, Invoke, Assign, Sequence, etc. as defined in the BPEL specification). This latter is automatically deduced by the *"WS-BScript"* toolset and injected in the OCL expression. The OCL expression format given in the script example navigates in an *Ecore* implemetation of the BPEL meta-model (see Figure 2.3).

The OCL expression is executed in Line 08 through the *"query"* action and the result is saved. Then, the script adds in Lines 09 and 10 an `Assign` activity for variables setting before adding a `Sequence` activity (Lines 11 and 12) inside which the remaining BPEL activities composing the pattern will be inserted. We should note that in the *"add"* action, the "0" value means an insertion at the beginning of the container activity and the "-1" value means an insertion at the end, otherwise the architect has to specify the exact position. After that, the script adds a `partnerLink` BPEL activity to the targeted orchestration (Lines 13 and 14). Just after, an `Invoke` activity is added to the orchestration (Lines 15 and 16), having as attribute the `partnerLinkOperationName` parameter which indicates the operation to invoke in the previously inserted PartnerLink. Line 17 binds the `Invoke` activity to the PartnerLink. The script adds If-Else BPEL elements (Lines 18-21, 25 and 26) to specify the case of success, or failure of the authentication for which a `Reply` is intended (Lines 31-33) to answer the consumer a non-granted access. The script interrupts the execution through the *"ask"* action, asks on the fly for additional customization parameters and assists the architect to set the condition of the `If` element (Line 22). This script is executed on the BPEL description of the Web service orchestration which results in a new Web service orchestration implementing the security quality characteristic.

4.5 Quality Impact Analysis

In SAQIM we provide a mean that proposes to architects a collection of SOA patterns implementing the desired quality attributes as well as a mean to apply them in the service orchestration. Now, we need to know what are the consequences of integrating each pattern into the service orchestration. What impact may a pattern have on the other embodied patterns. Hence, what impact may the satisfaction of a quality attribute may have on the other already achieved quality attributes in the service orchestration. Moreover, in the presence of various alternative patterns for the same intended quality attribute what it the most satisfactory one to the architects preferences. To answer the aforementioned questions a quality impact analysis is required in the quality integration assistance process. We proposed for this end a quality-oriented impact analysis process that assists architects with a quality-oriented change assistance algorithm and a recommendation system of SOA patterns satisfying quality attributes for service orchestrations. The quality impact analysis process is detailed in chapter 5.

4.6 New Patterns Definition

It is up to the architect to validate her/his choice of a specific pattern or to reject it. If the architect is not satisfied with any of the proposed patterns, then she/he can define new patterns (specialization of existing patterns, for example), which she/he is asked to document according to the proposed specification (Figure 4.2). They will be considered as new reusable architecture design decisions that could potentially be applied on some architecture descriptions in the future. The architect plays in this step the role of the catalog administrator to feed the catalog with the newly defined pattern.

After that, the architect is redirected to the "Patterns Application" step to simulate the effect of the new catalogued pattern. This is an important transition backward in the process, especially if the architect who catalogued the pattern is not the one who chose the patterns that are implemented in the architecture, and therefore, potentially did not know them. Consequently, she/he does not know the impact of the new pattern application on the other implemented qualities in the architecture. Hence, returning back to the "Patterns Application" step is necessary to assist the architect.

4.7 Pattern Cancellation

As we have mentioned in Section 4.2, the architect may want to remove or weaken a given quality attribute. In this case, the process execution takes another path, as illustrated in Figure 4.1. The process goes through the pattern cancellation step where an elimination of the concerned pattern is performed. This is done by deducing the opposite effect of the pattern's architectural actions, hence avoiding to the architect the burden of doing it manually or specifying the cancellation script. The generated cancellation script is then executed on the Web service orchestration. The generation of a cancellation script is handled automatically (by the "*WS-BScript*" toolset) following a bottom-up approach starting by the last action in the script and going up to the first one, by respecting some specific rules which are enumerated hereafter: 1) keep the script parameters specified in the original script; 2) maintain the loops and if-else statements as they are; 3) ignore the “ask”, “return”, “create”, “query” actions; 4) replace the “add” action by the “remove” action, and the “wire” action by the “unwire” action; 5) replace a script call by its corresponding cancellation script. 6) Replace the remove (BpelElement element) action by two primitives:

```
i) let element= create(BpelElement.Kind), and
ii) add (BpelElement element, BpelElement AttachedParentelement, int  elementPosition)
```

In the following listing 4.2 we show the cancellation script of the “Trusted Subsystem” pattern, whose application script is given in Section 4.4:

```
1  script cancelTrustedSubsystemPattern (String BpelElementName,
2  String wsdlFileName, String partnerLinkOperationName,
3  String processOperationName) {
4   unwire (aReply, ProcesspartnerLink,processOperationName);
5   remove (aReply);
6   remove (aAssign2);
7   remove (aSequence1);
8   remove (aElse);
9   remove (aAssign1);
10  remove (aCondition);
11  remove (aIf);
12  unwire (aInvoke,aPartnerLink,partnerLinkOperationName) ;
13  remove (aInvoke);
14  remove (aPartnerLink);
15  remove (aSequence);
16  remove (aAssign);
17 }
```

LISTING 4.2 : Trusted Subsystem pattern cancellation script

The script presented above cancels the application of the “Trusted Subsystem” pattern by reversing its actions from the last one to the first one. Line 01 unbinds the “Reply” activity from the “PartnerLink” before removing it (Line 02). Similarly, the other BPEL elements are removed in the opposite order they were added (Rule 4 stated above). The script parameters remain unchanged (rule 1).

The cancellation of a pattern from a service orchestration involves the following steps: i) looking for all the pattern instances the architect wants to remove from the pattern registry, and listing them to the architect, then ii) the architect should choose manually the pattern instance to cancel; iii) if the pattern cancellation script has been already generated, apply the script, otherwise, generate the script and add it to the registry then apply it, and finally iv) manage pattern intersections (handled automatically by*"WS-Bscript" toolset*) by showing to the architect the BPEL elements pertaining to other pattern(s) that could be eventually removed when applying the script. If it is the case, it is up to the architect to validate the change or not.

The consequences of removing the pattern instance implementing a quality attribute are reported to the architect by the quality impact analysis (See chapter 5) using the quality oriented assistance service (architectural constraints checking), and it is the architect's responsibility to validate the change and hence documenting the new architecture, or repeat again the different steps of the process for a new architecture decision.

4.8 Documentation of the New Architecture

In this step, the chosen pattern is applied to the orchestration and added in the architecture decision documentation as a new design decision. This documentation contains all design decisions (SOA patterns) that was made to build the architecture. In addition, the architect has to complete a part of this documentation, namely the formalization degree of the pattern, and also the related qualities of the quality attribute. The criticality degree of the quality attribute the pattern implements, and the satisfaction degree of the pattern for the quality attribute are automatically added to the documentation by the *"WS-BScript toolset"*. This information is necessary for the futur quality integrations especially in the patterns selection process (quality impact analysis step). We show below an excerpt of the Travel Reservation System TRS (See Figure 6.1) system's architecture documentation. Its architecture documentation is presented in a synthetic way (in order to not be too verbose with its original XML-based description) in the listing below:

```
Architecture-Documentation :
1. Architecture-Tactic :
    This tactic ensures the Access Security quality requirement by using
    a Trusted subsystem pattern
    - Quality-Attribute name="Access Security" degreeOfCriticality="34,8"
    - Related-Quality name="Availability" relationship="Enhances"
                               relationType="weak" influence="negative"
    - Architecture-Decision name="Trusted subsystem pattern"
                                         degreeOfSatisficing="18,15"
                                         degreeOfContext-suitability="72,6"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="90"

2. Architecture-Tactic :
    This tactic ensures the Data Security quality requirement by using
    a Exception Shielding pattern
    - Quality-Attribute name="Data Security" degreeOfCriticality="24,6"
    - Related-Quality name="Portability" relationship="Enhances"
                           relationType="weak" influence="negative"
    - Architecture-Decision name="Exception Shielding pattern"
                                     degreeOfSatisficing="75"
                                     degreeOfContext-suitability="80"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="90"

3. Architecture-Tactic :
    This tactic guarantees the Portability quality requirement by using
    a Service facade pattern
    - Quality-Attribute name="Portability" degreeOfCriticality="5,8"
    - Related-Quality name="Performance" relationship="CollidesWith"
                           relationType="tight"
    - Architecture-Decision name="Service facade pattern"
                                    degreeOfSatisficing="90"
                                    degreeOfContext-suitability="95"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="80"
```

The architecture documentation contains three architectural tactics. They document the links between architectural decisions (SOA patterns) and their corresponding quality attributes (QA1, QA2, QA4 in Table 6.1). In this documentation we can see among others the different relations between quality attributes (`Related-Quality` element in the listing above). For example, in the third tactic, the `Related-Quality` element shows that the portability and performance quality attributes are colliding and are tightly coupled.

4.9 Summary

In this chapter we have presented SAQIM, a method for quality integration in service orchestrations which relies in a complementary way on the service-oriented impact analysis process and the documentation model detailed respectively in chapter 5 and chapter 3. We presented the different steps of the method starting from an "intent" specification which expresses a request to satisfy a quality attribute in the service orchestration to its achievement using SOA patterns. We showed how the method and its accompanying "toolset" assists the architect and leads to concrete architecture changes with minimal negative effect on the overall service orchestration quality attributes.

The next chapter details another contribution in this thesis which is the quality-oriented impact analysis process. This process is used in a complementary way with SAQIM (in the quality impact analysis step) and offers valuable assistance in the quality integration process.

CHAPTER 5

Quality-oriented impact analysis process

In this chapter we expose the quality impact analysis process. We present first in section 5.1 a micro-process of architecture evolution and where in this process, the quality-oriented change assistance takes place. The latter, is detailed in section 5.2.

5.1 A Micro-Process of Architecture Evolution

Figure 5.1 shows a simple micro-process of service-oriented architecture evolution[1]. In this process, the triggers for requesting architecture evolution can be either new business requirements (for perfective evolution), bug reports (for corrective evolution) or quality enhancement (for perfective, adaptive or preventive evolution). Then the developer has to go through multiple steps, ranging from architecture comprehension to the proposition of a new architecture.

Among these steps, the developer performs some testing to check if there is a

[1]This micro-process addresses software evolution in general, and not service-oriented architectures in particular. Adaptations to this specific context are detailed later.

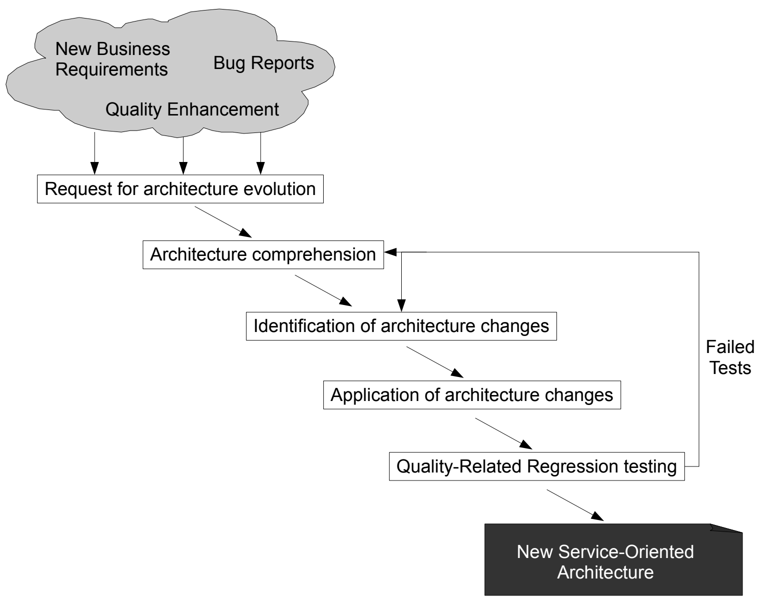

New Business Requirements
Bug Reports
Quality Enhancement
Request for architecture evolution
Architecture comprehension
Identification of architecture changes
Application of architecture changes
Quality-Related Regression testing
Failed Tests
New Service-Oriented Architecture

Figure 5.1 : A Micro-Process of Architecture Evolution

"clean" progression (verify if the additional services, operations or activities work correctly) and no regression (existing features are not negatively impacted by the additions). We address exclusively quality-related regression testing. In practice there are few works that dealt with this aspect by proposing some automatic support. Even with the existence of such approaches, if some tests fail, the developer iterates (eventually many times) to fix the problems. She/He is asked to look for the architecture changes to be applied, and sometimes she/he is led to the step of "Architecture comprehension".

The proposed quality impact analysis aims at assisting this process by notifying the developer on-the-fly if there are some architecture changes that affect quality requirements. This is illustrated in Figure 5.2. The quality impact analysis is an important step which is used by SAQIM (See section 4.5).

The approach introduces two concepts: an architecture documentation model (bottom left of Figure 5.2) and a quality-oriented architecture change assistance that uses the architecture documentation. The first concept encompasses the set of SOA patterns (as design decisions) already instantiated and used to build the service architecture. For each pattern we specified following the model introduced in section 3.2, its degree of satisfaction for the quality attribute it implements, the relationships of this latter with the other already introduced qualities, its degree of formalization, among

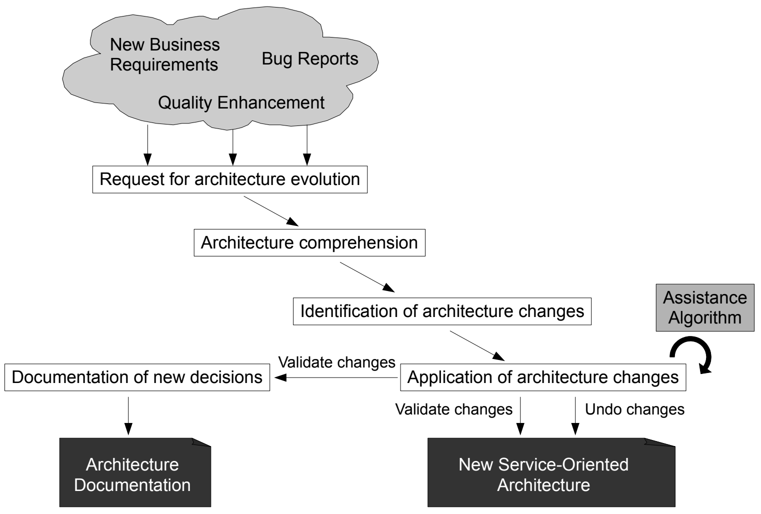

New Business Requirements
Bug Reports
Quality Enhancement
Request for architecture evolution
Architecture comprehension
Identification of architecture changes
Assistance Algorithm
Validate changes
Documentation of new decisions
Application of architecture changes
Validate changes
Undo changes
Architecture Documentation
New Service-Oriented Architecture

Figure 5.2 : The Proposed Micro-Process of Architecture Evolution

others.

The architecture change assistance is used when developers apply changes on an architecture to notify them with the possible impact of their changes on quality requirements. Since the process (SAQIM) that we propose to integrate a quality requirement in a service architecture is pattern-based, a change is often the instantiation of a pattern in the service architecture. Then, it is the developer's responsibility to validate or undo changes. If changes are validated the developer is asked to document the new decisions taken while evolving the service-oriented architecture. Another role of the quality-oriented architecture change assistance is to help architects in choosing the most satisfactory pattern to apply in her/his service orchestration since she/he may be faced with many alternatives for the same quality attribute. The quality-oriented architecture change assistance is detailed in the following section.

5.2 Quality-Oriented Architecture Change Assistance

There are two key elements that are used in this step: i) the use of a quality-oriented assistance service that helps in diagnosing the consequences of any applied pattern on the other implemented qualities, and ii) the use of a Multi-Criteria Decision Making (MCDM) method, named "WSM" [Fishburn, 1967] (Weighted Sum Model), to evaluate a number of SOA pattern alternatives and to help the architect to select the most satisfactory pattern in a quality requirement integration step.

The algorithm 1 shows the behavior of this step. It is composed of several functions. The algorithm is launched after the selected patterns are applied on the service orchestration in the “Pattern Application” step of SAQIM (See section 4.4). Each pattern is applied on an instance of the targeted service orchestration.

During the quality integration process, the information encapsulated in the architecture documentation (See section 4.8) is exploited by the assistance algorithm in order to assist architects. The main purpose is to drive software architecture change to a situation where the quality integration intent is satisfied and the existing quality is minimally affected. This is done in three main steps: i) constraint evaluation and data collection; ii) pattern ranking, and iii) result reporting. The algorithm starts first by looking for the architecture documentation associated to the service orchestration which has been changed. Then, in the first step the algorithm checks (`checkArchitecturalConstraint (..)` function) each constraint (Line 21) in the documentation (by calling a function which is detailed in section 5.2.1) and collects a part of the necessary data to partially configure the ranking system (WSM). In the second step, the ranking system collects first the remaining data required to complete its configuration then computes and returns the ranking scores of all the patterns (ADs) in a descending order (Lines 22- 27). It is obvious that there is no need for the ranking system if there is only one selected pattern. In the last step, the results are reported (Line 28) to the architects to allow her/him to choose a pattern from the selected ones. After that, the developer is asked to pinpoint the architecture decision (a pattern) and the quality attribute associated to the changes, if any (Lines 30- 33). At last, if the changes generate a new architecture decision (the choice of a pattern), the algorithm adds (`addNewArchitecturalTactic (..)` function) to the documentation the couple composed of this new decision associated to its quality attribute, which is called an architectural tactic (Line 34). In addition the algorithm tries to infer the quality attributes affected by this new tactic (Line 36).

We note here that the patterns (as design decisions) are previously documented by the architect according to the model introduced in section 3.2. This model introduces some fine-grained information (see Figure 3.1) namely, the criticality degree of a quality attribute which represents its importance in the architecture, the formalization degree, which represents the extent to which some checkable constraints (present

Algorithm 1: Quality-Oriented Change Assistance

```
begin
    let AE := Architectural Element;
    // a service orchestration;
    and AD := Architectural Decision;
    and AC := Architectural Constraint;
    and QA := Quality Attribute;
    and AT := Architecture Tactic;
    // a couple composed of a QA and an AD;
    and Doc:= architecture documentation associated to changed AE;
    and wsmParams:= { };
    // an empty list of WSM system parameters (Aij, Wj);
    and rankedPatterns:= { } ;
    // an empty list of pairs (AD, score);
    and affectedQAs:= { } ;
    Function main(){
    begin
        after Pattern Application {
        foreach (AT in Doc) do
            QA := QA in AT;
            AD := AD in AT;
            $checkArchitecturalConstraint(AD)$;
            let A2j= ask for the context-suitability decision criterion value;
            $wsmParams := wsmParams + (A2j, W2)$;
            let score := runWsmSystem( );
            rankedPatterns := rankedPatterns + (AD,score);
        end
        $sort(rankedPatterns)$;
        $displayResults()$;
        $UpdateArchDocumentation()$ ;
        let newAD := ask for AD associated to the new architecture, if any;
        if $newAD \neq null$ then
            let newQA := ask for the QA associated to newAD;
        end
        $addNewArchitecturalTactic(newAD, newQA)$;
        }
        $checkAffectedQAs()$;
    end
    }
end
```

in the documentation) formalize the pattern, and the satisfaction degree, which represents the degree to which a design pattern contributes to satisfy a quality attribute. The documentation is enriched with a context-suitability degree, which is specified and documented at quality integration time because it depends on the pattern's suitability to a given situation and to the orchestration. This degree cannot be reused in different service orchestrations. It can however be reused in the future evolutions of the same service orchestration.

Then, the developer is asked to validate the new architecture through the `UpdateArchDocumentation (..)` function (Line 29 in algorithm 1) fully aware with the possible consequences of her/his changes, or to undo changes (Line 3 in the the `UpdateArchDocumentation (..)` function detailed in algorithm 2).

Algorithm 2: Update Architecture Documentation

```
1  Function UpdateArchDocumentation(AD)
2  begin
3      ask to validate the new architecture or undo changes ;
4      if new architecture maintained then
5          affectedQAs := affectedQAs + QA + QA_Relationships(QA, "enhances",
             "tight");
6          warn "Architecture documentation will be changed ...";
7          Doc := Doc - AT(AD,QA);
8          ask to review satisficing degrees of ATs related to QA_Relationships(QA,
             "enhances", "tight");
9          ask to review Non-Functional Requirements specification;
10     end
11 end
```

In this last case, the architecture documentation should be updated by the algorithm (this is another important role of this assistance algorithm). The affected decisions and their associated quality attributes are removed from the documentation (Line 7). The developer is at last asked to review the degrees in the documentation, as some tactics are removed. In addition, she/he is invited to review the non-functional (or quality) requirements specification.

The function `addNewArchitecturalTactic(...)` (detailed in the algorithm 3) creates a new architectural tactic and adds it to the documentation. Before that, if the quality attribute has been voluntarily added by the developer, it is removed from the set of affected quality attributes (Line 5). Else this attribute is considered as a new quality and a checking is performed to alert the developer of the other qualities that are possibly affected by this attribute (Line 7). At last, the algorithm asks the developer to change the quality requirements specification.

The last function (algorithm 4) just recalls to the developer that there still remain some affected quality attributes, if any. The developer is asked to review the architecture documentation and the quality requirements specification.

Algorithm 3: Add Architectural Tactic

```
Function addNewArchitecturalTactic(AD,QA)
begin
    newAT := new AT(AD,QA) ;
    if QA is in affectedQAs then
        affectedQAs := affectedQAs - QA;
    else
        warn "Other QAs may be in conflict with "+QA+": " + QA_Relationships
          (QA,"collidesWith","both");
    end
    warn "Architecture documentation will be changed ...";
    Doc := Doc + newAT; ask to change Non-Functional Requirements
      specification;
end
```

Algorithm 4: Check Affected Qualities

```
Function checkAffectedQAs()
begin
    if $affectedQAs \neq null$ then
        foreach (QA in affectedQAs) do
            warn QA + "is still affected by your changes";
            ask to review satisficing degrees of ATs implying QA ;
        end
        ask to change Non-Functional Requirements specification ;
    end
end
```

The overall goal of this algorithm is threefold. First, it assists developers during architecture evolution with information about the impact of their changes on architecture design decisions and on quality attributes. Second, it helps architects faced to a variety of patterns implementing the same quality attribute to choose the one that best satisfies their needs. Third, it helps to maintain the documentation of non-functional (or quality) requirements up-to-date in a semi-automatic fashion. This can be observed in updates made automatically on the documentation, requests to review satisficing degrees of the affected quality attributes, and requests to change or review NFRs specification.

5.2.1 Quality-Oriented Assistance Service

The first element of the quality-related impact analysis step is an assistance service which aims to notify the architect of the consequences of the applied pattern on the other qualities. It indicates what are the related qualities that may be altered when applying the pattern which implements the new quality attribute. This assistance is mainly based on the evaluation of some OCL constraints that we used to specify SOA patterns parameterized architectural constraints for Web service orchestrations. These constraints are defined using OCL and navigate in a metamodel of BPEL.

Algorithm 5: Architectural Constraints Checking

```
1  Function checkArchitecturalConstraint($AD$)
2  begin
3      let result := check AC ;
4      if $result == false$ then
5          $AffectedQAsNotifier(AD)$;
6          warn "Other QAs may be in conflict with "+QA+": ";
7          + QA_Relationships (QA,"collidesWith","both");
8          $wsmParams := wsmParams + (A1j, W1)$;
9      end
10 end
```

The function `checkArchitecturalConstraint(..)` detailed in the algorithm 5, checks the constraints associated to a given architecture decision (a pattern) received as an argument. It starts by checking the constraint expressions associated to the decision. If the checking does not succeed for a given constraint, a set of warnings are displayed to the architect by the `AffectedQAsNotifier (..)` function (Line 5). The displayed information includes the architecture decision, the exact architectural element impacted by the change, the degree of formalization of the decision, the quality attribute, its degree of satisficing and its criticality degree (Lines 8 and 9 in the algorithm 6). In addition, it shows to the developer the list of quality attributes which are eventually impacted by the change (Line 10). For doing so, it uses the recorded information in the architecture documentation namely, the “related-quality” attribute (See section 4.8 for un example). It notifies also the developer when adding a quality attribute to the service orchestration, about the quality attributes which are indirectly impacted (*i.e.* the quality attribute that its constraint still hold and is related to the added quality attribute). For example, when adding the portability quality attribute, the change may not invalidate the constraints formalizing the performance quality attribute implemented in the service orchestration but, this latter could be in a conflicting conceptual relationship with the portability.

Algorithm 6: Affected Qualities Notifier

```
1  Function AffectedQAsNotifier(AD)
2  begin
3      AE := AE in the context of AC;
4      QA := QA associated to AD;
5      warn "The following architecture decision " +AD+" is affected.";
6      warn "This concerns the architectural element: "+AE;
7      warn "The affected architecture decision is formalized by the constraint up to "
         + degreeOfFormalization (AD,AC)+ "%" ;
8      warn "The affected architecture decision is satisficing "+QA + " up to "
         +degreeOfSatisficing(AD,QA)+"%";
9      warn "The degree of criticality of this QA is: "+ degreeOfCriticality(QA);
10     warn "Other QAs may be affected. This concerns: " + QA_Relationships
         (QA,"enhances", "tight") ;
11 end
```

Finally, the `checkArchitecturalConstraint(..)` function collects from the architecture documentation for each applied pattern a part of the necessary data for the ranking system (WSM) configuration (Line 8). This data is the criticality degree value (C_1) of the directly impacted quality attributes[2].

5.2.2 Weighted Sum Model for Patterns ranking

The “WSM” method is the second key element of the quality impact analysis step and is used only when the “Pattern Selection” step (See section 4.3) results in a collection

[2]The change produced by the application of an architecture decision (a SOA pattern) may impact several quality attributes

of patterns for a targeted quality attribute. Its goal is to give a ranking on the selected patterns to choose the best alternative (having the highest WSM score).

Concerning this element, the MCDM problem we want to solve can be expressed as following: "what is the pattern that impacts the less the most important quality attributes, and is the most suitable to the architect preferences (context suitability, e.g., price, applicability related conditions, etc.)?" We have formulated the MCDM problem as follows:

- Alternatives are some selected patterns we want to classify;
- Decision criteria are defined as follows:
    1. Criticality of the impacted quality attribute (C_1);
    2. Context-Suitability of the pattern (C_2).

The "WSM" is considered as one of the most widely used methods for its simplicity [Triantaphyllou *et al.*, 1999]. If there are M alternatives and N criteria, then the best alternative (pattern) is the one that satisfies (in the maximization case) the following formula [Fishburn, 1967]:

$$A_i^{wsm} = max_i \sum_{j=1}^{N} a_{ij} w_j, for \qquad i = 1, 2, 3, ..., M. \tag{5.1}$$

$\sum_{j=1}^{N} w_j = 1$ and $w_j > 0, j = 1, ..., N$

a_{ij} is the value of an alternative "i" (pattern) in terms of a decision criterion "j". Weights represent the importance of each criterion according to the architect's preferences in the quality integration process.

In our approach, we choose the "Pairwise Comparison" method introduced in the "AHP" (Analytic hierarchy Process) method [Saaty, 1980] to derive the data. AHP is highly mature, has a shallow learning curve (simple to learn within a reasonable length of time), uses quantitative measures and has clear-cut steps [Mead, 2006]. "Pairwise comparison" is known to have a good theoretical foundation and is easy for decision makers to understand [Triantaphyllou *et al.*, 1999]. In this approach the decision maker has to express her/his opinion about the value of one single pairwise comparison at a time by using the scale proposed by Saaty [Saaty, 1980] depicted in Table 5.1.

Table 5.1 : Scale of relative importance

Intensity of importance	**Definition**
1	Equal importance.
3	Weak importance of one over another.
5	Essential or strong importance.
7	Demonstrated importance.
9	Absolute importance.
2, 4, 6, 8	Intermediate values between the two adjacent judgments.
Reciprocals of above nonzero	If activity i has one of the above nonzero numbers assigned to it when compared with activity j, then j has the reciprocal value when compared with i.

Pairwise comparisons are represented in a decision matrix. In our MCDM problem the data consist in the criteria weights (W_j) as well as the criteria values themselves (C_1 and C_2). This data constitute the parameters for the WSM ranking system. Weights should be derived in advance by the patterns catalog administrator, that means before using the proposed method. The criticality degree values (C_1) of the quality attributes defined in the adopted quality model[3] are derived by developers when expressing their preferences over quality attributes. The data (C_1) creation is done in the context of a service orchestration which may make a quality attribute more desirable than another (for example, security may be more advantaged than portability). Additionally, the criticality degree values should be also prepared beforehand and should be available to be used in the proposed method. They are automatically extracted after executing the scripts of the patterns being evaluated, because it depends on the criticality degree of the impacted quality attributes. If there is only one impacted quality attribute we take its criticality degree, if there are many, we take the sum of the criticality degrees of the impacted quality attributes.

Figure 5.3 shows an example of a decision matrix which represents the architect's preferences for the quality attributes defined in a service-oriented system project quality plan. An entry in the matrix, labeled a_{ij}, indicates how much the criticality for quality "i" is higher (or lower) than that for quality "j". Each quality has a value of "1" when compared to itself. Figure 5.4 shows the derived values for C_1[4].

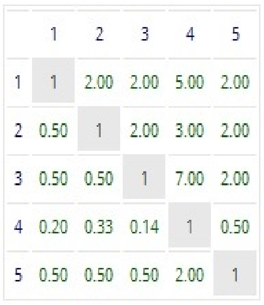
[3]A company may define its quality attributes based on the developers experience.

[4]We used an online AHP priority calculator to calculate weights based on pairwise comparisons: http://bpmsg.com/academic/ahp_calc.php

In AHP, the pairwise comparisons in a decision matrix are considered to be consistent if the corresponding *"consistency ratio (CR)"* is less than 10% [Saaty, 1980]. The CR derived for the values in the below decision matrix is 5.4%. Finally, the context-suitability values (C_2) are derived when patterns are selected to be applied on the service orchestration. The data is specified before executing the pattern script because it is not documented yet since it is a context-dependent value and should be specified at design time.

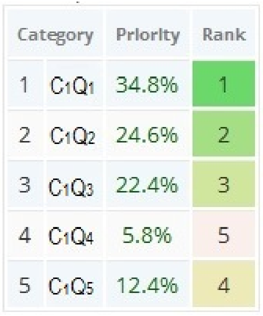
- An example of the weights vector: W_1= 0.750, W_2= 0.250 respectively for C_1 and C_2 (prioritizing criteria weights show that the architects give more importance to C_1).

- The criticality degree weights vector (Figure 5.4) for the five quality attributes defined in the project quality plan: C_1Q_1= 0.348, C_1Q_2= 0.246, C_1Q_3= 0.224, C_1Q_4= 0.058, C_1Q_5= 0.124 respectively for QA1, QA2, QA3, QA4 and QA5.



	1	2	3	4	5
1	1	2.00	2.00	5.00	2.00
2	0.50	1	2.00	3.00	2.00
3	0.50	0.50	1	7.00	2.00
4	0.20	0.33	0.14	1	0.50
5	0.50	0.50	0.50	2.00	1

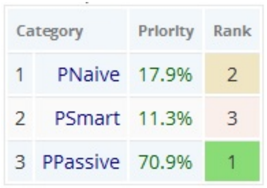
Figure 5.3 : Decision Matrix.



	Category	Priority	Rank
1	C_1Q_1	34.8%	1
2	C_1Q_2	24.6%	2
3	C_1Q_3	22.4%	3
4	C_1Q_4	5.8%	5
5	C_1Q_5	12.4%	4

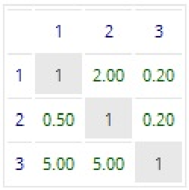
Figure 5.4 : Weights for C1.

Hereinafter, an example in the selection process when dealing with the reliability quality attribute (QA3). The proposed solution (Pattern Selection step) for ensuring Reliability (QA3) was the *"Replication Pattern"* with its three different variants namely, the *"Naive Replication (RP_1)"*, the *"Smart Replication (RP_2)"*, and the *"Passive Replication (RP_3)"*. The Replication pattern considers multiple implementations (as backups) of a service actively used, thus representing a point of failure in the system architecture. The architect decides to design a rescue system by the use of a backup service for the Airline service, which is used sequentially. A call to the second service is planned only if the first does not answer. The architect prefers the last variant of the pattern since its design solution organizes the service invocations in a hierarchical way, while the first two variants plan parallel invocations (the first waits for the first answer then continues, the second waits for all answers then picks the best one). Therefore, she/he gives a score (Pattern Application step) for the Context-Suitability Degree which is more important than the other patterns. Another advantage of the last criterion (context-suitability) is to distinguish between pattern variants suitability for a specific situation and a specific orchestration. Even if the same pattern variant is applied again on the same orchestration it would not have the same impact because the context is frequently not the same. The architect could have a preference for the "Smart Replication" if it is a matter of price of the delivered service. The architects proceed by configuring the WSM system with Context-Suitability criterion values (C2) of each pattern based on its preferences. Figures 5.5 and 5.6 show the derived values for C_2. For example, in Figure 5.6, in the row 1 column 2 of the matrix the architect slightly favors the "Naive Replication" over the "Smart Replication", hence she/he puts her/his judgment value "2". In the row 1 column 3 of the matrix, when comparing the "Naive Replication" with the "Passive Replication" the architect strongly advantages the latter, hence she/he puts the reciprocal value of "5" (0.20). The architect has just to fill (in case of manually doing the calculation) one half of the matrix (the upper half). The other half represents the reciprocal values.



	Category	Priority	Rank
1	PNaive	17.9%	2
2	PSmart	11.3%	3
3	PPassive	70.9%	1

Figure 5.5 : Weights of C2 for the replication pattern.



	1	2	3
1	1	2.00	0.20
2	0.50	1	0.20
3	5.00	5.00	1

Figure 5.6 : Decision Matrix.

When the WSM method is applied on the previous data, the scores of the three alternatives are:

- PNaive (WSM score)= 0* (0.750) + 0.179* (0.250) = 0,04475
- PSmart (WSM score)= 0* (0.750) + 0.113* (0.250) = 0,02825
- PPasive (WSM score)= 0* (0.750)+ 0.709* (0.250) = 0,17725

The notification report shows a higher score of the *"Passive Replication (RP_3)"* with no impacted related qualities (those directly impacted and their related quality attributes in the orchestration), followed by RP_1 then RP_2. The results that yield the application of the WSM method are considered as the satisfaction degrees of each applied pattern for a quality attribute. Note that values between parentheses are weights. All variants have had no impact on any implemented quality attribute in the orchestration, which explains the "0" values for the first criterion (C_1).

5.3 Summary

We proposed first in this chapter a micro-process of architecture evolution and we have showed where in this process the quality-related analysis is involved. We presented a method which makes operational the pattern-based documentation model of design decisions through its use during architecture evolution, and an algorithm which implements the supervision of architecture evolution. This supervision aims at deducing on-the-fly the possible impact of a given architectural change on design decisions (SAO patterns) and consequently identify the affected quality requirements. Additionally, it provides a recommendation system for SOA patterns that allows architects to choose the alternative that satisfies the best its preferences. The latter is based on the weighted sum model and uses pairwise comparison technique of the AHP method as a weighting technique. Here again, the recommendation system exploit the information (the criticality degree of a quality attribute and the context-suitability degree of a pattern) found in the documentation model to give a ranking on the selected patterns to choose the best alternative (having the highest WSM score).

Our approach has been applied on a specific kind of software architectures, which are service-oriented ones. A concrete implementation of this kind of software architectures has been considered in our work, which are BPEL Web service orchestrations.

In the following chapter, we present an evaluation of our contributions through a number of simulation-based experiments to show the usefulness of the illustrated kind of evolution assistance.

CHAPTER 6

Evaluation

To evaluate our contributions that are respectively presented in chapter 3, chapter 4 and chapter 5, we first show in detail through a case study how the proposed approaches are used in practice. Then, to show the benefit of our proposals we conducted some experimentations on real-word web service orchestrations using real data.

6.1 SAQIM in Practice

The Web service orchestration, implemented by a BPEL process, that we use as an illustrative and running example here represents a Travel Reservation Service (TRS) of a travel agency. The TRS service[1] is an example of real-life service for travel organization. This system enables the users to plan and book trips in the Web. For this end, the service interacts with four service partners namely a flight reservation service, hotel reservation service, train reservation service, and a car rental service.

As in any software development the design of the TRS business process is based on requirements which consist of functional requirements (*FR*), non-functional require-

[1]Released with NetBeans from Oracle Website.

ments (*NFR*), and technical requirements[2]. The functional requirements include the main functionality in a travel agency reservation system which are in our example the four service partners.

In addition to the functional requirements, the TRS system has initially the following non-functional requirements:

- **NFR1**: Service consumers are granted access only if they are authenticated, and no direct access to the backend resources of the service is allowed. The transmitted data must not be intercepted by unauthorized service consumers.
- **NFR2**: The TRS system must not deliver any sensitive data that may be used by malicious users which could compromise the integrity of the overall service.
- **NFR3**: The TRS system must ensure that the flight reservation service should be available during the reservation time (8:00 AM-6:00 PM) in the working days. If the service does not respond within 60 seconds the TRS system should notify the system administrator.

The three NFRs are integrated into the orchestration at design time. After the NFRs specification analysis the architect identified the first quality attribute she/he wants to implement in the web service orchestration from NFR1, which is the "access security" (QA1). The second and the third quality attributes, "data security" (QA2) and "reliability" (QA3) are identified respectively from NFR2 and NFR3.

At the beginning, the architect designing this orchestration starts by looking (and/or developing) for candidate service description interfaces that offer the needed functionality of the aforementioned services of the TRS system. After getting the identified service description interfaces, she/he integrates them into the web service orchestration and invokes them in the desired logic.

We will see now some evolution scenarios which target quality requirements of this service-oriented system, in which two additional NFRs emerged after a certain period of time in the system's lifetime.

[2]We are not interested in our work in this last kind of requirements.

Table 6.1 : Embodied Patterns and their achieved Quality attributes

Pattern	Trusted Sub-system (1)	Exception Shielding (2)	Replication (3)	Service Facade (4)	Brokered Authen-tication (5)
Quality Attribute	Access Security (QA1)	Data Security (QA2)	Reliability (QA3)	Portability (QA4)	Access Security (QA1)

After a period of time, the architects realized that the service needs to access additional databases (of different airline companies) having different formats, which resulted in a portability (labelled QA4) quality evolution.

A long time after creating the system, the company providing these services has expanded significantly, and therefore more users requested the TRS system. Consequently, the architect observed that the performance (QA5) of the overall service (TRS Service) has decreased due to a subsequent increasing number of user requests, which imposed managing a large amount of data. As the amount of concurrent usage increases, so does the amount of the generated responses, leading to increased resource consumption of the entire service.

The two new additional NFRs are:

- **NFR4**: The TRS system should be able to support new data formats required by the service partners and therefore, compensates their behavior modifications so that the consumers are not impacted.

- **NFR5**: The TRS system processes and validates a large amount of data. To increase performance, the transmission of unnecessary data to the consumers should be avoided.

In order to satisfy the previous NFRs, several SOA patterns have been applied by the architect in the TRS business process. Figure 6.1 shows the distribution of patterns in this business process. Its design involved the use of five patterns that are introduced incrementally into the orchestration[3]. Table 6.1 enumerates each of the embodied patterns and its achieved quality attribute.

[3]We presented four (4) of the patterns in Figure 6.1 for space limitation.

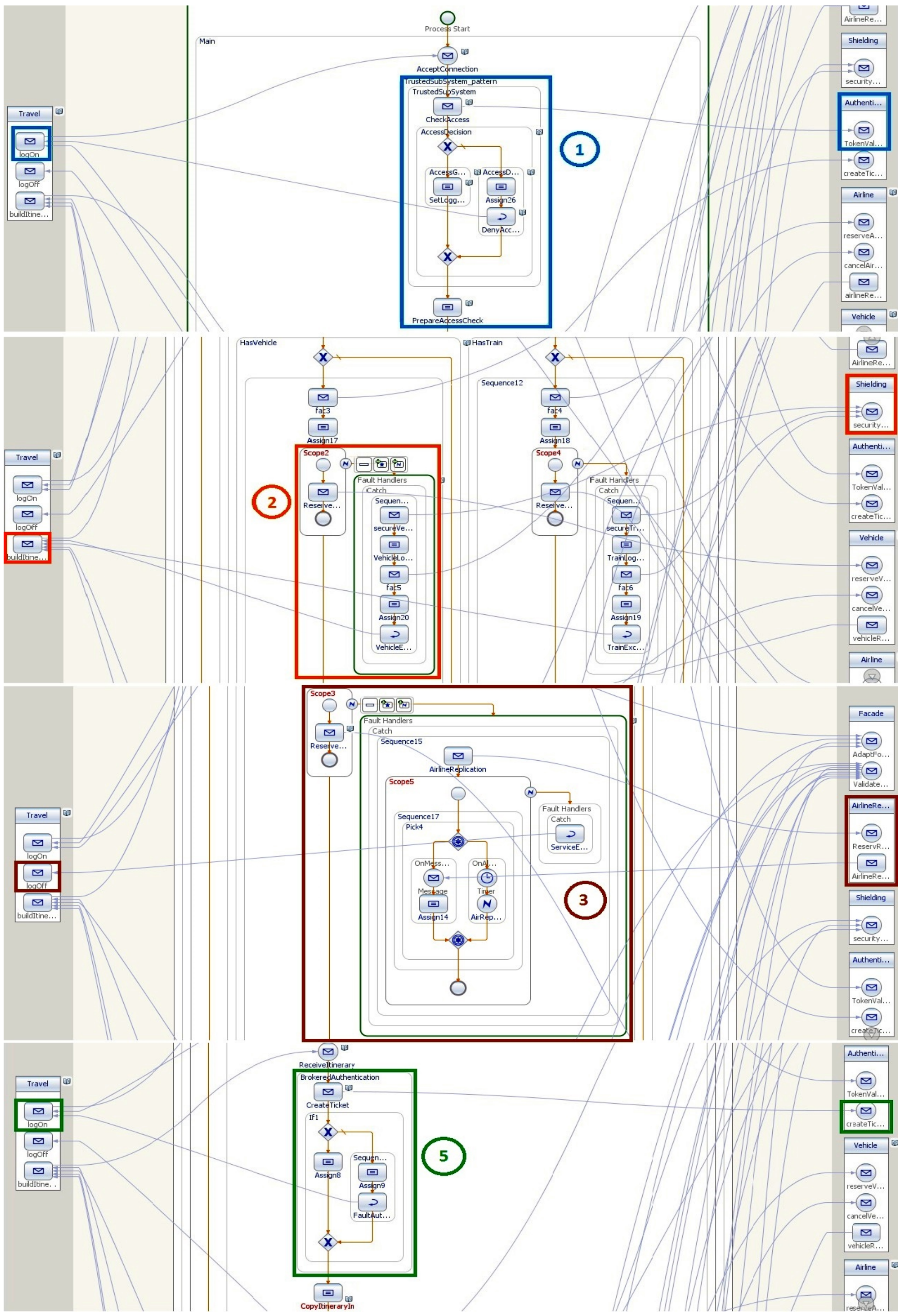


Figure 6.1 : An excerpt of the TRS Business process showing the distribution of the embodied patterns

Let us see now how quality integration intents are handled to address quality requirements through the different steps of the proposed method (SAQIM). In the following we present one possible scenario for the TRS service orchestration design. The first pattern (see Table 6.1) is embodied into the orchestration when the architect wants to achieve the "access security" quality attribute (QA1). Table 6.2 depicts the specification of the first quality integration intent (QII1). It shows that the quality integration targets the "access security" quality attribute (QA1) which will be eventually added to the orchestration. The architect specified the BPEL elements being involved in the change which is shown in the "Architectural Area" section of Table 6.2. This shows that the change will occur after a `Receive` BPEL activity in the TRS system named `AcceptConnection`. In the next step (Pattern Selection step), an analysis of the patterns catalog is performed to extract patterns implementing the targeted quality attribute (access security).

Three patterns that serve QA1 were proposed to the architect which are: 1) *Trusted Subsystem Pattern* (P_1); 2) *Brokered Authentication Pattern* (P_2), and 3) *Direct Authentication Pattern* (P_3).

Pattern P_1 prevents from unauthorized access to the resources of the TRS service by malicious attackers. It adds an authentication service on top of the invocation sequence in the orchestration to secure the service from direct access to the backend resources. The service authenticates the clients then uses its own credentials to access the backend resources. Pattern P_2 adds an authentication broker service in the invocation sequence of an orchestration and takes the responsibility for authenticating the client of the service. Then, it issues a token that the client can use to access the other required services it composes without the need for the client to have a direct relationship with them. The third pattern (P_3) adds a service which requires the client services to present credentials for direct authentication to access the functionality of the service. This pattern may involve contrarily to pattern P_2, a bandwidth consumption. This is especially true if the client needs access to several services each one requiring a direct authentication which may compromise the system's performance.

The architect proceeds then in the "Patterns Application" step, by configuring the WSM system with Context-Suitability criterion values (C_2) of each pattern based on its preferences. The resulting weights for the criterion obtained by the application of the AHP pairwise comparisons method are ranked as follows: (P_1)= 0.726, (P_2)= 0.172,

Table 6.2 : Intent Specification for QII1

Integration Quality Attribute	Security/Access Security
Integration Kind	Add
Related Quality Attribute	
Architectural Area	after AcceptConnection: Receive

(P_3)= 0.102; The result shows that the architect's most preferable pattern for the current context is (P_1), followed by (P_2) and (P_3). This comes from the architect's need to secure the backend resources of the TRS system. The architect then configures the script of each one of the proposed patterns by providing the needed arguments and apply them.

- The weights vector: W_1= 0.750, W_2= 0.250 respectively for C_1 and C_2
- The criticality degree weights vector for (Figure 5.4 in section 5.2.2) the five quality attributes defined in the software project quality plan: C_1Q_1= 0.348, C_1Q_2= 0.246, C_1Q_3= 0.224, C_1Q_4= 0.058, C_1Q_5= 0.124 respectively for QA1, QA2, QA3, QA4 and QA5.

When the WSM method is applied on the previous data, the scores of the three alternatives are[4]:

- Trusted Subsystem(WSM score)= 0* (0.750) + 0.726* (0.250) = 0.1815
- Brokered Authentication(WSM score)= 0* (0.750) + 0.172* (0.250) = 0.043
- Direct Authentication(WSM score)= 0* (0.750)+ 0.102* (0.250) = 0.0255

After the execution of the "Quality Impact Analysis" step, the notification report shows a high score of the "Trusted Subsystem Pattern P_2" with no impacted related qualities (those directly impacted and their related quality attributes in the orchestration), followed by P_2 then P_3. The results that yield the application of the WSM method are considered as the satisfaction degrees of each applied pattern for a quality attribute. Note that values between parentheses are weights. It is worth mentioning that, since it is the first quality attribute to integrate in the service orchestration, all

[4]We omitted here the representation of the decision matrix.

Table 6.3 : Intent Specification for QII2

Integration Quality Attribute	Security/Data security
Integration Kind	Add
Related Quality Attribute	
Architectural Area	before ReserveVehicle, ReserveTrain, ReserveHotel : Invoke

variants have had no impact on any implemented quality attribute in the orchestration, which explains the "0" values for the first criterion (C_1). Consequently, the architect documents the selected pattern as a new design decision in the architecture decision documentation from the one hand, and commits the resulted new Web service orchestration from the other hand.

The architect continues then the service orchestration design based on its functional requirements where multiple BPEL elements are added to implement the TRS service orchestration business logic. After that, to answer QII2 for achieving "data security" quality attribute (QA2) the architect uses the proposed process and the *"Exception Shielding Pattern"* is suggested. The architect decides to secure the `Train`, `Vehicle` and the `Hotel` services. This is done by replacing the unsafe data handling mechanism with one which is safe through the use of a specialized exception shielding service. The latter is invoked when an exception occurs.

She/he configures then the pattern's script (Patterns Application step) for each of the three services by providing the list of parameters as required in the script. Then, she/he applies each pattern instance. After that, the "Quality impact analysis" step reports to the architect that no quality attributes were impacted by the change brought by the pattern instances application (no constraints formalizing the first pattern "Trusted Subsystem Pattern" were violated). Therefore, the architect documents the newly integrated design decision in the architecture decision documentation and saves the new orchestration.

The proposed solution for ensuring Reliability (QA3) was the *"Replication Pattern"* with its three different variants namely, the *"Naive Replication (RP_1)"*, the *"Smart Replication (RP_2)"*, and the *"Passive Replication (RP_3)"*. The specification of its quality integration intent (QII3) is shown in Table 6.4.

The selection process for the reliability quality attribute (QA3) was explained in section 5.2.2 of chapter 5.

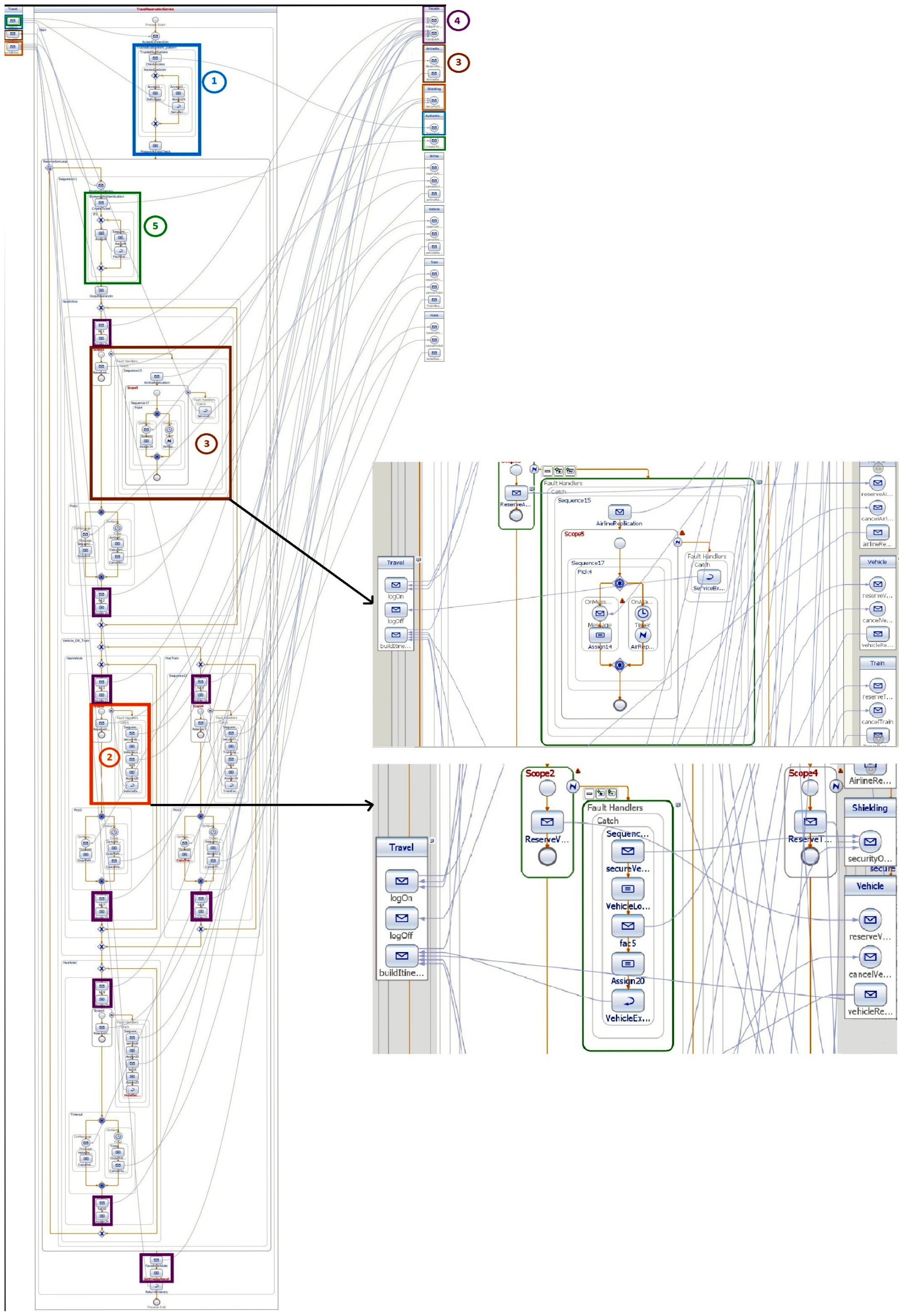


Figure 6.2 : TRS Business process

Table 6.4 : Intent Specification for QII3

Integration Quality Attribute	Reliability
Integration Kind	Add
Related Quality Attribute	
Architectural Area	before ReserveAirline: Invoke

To satisfy the fourth identified quality attribute (QA4) the process proposed the *"Service Facade Pattern"*. The pattern's script[5] application involved several BPEL elements integration (that was not present in the orchestration) at different positions (specified as script arguments) in the orchestration. Several instances of the pattern were applied on the orchestration which are shown by the pattern (4) in Figure 6.2. The application of this pattern brings a level of abstraction into the architecture to accommodate potential changes that could occur in the service business logic. It ensures the adaptation between the message format used by the TRS service and the data format handled by the service partners. Also, it validates the data received from the service partners. Here again there are no impacted related qualities (those directly impacted and their related quality attributes in the orchestration). Consequently, the architect documents the proposed pattern as a new design decision in the architecture decision documentation and saves the resulted new Web service orchestration.

Always in the context of the same intent of embodying the "access security" quality attribute (QA1), let us suppose that, after a period of time some of the service partners have changed their security policy by imposing authentication filters. The architect formulates a new quality integration intent. The aim was to establish an authentication policy specific to security requirements imposed by the TRS system. The same three patterns P_1, P_2 and P_3 were proposed to the architect. The application of the patterns shows that the "Direct Authentication" pattern (P_3) breaks the portability (QA4) quality attribute. This is illustrated by its criticality degree value (0.058) in the third formula below. Introducing the pattern involves adding BPEL elements which use a message format that is not adapted for the TRS system before the use of the `Facade` service. This leads to break the facade pattern. The other patterns (P_1 and P_2) have had no impact on the other quality attributes. The architect documents the selected pattern as a new design decision in the architecture decision documentation, and commits the resulted new Web service orchestration. The architect then re-applies again the facade

[5]The complete script can be found here: https://sites.google.com/site/wsbscript/soa-patterns-examples/facade-pattern

Table 6.5 : Intent Specification for QII5

Integration Quality Attribute	Security/Access Security
Integration Kind	Add
Related Quality Attribute	
Architectural Area	after ReceiveItinerary: Receive

pattern on the Web service orchestration.

- Trusted Subsystem(WSM score)= 0* (0.750) + 0.085* (0.250) = 0.02125
- Brokered Authentication(WSM score)= 0* (0.750) + 0.644* (0.250) = 0.161
- Direct Authentication(WSM score)= 0.058* (0.750)+ 0.271* (0.250) = 0,1112

To deal with performance quality attribute (QA5) the method proposed the *"Partial Validation Pattern"* [Erl, 2009]. The use of this pattern allows unnecessary data to be avoided before the needed data is transmitted to the clients, hence, decreasing message processing and memory consumption. So the architect decides to make some architectural changes related with data management to increase the performance of the service: i) Short-circuiting the `Validate` operation call to the `Facade` service, and ii) Creating a service with specialized routines in data validation more sophisticated than the validation operation offered by the `Facade` service. The first change is made manually by the architect since it is an isolated simple action, while the second one is performed by configuring the pattern's script. The "Quality impact analysis" report shows that the portability (QA4) quality attribute is affected by the change, since the architectural constraint formalizing the "Service Facade Pattern" was violated (`Invoke` activities having specific positions in the orchestration have been removed). It also shows from the architecture documentation that, the portability (QA4) quality attribute has a conflicting relation type with performance (QA5). The architect decides then to cancel the pattern (Pattern cancellation step) where the corresponding cancellation script of the "Partial Validation Pattern" implementing QA5, which has been already generated, is executed. This leads to remove it from the orchestration. The process returns back to the "Quality Impact Analysis" to make sure that the pattern cancellation has been performed correctly by checking if the constraints of the previously embodied patterns still hold. The architect keeps the orchestration as it was.

Table 6.6 : Intent Specification for QII6

Integration Quality Attribute	Security/Data Security
Integration Kind	Withdraw
Related Quality Attribute	Performance
Architectural Area	after Assign23: Assign

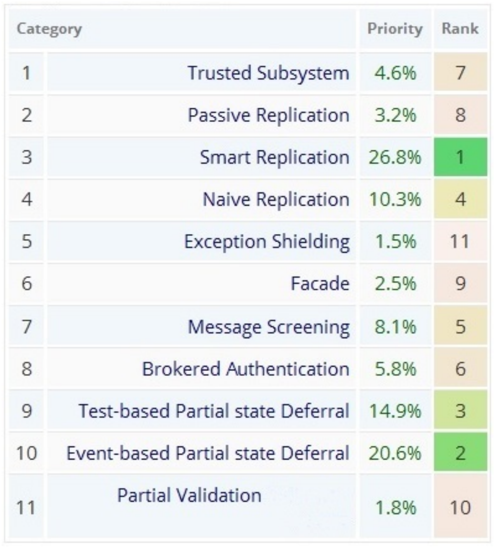
As an attempt (QII6 in Table 6.6) to integrate the performance (QA5) the architect did not find another solution than to try to minimize security policies (QA2). She/He decides to remove the "Exception Shielding" pattern instance applied to secure the `Hotel` service. The process takes another path, since the "Integration Kind" is to withdraw a quality attribute, hence, there is no proposed pattern for the architect here. It goes through the "Patterns Cancellation" step where the corresponding cancellation script of the "Exception Shielding" pattern instance implementing QA2, which has been already generated, is executed. The quality-related impact analysis reports that the security quality attribute is impacted since its constraints do not hold anymore. Also, it reports that its related quality attribute QA4 (Portability) may eventually be impacted by the change. Being aware of the consequences, the architect decides to validate the change.

6.2 Experiment Process

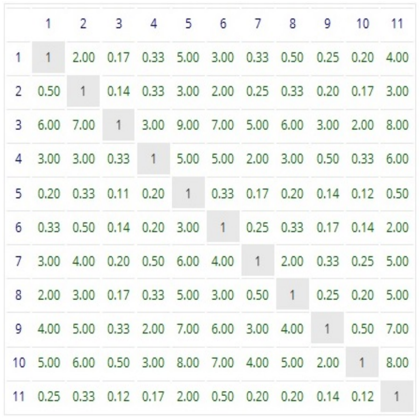
We can distinguish two main roles of SAQIM. First, it is a system that provides an automated support for the integration (application and analysis) of SOA patterns into service orchestrations. Second, it is a recommendation system of SOA patterns satisfying quality attributes for service orchestrations. Due to the actual size of the pattern catalog which includes eleven patterns, we will focus on the evaluation of the first role. Indeed, it is not pertinent for example, to calculate the "precision" and "recall" as metrics to measure the efficiency and thus evaluate the research and selection aspects in SAQIM with the actual size of the catalog. Thereby, we addressed in particular the following research question:
"Compared to a manual quality integration, does the automated support provided by SAQIM give substantial help to architects?".
To answer the research question, we pursued the steps detailed in the following subsections.

6.2.1 Methodology

We compared some measures (presented later) obtained by using SAQIM with those obtained “without using" it[6]. To do so, we simulated quality integration (with and without SAQIM) by using a collection of 16 patterns: eleven of them are real patterns, and the remaining five are “imaginary"[7]. These latter, are unreal patterns in which we have varied randomly the number of BPEL elements (for scripts time specification), and the number of tokens (for OCL constraints time specification) to estimate their specification time by following a specific protocol (explained in the following subsection). Imaginary patterns are introduced in the experimentation to represent a relatively acceptable number of SOA patterns that we can find and use in a real development process. From the other hand, they allow to run simulations with a configurable number of patterns in the catalog so that we can evaluate our method in a reliable way.



Category		Priority	Rank
1	Trusted Subsystem	4.6%	7
2	Passive Replication	3.2%	8
3	Smart Replication	26.8%	1
4	Naive Replication	10.3%	4
5	Exception Shielding	1.5%	11
6	Facade	2.5%	9
7	Message Screening	8.1%	5
8	Brokered Authentication	5.8%	6
9	Test-based Partial state Deferral	14.9%	3
10	Event-based Partial state Deferral	20.6%	2
11	Partial Validation	1.8%	10



	1	2	3	4	5	6	7	8	9	10	11
1	1	2.00	0.17	0.33	5.00	3.00	0.33	0.50	0.25	0.20	4.00
2	0.50	1	0.14	0.33	3.00	2.00	0.25	0.33	0.20	0.17	3.00
3	6.00	7.00	1	3.00	9.00	7.00	5.00	6.00	3.00	2.00	8.00
4	3.00	3.00	0.33	1	5.00	5.00	2.00	3.00	0.50	0.33	6.00
5	0.20	0.33	0.11	0.20	1	0.33	0.17	0.20	0.14	0.12	0.50
6	0.33	0.50	0.14	0.20	3.00	1	0.25	0.33	0.17	0.14	2.00
7	3.00	4.00	0.20	0.50	6.00	4.00	1	2.00	0.33	0.25	5.00
8	2.00	3.00	0.17	0.33	5.00	3.00	0.50	1	0.25	0.20	5.00
9	4.00	5.00	0.33	2.00	7.00	6.00	3.00	4.00	1	0.50	7.00
10	5.00	6.00	0.50	3.00	8.00	7.00	4.00	5.00	2.00	1	8.00
11	0.25	0.33	0.12	0.17	2.00	0.50	0.20	0.20	0.14	0.12	1

Figure 6.3 : Weights for OCL constraints. Figure 6.4 : Decision Matrix.

The experiment was conducted following the next steps:

6.2.2 Data Collection

We have invloved in our experiment three Ph.D students in software engineering and

[6]We mean by “without using SAQIM" that the architect has to choose him(/her)self the patterns and uses the NetBeans BPEL designer to apply them manually.

[7]We used approximately the half of the real patterns total number.

Table 6.7 : OCL constraints specification results

Pattern	Tocl (min)	Var	CD	1-CD	Tocl*(1-CD)	Uocl	Uocl-Umec	NbTokens
Facade (1)	40,25	1,79	0,046	0,975	39,24	0,072	0,062	154
Trusted SubSystem (2)	78,5	0,16	0,032	0,954	74,89	0,058	0,048	366
Passive Replication (3)	65,5	4,04	0,268	0,968	63,40	0,055	0,045	333
Smart Replication (4)	115,5	5,54	0,103	0,732	84,55	0,049	0,039	497
Naive Replication (5)	68	0,66	0,015	0,897	61	0,044	0,034	394
Exception Shielding (6)	36	1,16	0,025	0,985	35,46	0,044	0,034	239
Message Screening (7)	63,5	2,79	0,081	0,919	58,36	0,043	0,033	365
Brokered Authentication (8)	56	1,54	0,058	0,942	52,75	0,041	0,031	363
Test-based Partial state Deferral (9)	62,25	1,62	0,149	0,851	52,97	0,036	0,026	459
Event-based Partial state Deferral (10)	75,5	2,16	0,206	0,794	59,95	0,036	0,026	461
Partial Validation (11)	30	1,29	0,018	0,982	29,46	0,034	0,024	213

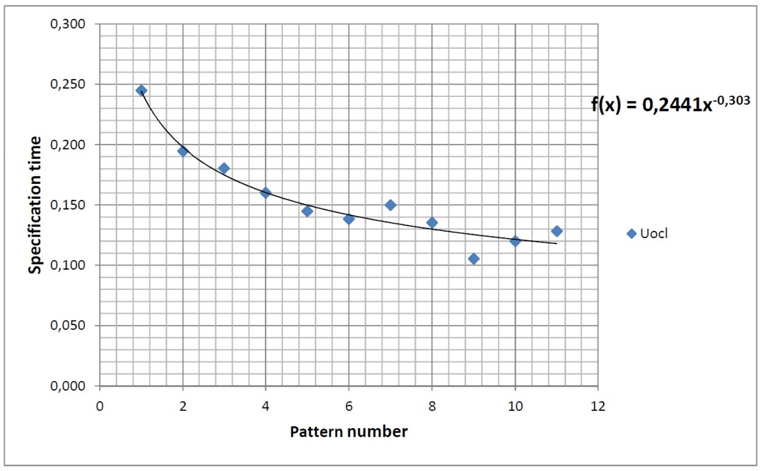


Figure 6.5 : Inverse power regression on Uocl values

programming languages. They have had the task of applying the patterns using NetBeans BPEL designer then measuring and recording the approximative time spent for each pattern. They were also asked to record the time spent in understanding each pattern after reading a textual documentation (retrieved from the literature). In addition, they were taught examples about the *WS-BScript* language. In addition to the first task, they were asked to specify patterns by writing OCL constraints and scripts for the eleven real patterns. These Ph.D students have basic OCL skills, a good knowledge of frameworks, styles and basic patterns of software design. The students were separated and were not told about the final goal of the experiment. Additionally, they were not told about their recorded results to ensure confidentiality. Moreover, students were selected with a relatively similar level of knowledge and background.

Because the level of the Ph.D students skills is close one to another, we notice an insignificant variance (see Column 3 in Table 6.7 and Table 6.8) in the measured times across students for each pattern. Therefore, the recorded times were "homogeneous" and this is why we took the average time. Now, to estimate the specification time for both OCL constraints and scripts for a number of imaginary patterns in a reliable way, we followed a specific protocol. The aim is to estimate the pattern catalog specification overhead from the one hand, and to simulate quality integration with a configurable number of patterns from the other hand.

OCL constraints: The obtained values for the 11 real patterns are depicted in Table 6.7. We first normalized these estimated time values. Indeed, the Ph.D students have naturally acquired experience when specifying each time a new constraint. This experience can bias our experiment. We have thus decided to dismiss it, in order to get the most possible objective values. We have measured an approximative *coefficient of difficulty* (CD) for each constraint. CD represents the architect's opinion on the perceived difficulty when specifying a constraint. We applied here AHP pairwise comparisons for prioritizing OCL constraints difficulty (Figures 6.3 and 6.4) in the same way as for C_1 values (Figures 5.3 and 5.4). The developer expresses her/his opinion (measured on the scale of Table 5.1) like: "constraint i has an absolutely higher difficulty than j" has a value of "9". Then, we multiplied the previous specification time values by $1-CD$.

The next step was to calculate the specification time for a lexical unit (token) in a constraint "`Uocl`". Values were obtained as follows:

$$Uocl = \frac{Tocl * (1 - CD)}{NbTokens} \tag{6.1}$$

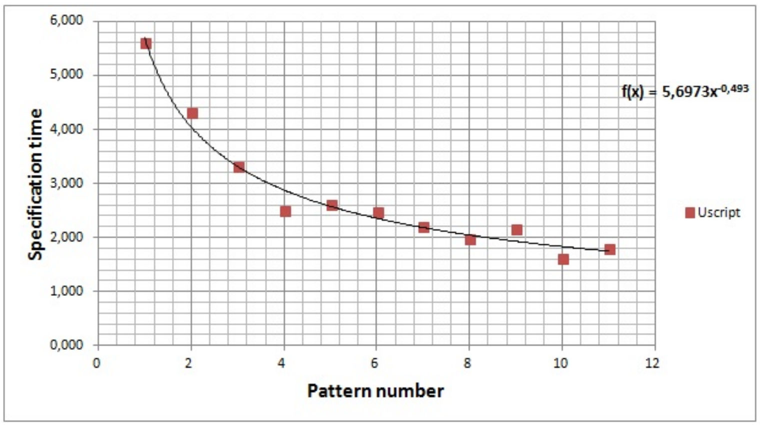


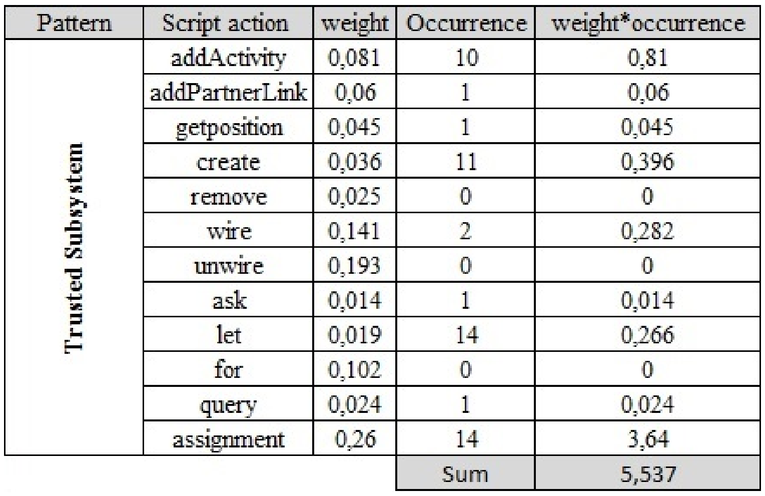
Figure 6.6 : Inverse power regression on Uscript values

Now, having time unit values for each pattern constraint we can apply a regression *f(x)* model to extrapolate values for the other imaginary patterns. We can notice that "Uocl" values (in Table 6.7) have a decreasing trend, but actually they do not converge to zero. Instead, they converge to a minimal value corresponding to specifying a constraint as a "mechanical task", *i.e.* without having to think about complex parts in it. Therefore, we calculated "Umec" and we obtained 0.039 minute/token. So, the function of our regression should be defined as : *f(x)+Umec* . After that, we subtracted "Umec" from "Uocl" values then we applied an inverse power regression on the new values (Uocl-Umec in Table 6.7). We found that the inverse power model is the one that best fits our data. The result is illustrated in Figure 6.5. At the end, we used the following inverse power regression function to extrapolate time unit values for OCL constraints: $f(x) = 0,244x^{-0,303} + 0,039$. Finally, using Formula 6.1 we obtained the specification time *Tspec_ocl*:

$$Tspec\_ocl = \frac{(Uocl - Umec) * NbTokens}{(1 - CD)} \tag{6.2}$$

Scripts: We followed the same steps as for OCL constraints to determine the specification time for pattern scripts, except for the "*coefficient of difficulty*" (CD) estimation. We started first by listing the different script actions[8] used in the scripts, then using pairwise comparisons we calculated the weight of each action according to its difficulty of use (Figures 6.8 and 6.9). The next step was to calculate the occurrences of each action in each script to get its global weight. Figure 6.7 shows an example of the way "CD" values have been estimated. Column 3 in Figure 6.7 shows the individual

[8]To distinguish between adding a BPEL "activity" and adding a "PartnerLink" BPEL element we suffixed "add" by the terms "Activity" and "PartnerLink" (Lines 1 and 2 in Figure 6.7).



Pattern	Script action	weight	Occurrence	weight*occurrence
Trusted Subsystem	addActivity	0,081	10	0,81
	addPartnerLink	0,06	1	0,06
	getposition	0,045	1	0,045
	create	0,036	11	0,396
	remove	0,025	0	0
	wire	0,141	2	0,282
	unwire	0,193	0	0
	ask	0,014	1	0,014
	let	0,019	14	0,266
	for	0,102	0	0
	query	0,024	1	0,024
	assignment	0,26	14	3,64
			Sum	5,537

Figure 6.7 : Scripts CD values estimation

CD values obtained for each action (from Figure 6.8). The overall value of the different weights constitute the pattern's script CD value (5.537 for the script of the Trusted Subsystem Pattern, see Figure 6.7). This corresponds to 9.05% of the overall value for all the pattern scripts which represents 0.0905 (See Table 6.8). We defined the time for adding a single BPEL element by a script as the time unit "`Uscpt`":

$$Uscpt = \frac{Tscpt * (1 - CD)}{NbBpelElem} \tag{6.3}$$

Figure 6.6 shows the result of applying an inverse power regression model:

$$f(x) = 5,6973x^{-0,493} + 1,89$$

$$Tspec\_script = \frac{(Uscpt - Umec) * NbBpelElem}{(1 - CD)} \tag{6.4}$$

$$Tspec\_pattern = Tspec\_ocl + Tspec\_script \tag{6.5}$$

Using the formulas 6.1, 6.2, 6.3 and 6.4 we were able to estimate the specification time (formula 6.5) for the 5 "imaginary" patterns.

6.2.3 Simulation

The aim of the simulation is to evaluate SAQIM's cost effectiveness. Table 6.9 shows the measures for the real patterns used in our simulation process. We calculated the necessary time for integrating a quality attribute without using and with using SAQIM

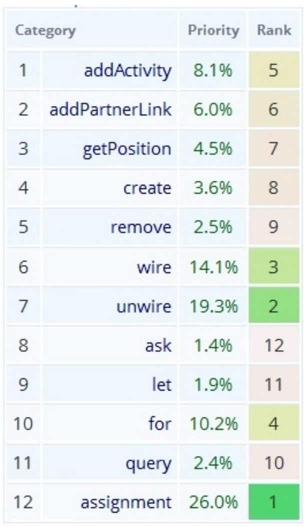
Table 6.8 : Scripts specification results

Pattern	Tscpt	Var	CD	1-CD	Tscpt*(1-CD)	Uscpt	Uscpt-Umec	NbBpel Elem
(1)	39	1,54	0,0378	0,9622	37,52	7,50	5,615	5
(2)	75	2,54	0,0905	0,9095	68,21	6,20	4,311	11
(3)	90	3,5	0,1297	0,8703	78,33	5,22	3,332	15
(4)	81,5	4,66	0,1353	0,8647	70,47	4,40	2,514	16
(5)	50	2,16	0,0967	0,9033	45,17	4,52	2,627	10
(6)	38	1,16	0,0771	0,9229	35,07	4,38	2,494	8
(7)	40,25	0,87	0,0819	0,9181	36,95	4,11	2,216	9
(8)	38,5	1,04	0,0968	0,9032	34,77	3,86	1,974	9
(9)	36	0,29	0,1011	0,8989	32,36	4,04	2,155	8
(10)	50,75	0,79	0,1016	0,8984	45,59	3,51	1,617	13
(11)	19,50	0,29	0,0514	0,9486	18,50	3,70	1,809	5

(Columns 2 and 3). Column 3 includes the pattern script configuration and automatic application time (Column (a)), the OCL constraint configuration time (Column (b)), and the pattern documentation time (Column (c)), when using SAQIM. Column 2 includes the time spent in understanding each pattern as well as the time spent in manually applying a pattern using the Netbeans BPEL editor (without using SAQIM). The last column shows the specification time (OCL constraints and scripts) for each pattern which is used in the simulation process when using SAQIM with measures of Column 3 (Columns (a), (b), and (c)). The simulation has been run in two different situations based on an assumption stating that without SAQIM, the architect has at her/his disposal the same patterns used by the architect that uses SAQIM and which she/he should apply manually to integrate quality attributes. This simplification assumption does not bias the results of the experiment. Rather, it ignores the time spent by architects for searching appropriate patterns, which favors the situation of "not using SAQIM".

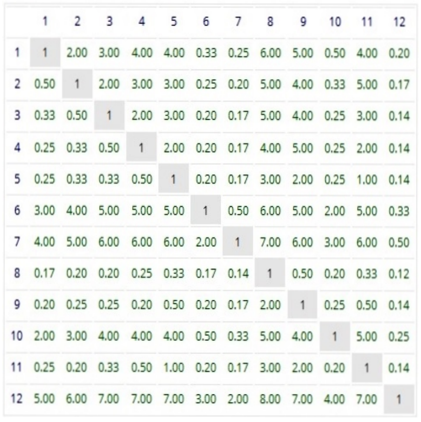
We conducted our simulation process on the TRS system according to three different scenarios. The simulation has been iterated around 50 times to maximize randomness.

Worst case: We simulated the application of SAQIM, using a random generated order of 16 patterns, then we re-applied SAQIM using the same order of patterns but without counting their specification time. We simulated also the quality integration without using SAQIM. The aim in this case is to observe the situation where SAQIM is



Category		Priority	Rank
1	addActivity	8.1%	5
2	addPartnerLink	6.0%	6
3	getPosition	4.5%	7
4	create	3.6%	8
5	remove	2.5%	9
6	wire	14.1%	3
7	unwire	19.3%	2
8	ask	1.4%	12
9	let	1.9%	11
10	for	10.2%	4
11	query	2.4%	10
12	assignment	26.0%	1

Figure 6.8 : Weights for script actions.



	1	2	3	4	5	6	7	8	9	10	11	12
1	1	2.00	3.00	4.00	4.00	0.33	0.25	6.00	5.00	0.50	4.00	0.20
2	0.50	1	2.00	3.00	3.00	0.25	0.20	5.00	4.00	0.33	5.00	0.17
3	0.33	0.50	1	2.00	3.00	0.20	0.17	5.00	4.00	0.25	3.00	0.14
4	0.25	0.33	0.50	1	2.00	0.20	0.17	4.00	5.00	0.25	2.00	0.14
5	0.25	0.33	0.33	0.50	1	0.20	0.17	3.00	2.00	0.25	1.00	0.14
6	3.00	4.00	5.00	5.00	5.00	1	0.50	6.00	5.00	2.00	5.00	0.33
7	4.00	5.00	6.00	6.00	6.00	2.00	1	7.00	6.00	3.00	6.00	0.50
8	0.17	0.20	0.20	0.25	0.33	0.17	0.14	1	0.50	0.20	0.33	0.12
9	0.20	0.25	0.25	0.20	0.50	0.20	0.17	2.00	1	0.25	0.50	0.14
10	2.00	3.00	4.00	4.00	4.00	0.50	0.33	5.00	4.00	1	5.00	0.25
11	0.25	0.20	0.33	0.50	1.00	0.20	0.17	3.00	2.00	0.20	1	0.14
12	5.00	6.00	7.00	7.00	7.00	3.00	2.00	8.00	7.00	4.00	7.00	1

Figure 6.9 : Decision Matrix.

Table 6.9 : Measures used in the simulation process

Pattern	Time (minutes)				
	without using SAQIM	using SAQIM			pattern specification time
		(a)	(b)	(c)	
(1)	43,37	0,59	0,72	1,57	76,77
(2)	52,42	0,65	0,39	3,20	143,10
(3)	44,47	0,93	0,66	1,40	141,73
(4)	56,28	0,86	1,00	1,10	155,02
(5)	56,27	0,71	0,65	1,03	106,16
(6)	59,12	0,67	1,18	2,17	70,53
(7)	49,42	1,17	1,23	2,05	95,31
(8)	49,23	0,63	1,03	1,58	87,53
(9)	56,16	0,57	1,18	2,16	85,33
(10)	48,09	0,56	0,98	1,25	105,54
(11)	57,48	0,71	1,09	1,54	47,96

less beneficial. Figure 6.10 shows the experiment result.

Best case: This case represents the parallel use of patterns. So, we used the same pattern order of the worst case, then we simulated the application of SAQIM by considering the parallel design of three (3) BPEL orchestrations. That means, in the first time we took into account the pattern's specification time. In the remaining two applications we have not taken it into account. We simulated the quality integration without

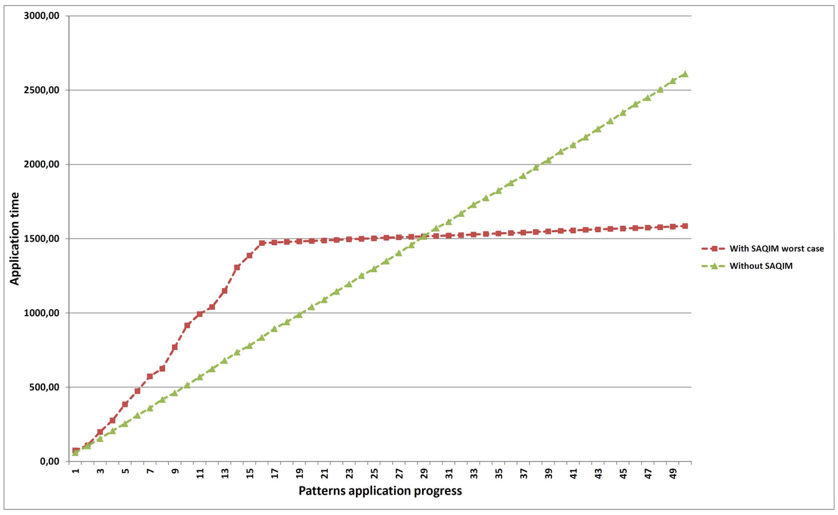


Figure 6.10 : Worst case pattern application time variation

using SAQIM with the same pattern order. The aim in this case is to observe the situation where SAQIM is the most beneficial. The result is given in Figure 6.11.

Random case: we simulated the application of SAQIM by adding each time we want to integrate a quality, the pattern specification time (last column in Table 6.9) only if it is the first use of the applied pattern. In the second step, we simulated the quality integration without using SAQIM. The experiment result is shown in Figure 6.12.

6.2.4 Discussion

The worst case (Figure 6.10) shows that SAQIM begins to be cost effective starting from the 29-th iteration, which corresponds to (more or less) approximatively 25.23 hours (three full-time working days of 8 hours). It is worth noting that, according to our estimations, the pattern catalog specification (with 16 patterns) takes 24.51 hours. The difference between the two measures, which equals 43.2 minutes, is the estimated time taken in this simulation for making profitable the approximate three working days of the (16-pattern) catalog specification time, in the worst case.

The random case (Figure 6.12) shows that SAQIM begins to be cost-effective starting from the 19-th iteration, which corresponds to (more or less) approximatively 22.78 hours (2.84 working days of 8 hours). This period of time corresponds to the time of learning and familiarizing the architect with the method. Indeed, as in all engineering methods learning involves an additional cost. We can also say that the use of the pattern catalog is capitalized after the aforementioned period of time.

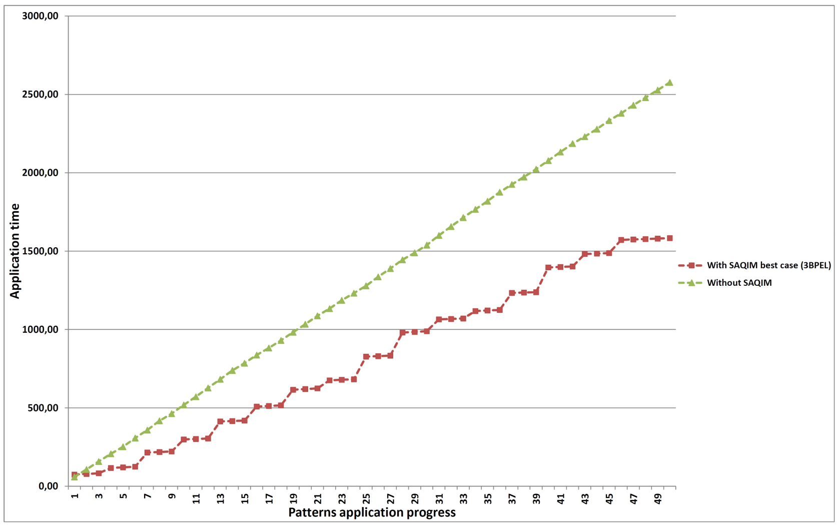


Figure 6.11 : Best case pattern application time variation

Note that in the TRS system which is a medium size project, we embodied using SAQIM ten patterns (including pattern instances like for the "Exception Shielding pattern"). Therefore, we can deduce that SAQIM becomes beneficial after the construction of the second BPEL process (when referring to the random case). Furthermore, if the patterns catalog is used with several BPEL processes in parallel (the best case in Figure 6.11), the catalog specification time will be distributed over all these processes, and hence, the use of SAQIM becomes more beneficial. If we consider that the pattern catalog will be developed by three architects, the specification time (24.51 hours) will be divided by three, i.e. approximately 8.17 hours (one working day).

We have tested SAQIM with a relatively small set of patterns (16 patterns). We repeated the experience by increasing the number of patterns to thirty (30). We kept the 11 real patterns and we created 19 imaginary ones. Then, we generated different random orders of patterns (6 random orders) and used them in the simulation process. The aim of the experiment is to observe the behavior of SAQIM with a larger number of patterns in the catalog. We found that SAQIM's cost effectiveness (for the random case) varies between the 8-th and 23-rd iteration. This variance is related to the patterns order. Note that the result found with 16 patterns is in the range found when using 30 patterns. We can deduce that the number of patterns in the catalog does not affect SAQIM cost effectiveness.

Moreover, to estimate the overhead of the quality impact analysis step (Section 5.2.1 and Section 5.2.2) we have measured its execution average time. We found

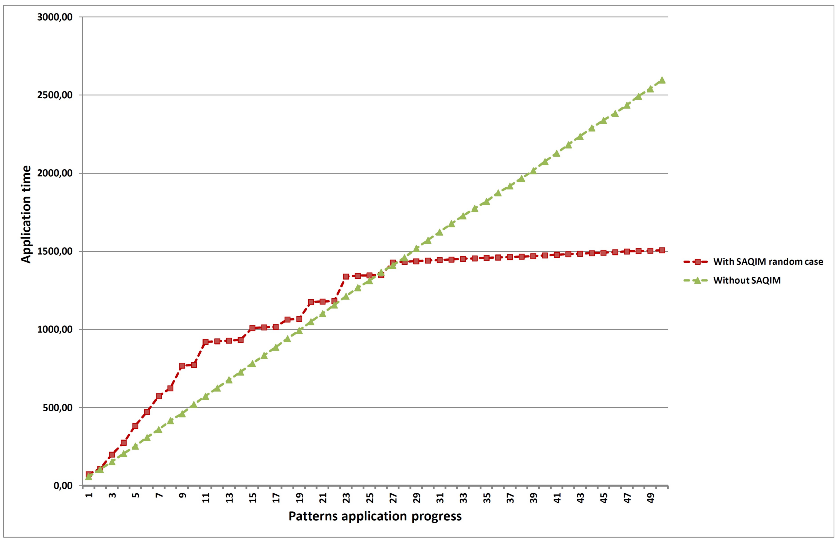


Figure 6.12 : Random case pattern application time variation

that, it takes approximately one minute (including the context-suitability criterion configuration time). This constitutes 25.08% (according to the Columns (a), (b), (c) of Table 6.9) of the overall time for the "Passive Replication" Pattern. This means that, in the worst case, it makes SAQIM beneficial after 25.72 hours instead of 25.23 hours, which represents a difference of a small period of time of 29.4 minutes (0.49 hours).

6.2.5 Threats to validity

Wohlin et al. describe four areas where the validity of the results may be threatened [Wohlin *et al.*, 2012], we discuss threats in each of these areas.

Internal validity: We used a combination of real data and simulated data, which was generated from the real data to construct our dataset. The assessment of SAQIM cost effectiveness may be biased by the person's level of expertise and experience participating in the patterns catalog specification. The specification time assessment of the pattern scripts as well as the OCL architectural constraints may differ from one person to another, which may yield to different results. In our experiment to deal with the selection threat [Wohlin *et al.*, 2012] which concerns the effect of natural variation in human performance, we selected a random group of Ph.D. students.

Construct validity: To increase construct validity, we avoided evaluation apprehension by separating students and by ensuring the confidentiality of the recorded results. Furthermore, the students were not told about the final goal of the experiment to avoid the experimenter expectancies threat, for example when measuring the time for document reading and comprehension of patterns.

Conclusion validity: To increase conclusion validity, we used regression analysis to get reliable estimation of the imaginary patterns specification time. Additionally, we selected students with relatively similar level of knowledge and background to limit the threat of random heterogeneity of subjects. Furthermore, we used an acceptable number of patterns in the simulation and we compared the results of SAQIM with those obtained using a well-known easy-to-use software tool (NetBeans BPEL designer).

External validity: An important threat to the external validity is the use of students as subjects. However, to increase the validity we involved Ph.D. students which have some development experience, and can easily play the role of architects in industry. Furthermore, despite the fact that the context in which the patterns catalog was developed is not part of a software development project it resembles to that of a real service-oriented software development situation.

6.3 Summary

This chapter presented the setup, the process and the result of the conducted evaluations. First, we show through a case study using a real-word web service orchestration the usability of the approach. We detailed each step by giving examples and explanations about the approach applicability. Secondly, we evaluated SAQIM's cost effectiveness. We built a SOA patterns catalog implementing the most experienced quality attributes in web service orchestrations. Then we compared some measures (presented in this chapter) obtained by using SAQIM with those obtained "without using" it. After that, we made a number of simulations in different context (worst, best and random cases). The obtained results was analyzed and discussed. We demonstrated that the use of SAQIM brings a significant assistance and gain of time.

However, it is worth noting that, the evaluation of SAQIM was made by considering it as a macro-process (considering its inputs/outputs). Indeed, the evaluation of SAQIM as a micro-process by addressing each of its steps need a more in depth evaluation where we should consider another experiment process.

CHAPTER 7

Conclusion and Future Work

This chapter summarizes the contributions of this thesis and presents several research directions which require further investigations in the future.

7.1 Conclusion

THIS thesis deals with the problem of integrating non-functional requirements in software architectures. We consider in our work a specific kind of software architectures, which are service-oriented ones, and we deal with a particular specialization of this kind of architectures which are Web service orchestrations concretely defined as BPEL processes. As any software the design of such service-based systems goes first through the development of their architectures. A software architecture is one of the first artifacts of the design process. It manifests the earliest design decisions of a software system, and thus allow to analyze and evaluate the system properties like quality attributes in the development process. In a software development project, quality requirements are important software artifacts that are mainly satisfied at software architecture design time [Bass *et al.*, 2012]. Architects are thus the software developers who are responsible for taking architectural design decisions (ADs) in order to satisfy this kind of requirements. One of the most common design decisions at this stage of a development process is the choice of an architectural style or design pattern. These decisions as well as the reasoning behind them constitute an important architectural knowledge in the development process. Most of this knowledge is ignored by architects and tend to be lost. Architecture design decision documentation models comes to rescue to avoid the vaporization of this knowledge. Additionally, the design process involve changes on the software architecture by making or removing design decisions. Assistance tools support are crucial to handle ADs that shape the software architecture. Specifically, addressing the problem of satisfying NFRs at the architectural design level, involves managing the related architecture design decisions (ADs).

To address these issues we propose an architecture design decision documentation model which is based on patterns as a kind of design decision. Indeed, the choice of architectural patterns as a centric part of our model is motivated by their interesting properties. First they are robust reusable solutions that have been tried and tested. Second, they communicate design decisions at a level that is appropriate to programmers implementing the code [de Silva et Balasubramaniam, 2012], which is appropriate to the kind of service architectures we are dealing with (BPEL orchestrations are XML based representations).

As catalogs of these well-known recurrent design decisions have been proposed in the literature and practice of software engineering (provided mainly with informal descriptions), we argue in this work that such catalogs can be documented in a (more or less) structured, automatically checkable and semi-automatically processable way. Hence, two language are proposed. The first ensures the presence or the absence of a pattern, thus the satisfaction of a quality attribute or not by verifying its structural aspects. This is done by OCL architectural constraints which operate on a BPEL process instance. The second language is a scripting language “WS-BScript” which is necessary to offer a guidance on how to use a pattern and how to apply it into a BPEL orchestration. Fine grained information about the pattern and quality attributes enriches the model. The usefulness of this information is perceived during the quality integration process. Such documentation is then operated in order to assist architects in integrating quality requirements. SAQIM is an on-demand quality integration method that provides such assistance to architects. It is based on a SOA pattern catalog where each pattern is specified by an architectural constraint and an architectural script. The method helps the architect in satisfying the targeted quality attribute, by suggesting to him some service-oriented patterns. In addition, it simulates the application of different competing alternative patterns that satisfy a targeted quality attribute by executing their corresponding scripts. Also, it allows to cancel the instantiation of a pattern by generating then executing its cancellation script using our implemented WS-BScript interpreter. Moreover, the method helps in choosing the most appropriate pattern alternative that satisfies its preferences between competing ones. By "most" appropriate pattern, we mean a pattern: i) that satisfies the more the tackled quality attribute (the pattern that gives the best scores for the evaluation criteria), and ii) that affects the less the other quality requirements, already satisfied and documented in the software architecture through the use of the quality impact analysis process. This latter, allows reasoning about the impact of a pattern on the previously integrated qualities of the service orchestration. It works then in a complementary way with SAQIM. The reasoning process operates on the architecture documentation built using the model we proposed. It is based on the evaluation of OCL architectural constraints formalizing the structural aspects of patterns. The process provides to the architect a warning system that notifies him about patterns related change and allow him to control the architecture evolution. We implemented a prototype tool that interprets and evaluates OCL constraints as well as the documentation of validated design decisions.

In summary, our approach contributes by a new vision for the integration and satisfaction of non-functional properties in Web services orchestrations defined with the BPEL language. The originality of the proposed approach comes from the fact that it operates on static quality properties of such architectures. It helps developers of these applications to build web service orchestration incrementally by answering at the same time quality constraints that must be considered upstream in the development cycle, unlike other approaches that consider dynamic qualities of already designed services. Moreover, our method intervenes on the way to arrange the elements that constitute the composition. Our method allows to preserve, on-the-fly, the architecture coherence with respect to the quality each time a new one is integrated by providing the means to implement it. It also allows a considerable gain in time and efficiency in the design process.

7.2 Future Work

There are several improvements that could be made to the work done in this thesis, and research directions which require further investigations in the future. We listed some of them in the items below:

- We plan to define a simulation (decision) system to study the effect that yields the application of all possible initial combinations of selected patterns (that implement the required quality attributes specified in the NFRs). The aim is to generate all possible patterns application sequences, simulate their application, and then record their impact on the embodied qualities. Then, we look for the best application sequence which gives a service orchestration with the minimum effect on the qualities. We believe that the chosen sequence to embody the desired quality attributes is important and yields to different results (*i.e.* a service orchestration with different qualities or differently affected qualities and with different design costs).
- Another future work we are considering is to evaluate SAQIM as a quality integration micro-process by addressing each of its steps. More particularly, we plan to conduct a validation of the pattern selection step as well as the quality impact analysis step. For example, to evaluate the weighted sum model for pattern ranking, a sensitivity analysis [Triantaphyllou et Sanchez, 1997] on the decision criteria weights and the criteria values could be introduced for studying and increasing the trustworthiness about the provided decision on patterns ranking.
- We plan also to enrich the specification of a pattern, by dependency relationships with other patterns. This may improve the structure of the pattern catalog with composite patterns therefore providing a better assistance to the architect when integrating quality attributes. We would like to enhance the organization of the catalog of patterns. Instead of a flat organization, we want to define a hierarchical one, built using some classification techniques like FCA (Formal Concept Analysis [Ganter et Wille, 1999]). In this way, we can easily look for substitutable patterns which can be proposed together to the architect in the process. We aim also to improve the design of patterns in the catalog so that they do not affect the overall service orchestration performance. In fact, patterns use elements of BPEL language that may include extensive use of fault tolerance techniques like

Recovery Blocks, Return Fastest Response and Deadline Mechanisms. The improvement could be achieved by techniques of QoS computation like the one of [Mukherjee *et al.*, 2008] which could be applied on elements composing a pattern.

- Another improvement would be to integrate in the proposed method an impact analysis activity on the business logic aspect. Indeed, our process evaluates the impact of the evolved quality attribute on the other quality attributes. We plan thus to evaluate also the impact on the existing functionality implemented in the software architecture.

- A future work we are planing to conduct is on the specification of set of metrics to measure the complexity of OCL expressions. The aim is to have a basis on which the developer can refer for specifying less complex patterns architectural constraints, therefore easily understandable and reusable.

- We envisage the development of a recommendation system of composite web services including patterns in their design, thus integrating quality attributes. This assumes that the individual services should be designed with the approach that we have proposed. Thus, pattern identification techniques could be employed to detect the highest possible number of patterns in a composite web service (BPEL orchestration) and possibly the service with higher quality of service. Therefore, this assumption could involve modifications on the BPEL language allowing such identification process. We believe that this approach allows to have Web services compositions with better quality. However, these improvements are research ideas that require further reflection and investigation to validate their feasibility and applicability.

Overall, while there are still improvements to be undertaken on the proposed work in this thesis, we believe that our approach is promising and could be used in a complementary way with Web service based business processes design methods for ensuring the quality in BPEL Web service orchestrations.

List of Figures

2.1 SOA infrastructure 11
2.2 Service orchestration and service choreography 16
2.3 An excerpt of the BPEL/WSDL metamodel 20
2.4 An excerpt of the ISO/IEC 9126 quality model 38

3.1 Links between Architecture Decisions and Quality Attributes 61
3.2 Relationships between Quality Attributes 64

4.1 A process for integrating quality requirements in engineering Web service business processes 79
4.2 Pattern Specification 83

5.1 A Micro-Process of Architecture Evolution 94
5.2 The Proposed Micro-Process of Architecture Evolution 95
5.3 Decision Matrix. 105
5.4 Weights for C1. 105
5.5 Weights of C2 for the replication pattern. 106
5.6 Decision Matrix. 106

6.1 An excerpt of the TRS Business process showing the distribution of the embodied patterns 112
6.2 TRS Business process 116
6.3 Weights for OCL constraints. 120
6.4 Decision Matrix. 120
6.5 Inverse power regression on Uocl values 121
6.6 Inverse power regression on Uscript values 123
6.7 Scripts CD values estimation 124

6.8 Weights for script actions. 126
6.9 Decision Matrix. 126
6.10 Worst case pattern application time variation 127
6.11 Best case pattern application time variation 128
6.12 Random case pattern application time variation 129

List of Tables

4.1 Template for Quality Integration Intent Description 81

5.1 Scale of relative importance 104

6.1 Embodied Patterns and their achieved Quality attributes 111
6.2 Intent Specification for QII1 114
6.3 Intent Specification for QII2 115
6.4 Intent Specification for QII3 117
6.5 Intent Specification for QII5 118
6.6 Intent Specification for QII6 119
6.7 OCL constraints specification results 121
6.8 Scripts specification results 125
6.9 Measures used in the simulation process 126

List of Listings

2.1 Hello service WSDL example 14
2.2 A general structure of a BPEL process 17
3.1 Brokered Authentication Pattern application script 68
3.2 Service Facade Pattern application script 69
3.3 Passive Replication Pattern Architectural constraint 72
3.4 Exception Shielding Pattern Architectural constraint 74
4.1 Trusted Subsystem Pattern application script 84
4.2 Trusted Subsystem pattern cancellation script 88

Bibliography

[Akkiraju *et al.*, 2003] Rama Akkiraju, Richard Goodwin, Prashant Doshi, et Sascha Roeder. A method for semantically enhancing the service discovery capabilities of UDDI. In *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), August 9-10, 2003, Acapulco, Mexico*, pages 87–92, 2003.

[Al-naeem *et al.*, 2005] Tariq Al-naeem, Ian Gorton, Muhammed Ali Babar, Fethi Rabhi, et Boualem Benatallah. A quality-driven systematic approach for architecting distributed software applications. In *Proc. of ICSE'05*, pages 244–253. ACM Press, 2005.

[Allen, 1997] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.

[Ardagna et Pernici, 2007] Danilo Ardagna et Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Softw. Eng.*, 33(6):369–384, 2007.

[Azmeh *et al.*, 2011] Zeina Azmeh, Maha Driss, Fady Hamoui, Marianne Huchard, Naouel Moha, et Chouki Tibermacine. Selection of composable web services driven by user requirements. In *Proc. of ICWS'11*, Washington DC, July 2011. IEEE CS.

[Balasubramaniam et Vasant, 1991] Ramesh Balasubramaniam et Dhar Vasant. Representation and maintenance of process knowledge for large scale systems development. In *In Proceeding of the 6th Knowledge-based Software Engineering Conference, KBSE*, pages 223–231, september 1991.

[Baligand *et al.*, 2006] Fabien Baligand, Didier Le Botlan, Thomas Ledoux, et Pierre Combes. A language for quality of service requirements specification in web services orchestrations. In *Proc. of ICSOC'06*. Springer-Verlag, 2006.

[Bass *et al.*, 2001] L. Bass, F. Bachmann, et M. Klein. Quality attribute design primitives and the attribute driven design method. In *Proceedings of the 4th International Conference on Product Family Engineering*, pages 169–186. Springer-Verlag, 2001.

[Bass *et al.*, 2003] L. Bass, P. Clements, et R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.

[Bass *et al.*, 2006] Len Bass, Paul Clements, Robert L. Nord, et Judith Stafford. *Capturing and Using Rationale for a Software Architecture*, pages 255–272. Springer, in a. h. dutoit et al., rationale management in software engineering édition, 2006.

[Bass *et al.*, 2012] L. Bass, P. Clements, et R. Kazman. *Software Architecture in Practice, 3rd Ed.* Addison-Wesley, 2012.

[Boehm *et al.*, 1976] B. W. Boehm, J. R. Brown, et M. Lipow. Quantitative evaluation of software quality. In *In Proceeding of the 2nd International Conference on Software Engineering*, pages 592–605, San Francisco, California, USA, 1976. IEEE Computer Society Press.

[Bosch, 2004] Jan Bosch. Software architecture: The next step. In *In Proceedings of First European Workshop, EWSA 2004*, pages 194–199, May 2004.

[BPL, 2007] Web services business process execution language specification, version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[Breivold *et al.*, 2012] Hongyu Pei Breivold, Ivica Crnkovic, et Magnus Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40, Janvier 2012.

[Briand *et al.*, 2005] L.C. Briand, Y. Labiche, M. Di Penta, et H. Yan-Bondoc. An experimental investigation of formality in uml-based development. *IEEE Transactions on Software Engineering*, 31:833–849, 2005.

[Broy *et al.*, 2006] Manfred Broy, Florian Deissenboeck, et Markus Pizka. Demystifying maintainability. In *In Proceeding of the 2006 international workshop on Software quality (WoSQ'06)*, pages 21–26. ACM Press, 2006.

[Burge et Brown, 2006] J. Burge et D.C. Brown. *Rationale-based Support for Software Maintenance*, chapitre Rationale Management in Software Engineering, pages 273–296. Springer-Verlag, in a. dutoit, r. mccall, i. mistrik, and b. paech édition, 2006.

[Buschmann *et al.*, 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, et M. Stal. *Pattern Oriented Software Architecture: A System of Patterns.* John Wiley & Sons, 1996.

[Canfora *et al.*, 2008] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, et Maria Luisa Villani. A framework for qos-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, Octobre 2008.

[Capilla *et al.*, 2007] Rafael Capilla, Francisco Nava, et Juan C. Duenas. Modeling and documenting the evolution of architectural design decisions. In *In Proceeding of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI'07)*. IEEE Computer Society, 2007.

[Cardoso *et al.*, 2004] Jorge Cardoso, Amit P. Sheth, John A. Miller, Jonathan Arnold, et Krys Kochut. Quality of service for workflows and web service processes. *J. Web Semantics*, 1(3):281–308, 2004.

[Choi *et al.*, 2006] Heeseok Choi, Youhee Choi, et Keunhyuk Yeom. An integrated approach to quality achievement with architectural design decisions. *JSW*, 1(3):40–49, 2006.

[Chung *et al.*, 1999] Lawrence Chung, B. A. Nixon, E. Yu, et Mylopoulos J. *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers, 1999.

[Chung et Nixon, 1995] L. Chung et Brian A. Nixon. Dealing with non-functional requirements: Three experimental studies of a process-oriented approach. In *Proceeding of 17th International Conference on Software Engineering (ICSE'95)*, pages 25–37, April 1995.

[Clements *et al.*, 2002] Paul Clements, Rick Kazman, et Mark Klein. *Evaluating Software Architectures, Methods and Case Studies.* Addison-Wesley, 2002.

[Clements *et al.*, 2003] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, et J. Stafford. *Documenting Software Architectures, Views and Beyond.* Addison-Wesley, 2003.

[Consortium, 1999] World Wide Web Consortium. Xml path language (xpath) version 1.0. http://www.w3.org/TR/xpath/, 1999.

[Cysneiros et Sampaio do Prado Leite, 2004] Luiz Marcio Cysneiros et Julio Cesar Sampaio do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. *IEEE TSE*, 30(5):328–350, 2004.

[de Boer *et al.*, 2007] Remco C. de Boer, Rik Farenhorst, Patricia Lago, Hans van Vliet, Viktor Clerc, et Anton Jansen. Architectural knowledge: Getting to the core. In *Proceedings of the Quality of Software Architectures 3rd International Conference on Software Architectures, Components, and Applications*, QoSA'07, pages 197–214, Berlin, Heidelberg, 2007. Springer-Verlag.

[de Boer et Farenhorst, 2008] Remco C. de Boer et Rik Farenhorst. In search of 'architectural knowledge'. In *Proceedings of the 3rd International Workshop on Sharing and Reusing Architectural Knowledge*, SHARK '08, pages 71–78, New York, NY, USA, 2008. ACM.

[de Silva et Balasubramaniam, 2012] Lakshitha de Silva et Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *J. Syst. Softw.*, 85(1):132–151, Janvier 2012.

[Deissenboeck *et al.*, 2007] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, et J.-F. Girard. An activity-based quality model for maintainability. In *In Proceeding of the 23rd International Conference on Software Maintenance (ICSM '07)*, pages 184–193. IEEE Computer Society, 2007.

[Deissenboeck *et al.*, 2009] Florian Deissenboeck, Elmar Juergens, Klaus Lochmann, et Stefan Wagner. Software quality models: Purposes, usage scenarios and requirements. In *In Proceeding of 7th International Workshop on Software Quality (WoSQ '09)*. IEEE Computer Society, 2009.

[Dresden., 2009] T. U. Dresden. Ocl compiler web site. http://dresden-ocl.sourceforge.net/, 2009.

[Driss *et al.*, 2010] Maha Driss, Naouel Moha, Yassine Jamoussi, Jean-Marc Jézéquel, et Henda Hajjami Ben Ghézala. A requirement-centric approach to web service modeling, discovery, and selection. In *Proc. of ICSOC'10*, pages 258–272. Springer-Verlag, 2010.

[Dromey, 1995] R. Geoff Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–163, 1995.

[Dromey, 1996] R. Geoff Dromey. Cornering the chimera. *IEEE Software*, 13:33–43, 1996.

[Dumas *et al.*, 2010] Marlon Dumas, Luciano García-Bañuelos, Artem Polyvyanyy, Yong Yang, et Liang Zhang. Aggregate quality of service computation for composite services. In *ICSOC*, éditeurs Paul P. Maglio, Mathias Weske, Jian Yang, et Marcelo Fantinato, volume 6470 de *Lecture Notes in Computer Science*, 2010.

[Durdik et Reussner, 2012] Zoya Durdik et Ralf Reussner. Position paper: approach for architectural design and modelling with documented design decisions (admd3). In *Proc. of QoSA '12*, pages 49–54, New York, NY, USA, 2012.

[Durdik, 2011] Zoya Durdik. Towards a process for architectural modelling in agile software development. In *Proc. of QoSA'11*, pages 183–192. ACM, 2011.

[Eick *et al.*, 2001] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, et Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[Erl, 2009] Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2009.

[Feng *et al.*, 2013] Yuzhang Feng, Le Duy Ngan, et Rajaraman Kanagasabai. Dynamic service composition with service-dependent qos attributes. In *2013 IEEE 20th International Conference on Web Services, Santa Clara, CA, USA, June 28 - July 3, 2013*, ICWS'13, pages 10–17, 2013.

[Fishburn, 1967] Peter C. Fishburn. *Additive Utilities with Incomplete Product Sets: Application to Priorities and Assignments*, volume 15. INFORMS, 1967.

[Foundation, 2009] Eclipse Foundation. Model Development Tools website. http://www.eclipse.org/modeling/mdt/, 2009.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Sofware*. Addison-Wesley Professional Computing Series, 1995.

[Ganter et Wille, 1999] B. Ganter et R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, Inc., 1999.

[Garlan *et al.*, 2000] David Garlan, Robert T. Monroe, et David Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, éditeurs Gary T. Leavens et Murali Sitaraman, pages 47–68. Cambridge University Press, 2000.

[Georgiadou, 2003] Elli Georgiadou. Gequamo—a generic, multilayered, customisable, software quality model. *Software Quality Journal*, 11(4):313–323, 2003.

[Grady, 1992] Robert B. Grady. *Practical software metrics for project management and process improvement*. Prentice Hall, 1992.

[Gross et Yu, 2000] Daniel Gross et Eric Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6:18–36, 2000.

[Groupe, 2011] Object Management Groupe. Business process model and notation (bpmn) specification, version 2.0. OMG Website: http://www.omg.org/spec/BPMN/2.0/PDF, January 2011.

[Heesch et Avgeriou, 2009] Uwe van Heesch et Paris Avgeriou. *A Pattern-based Approach Against Architectural Knowledge Vaporization*. University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science, 2009.

[Hochstein et Lindvall, 2005] Lorin Hochstein et Mikael Lindvall. Combating architectural degenration: A survey. *Information and Software Technology*, 47(10):693–707, July 2005.

[IEEE, 2000] IEEE. Ansi/ieee std 1471-2000, recommended practice for architectural description of software-intensive systems, 2000.

[ISO, 2001] ISO. Software engineering - product quality - part 1: Quality model. International Organization for Standardization web site. ISO/IEC 9126-1. http://www.iso.org, 2001.

[ISO/IEC/(IEEE), 2011] ISO/IEC/(IEEE). Systems and software engineering - architecture description, May 2011.

[Jackson, 2002] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodoly*, 11:256–290, April 2002.

[Jaeger *et al.*, 2004] Michael C. Jaeger, Gregor Rojec-Goldmann, et Gero Muhl. Qos aggregation for web service composition using workflow patterns. In *Proceedings of the Enterprise Distributed Object Computing Conference, Eighth IEEE International*, EDOC '04, pages 149–159, Washington, DC, USA, 2004. IEEE Computer Society.

[Jansen et Bosch, 2005] Anton Jansen et Jan Bosch. Software architecture as a set of architectural design decisions. In *In Proceeding of of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA'05)*, pages 109–120. IEEE CS, 2005.

[Jansen, 2008] Anton Jansen. *Architectural design decisions.* PhD thesis, University of Groningen, Institute for Mathematics and Computing Science, August 2008.

[Jintae, 1989] Lee Jintae. Decision representation language (drl) and its support environment. Rapport technique, MIT Artificial Intelligence Laboratory, August 1989.

[Kim *et al.*, 2009] Suntae Kim, Dae-Kyoo Kim, Lunjin Lu, et Sooyong Park. Quality-driven architecture development using architectural tactics. *Elsevier JSS*, 82(8):1211–1231, August 2009.

[Kim et Garlan, 2010] Jung Soo Kim et David Garlan. Analyzing architectural styles. *Journal of Systems and Software*, 83(7):1216–1235, 2010.

[Kitchenham *et al.*, 1997] Barbara Kitchenham, Steve Linkman, Alberto Pasquini, et Vincenzo Nanni. The squid approach to defining a quality model. *Software Quality Journal*, 6(3):211–233, 1997.

[Klein *et al.*, 1999] Mark H. Klein, Rick Kazman, Leonard J. Bass, S. Jeromy Carrière, Mario Barbacci, et Howard F. Lipson. Attribute-based architecture styles. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'99)*, pages 225–244, Deventer, The Netherlands, The Netherlands, 1999.

[Klein *et al.*, 2011] Adrian Klein, Fuyuki Ishikawa, et Shinichi Honiden. Efficient heuristic approach with improved time complexity for qos-aware service composition. In *IEEE International Conference on Web Services, ICWS 2011, Washington, DC, USA, July 4-9, 2011*, pages 436–443, 2011.

[Kläs *et al.*, 2009] Michael Kläs, Jens Heidrich, Jürgen Münch, et Adam Trendowicz. Cqml scheme: A classification scheme for comprehensive quality model landscapes.

In *In Proceeding of the 35th EUROMICRO Conference Software Engineering and Advanced Applications*, pages 243–250. IEEE Computer Society, 2009.

[Kruchten *et al.*, 2006] Philippe Kruchten, Patricia Lago, et Hans van Vliet. Building up and reasoning about architectural knowledge. In *In Proceedings of the Second International Conference on the Quality of Software Architectures, QoSA*, pages 43–58. Lecture Notes in Computer Science 4214, Springer-Verlag, 2006.

[Kruchten *et al.*, 2009] Philippe Kruchten, Rafael Capilla, et Juan Carlos Duenas. The decision view's role in software architecture practice. *IEEE Software*, 26(2):36–42, 2009.

[Kruchten, 1995] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6):42–50, 1995.

[Kruchten, 2004] Philippe Kruchten. An ontology of architectural design decisions in software intensive systems. In *In Proceeding of the 2nd Groningen Workshop Software Variability*, pages 54–61, 2004.

[Lago et van Vliet, 2005] Patricia Lago et Hans van Vliet. Explicit assumptions enrich architectural models. In *In Proceeding of the 27th International Conference on Software Engineering (ICSE'05)*, pages 206–214. ACM Press, May 2005.

[Lehman et Ramil, 2002] M.M. Lehman et J. F. Ramil. Software evolution. *Marciniak J. (ed.), Encyclopedia of Software Engineering, 2nd Ed, Wiley*, 2002.

[Lenhard, 2011] Jörg Lenhard. A pattern-based analysis of ws-bpel and windows workflow. Rapport Technique 88, Lehrstuhl für Praktische Informatik, 2011.

[Lindvall *et al.*, 2002] Mikael Lindvall, Roseanne Tesoriero, et Patricia Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *In Proceeding of the Eighth IEEE Symposium on Software Mertrics (METRICS'02)*, pages 77–86, Ottawa, Ontario, Canada, June 2002.

[MacKenzie *et al.*, 2006] Matthew MacKenzie, Ken Laskey, Peter F. Brown, Metz Rebekah, et Booz Allen Hamilton. *Reference Model for Service Oriented Architecture 1.0.* August 2006.

[Madhavji et Tassé, 2003] Nazim H. Madhavji et Josée Tassé. Policy-guided software evolution. In *Proceeding of the 19th International Conference on Software Maintenance (ICSM'03)*, pages 75–82. IEEE Computer Society Press, 2003.

[Marew *et al.*, 2009] Tegegne Marew, Joon-Sang Lee, et Doo-Hwan Bae. Tactics based approach for integrating non-functional requirements in object-oriented analysis and design. *Journal of Systems and Software*, 82(10):1642–1656, 2009.

[Marinescu et Ratiu, 2004] Radu Marinescu et Daniel Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In *In Proceeding of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 192–201. IEEE Computer Society, 2004.

[McCall *et al.*, 1977] J. McCall, P. Richards, et G. Walters. Factors in software quality. Rapport technique, (RADC)- TR-77-369, Vols. 1–3, Rome Air Development Center, United States Air Force, Hanscom AFB, MA, 1977.

[Mead, 2006] N.R. Mead. White paper: Requirements prioritization case study using ahp. Rapport technique, Software Engineering Institute, Carneige Mellon University, 2006.

[Mens et D'Hondt, 2000] Tom Mens et Theo D'Hondt. Automating support for software evolution in uml. *Automated Software Engineering Journal*, 7(1):39–59, 2000.

[Merkle, 2010] Bernhard Merkle. Stop the software architecture erosion. Tutorial in SPLASH'10, Reno, Nevada, USA, 2010.

[MIC, 2001] Xlang, web services for business process design. http://xml.coverpages.org/XLANG-C-200106.html, 2001.

[Milanovic et Malek, 2004] Nikola Milanovic et Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8:51–59, 2004.

[Monroe, 2001] Robert T. Monroe. Capturing software architecture design expertise with armani. Rapport technique, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2001.

[Mosser et Blay-Fornarino, 2013] SéBastien Mosser et Mireille Blay-Fornarino. "adore", a logical meta-model supporting business process evolution. *Science of Computer Programming*, 78(8):1035–1054, Aôut 2013.

[Mukherjee *et al.*, 2008] Debdoot Mukherjee, Pankaj Jalote, et Mangala Gowri Nanda. Determining qos of ws-bpel compositions. In *Proceedings of the 6th International Conference on Service-Oriented Computing*, ICSOC '08, pages 378–393, Berlin, Heidelberg, 2008. Springer-Verlag.

[Mylopoulos *et al.*, 1992] John Mylopoulos, Lawrence Chung, et Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE TSE*, 18(6):483–497, June 1992.

[Neto *et al.*, 2016] Plácido A. Souza Neto, Genoveva Vargas-Solar, Umberto Souza da Costa, et Martin A. Musicante. Designing service-based applications in the presence of non-functional properties: A mapping study. *Information & Software Technology*, 69:84–105, 2016.

[Niemelä et Immonen, 2007] Eila Niemelä et Anne Immonen. Capturing quality requirements of product family architecture. *Information and Software Technology*, 49(11-12):1107–1120, 2007.

[OASIS, 2002] OASIS. Uddi version 2.04 api specification. http://www.uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf, 2002.

[OMG, 2010] OMG. Object constraint language specification, version 2.2, document formal/2010-02-01. Object Management Group Web Site: http://www.omg.org/spec/OCL/2.2/PDF, 2010.

[Parnas, 1994] David Lorge Parnas. Software aging. In *In Proceeding of the 16th International Conference on Software Engineering (ICSE'94*, Sorrento, Italy, May 1994. IEEE Computer Society Press and ACM Press.

[Peltz, 2003] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct 2003.

[Perry et Wolf, 1992] Dewayne E. Perry et Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Rajan et Sullivan, 2005] Hridesh Rajan et Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *Proceeding of the 27th international conference on Software engineering (ICSE'05)*, pages 59–68. ACM, 2005.

[Rausch, 2000] Andreas Rausch. Software evolution in componentware using requirements/assurances contracts. In *Proceeding of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 147–156. ACM Press, 2000.

[Rosenberg *et al.*, 2007] Florian Rosenberg, Christian Enzi, Anton Michlmayr, Christian Platzer, et Schahram Dustdar. Integrating quality of service aspects in top-down business process development using WS-CDL and WS-BPEL. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*, pages 15–26, 2007.

[Saaty, 1980] T.L. Saaty. *The Analytic Hierarchy Process.* McGraw-Hill, New york, 1980.

[Sheng *et al.*, 2014] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, et Xiaofei Xu. Web services composition: A decade's overview. *Information Sciences*, 280:218–238, 2014.

[Steyaert *et al.*, 1996] P. Steyaert, C. Lucas, K. Mens, et T. D'Hondt. Reuse contracts: managing the evolution of reusable assets. In *In Proceeding of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, pages 268–285. ACM Press, 1996.

[Sun et Zhao, 2012] Sherry X. Sun et Jing Zhao. A decomposition-based approach for service composition with global qos guarantees. *Information Sciences*, 199:138–153, Septembre 2012.

[Tibermacine *et al.*, 2005] Chouki Tibermacine, Régis Fleurquin, et Salah Sadou. Nfrs-aware architectural evolution of component-based software. In *Proceedings of the 20th IEEE/ACM ASE'05*, pages 388–391, Long Beach, California, USA, November 2005. ACM Press.

[Tibermacine et Zernadji, 2011] Chouki Tibermacine et Tarek Zernadji. Supervising the evolution of web service orchestrations using quality requirements. In *Proc. of ECSA'11*, pages 1–16, Essen, Germany, September 2011. Springer-Verlag.

[Tibermacine, 2014] Chouki Tibermacine. *Software Architecture 2*, chapitre Software Architecture: Architecture Constraints. John Wiley and Sons, New York, USA, 2014.

[Ton That *et al.*, 2012] Tu Minh Ton That, Salah Sadou, et Flavio Oquendo. Using Architectural Patterns to Define Architectural Decisions. In *Proc. of WICSA/ECSA'12*, pages 196–200, Helsinki, Finland, Aôut 2012.

[Triantaphyllou *et al.*, 1999] E. Triantaphyllou, B. Shu, S. Nieto Sanchez, et T. Ray. *Multi-Criteria Decision Making: An Operations Research Approach*, volume 15, pages 175–186. J. Wiley, New York, 1999.

[Triantaphyllou et Sanchez, 1997] E. Triantaphyllou et A. Sanchez. A sensitivity analysis approach for some deterministic multi-criteria decision-making methods. *Decision Sciences*, 28(1):151–194, 1997.

[Tyree et Akerman, 2005] Jeff Tyree et Art Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, March/April 2005.

[van der Ven *et al.*, 2006] Jan Salvador van der Ven, Anton G.J. Jansen, Jos A.G. Nijhuis, et Jan Bosch. Design decisions: The bridge between rationale and architecture. In *Rationale Management in Software Engineering*, éditeurs Allen H. Dutoit, Raymond McCall, Ivan Mistrík, et Barbara Paech, pages 329–348. Springer Berlin Heidelberg, 2006.

[Wang *et al.*, 2004] Hongbing Wang, Joshua Zhexue Huang, Yuzhong Qu, et Junyuan Xie. Web services: problems and future directions. *Journal of Web Semantics*, 1:309–320, 2004.

[WCI, 2002] Web service choreography interface (wsci) 1.0. http://www.w3.org/TR/wsci/, 2002.

[WCL, 2004] Web services choreography description language version 1.0. https://www.w3.org/TR/ws-cdl-10/, 2004.

[Wohlin *et al.*, 2012] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, et Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[World Wide Web Consortium, 2004] World Wide Web Consortium. Web services architecture. 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[World Wide Web Consortium, 2007] World Wide Web Consortium. Web services policy 1.5 - framework. 2007. https://www.w3.org/TR/ws-policy/.

[WSC, 2002] Web services conversation language (wscl) 1.0. http://www.w3.org/TR/wscl10/, 2002.

[Yu *et al.*, 2008] Qi Yu, Xumin Liu, Athman Bouguettaya, et Brahim Medjahed. Deploying and managing web services: Issues, solutions, and directions. *The VLDB Journal*, 17(3):537–572, Mai 2008.

[Zeng *et al.*, 2003] Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, et Quan Z. Sheng. Quality driven web services composition. In *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, pages 411–421, New York, NY, USA, 2003. ACM.

[Zernadji *et al.*, 2014a] Tarek Zernadji, Chouki Tibermacine, et Foudil Cherif. Processing the evolution of quality requirements of web service orchestrations: a pattern-based approach. In *Proc. of WICSA'14*, Sydney, Australia, April 2014. IEEE CS.

[Zernadji *et al.*, 2014b] Tarek Zernadji, Chouki Tibermacine, et Foudil Cherif. Quality-driven design of web service business processes. In *Proc. of WETICE/AROSA'14*, Parme, Italie, Juin 2014. IEEE CS.

[Zheng *et al.*, 2013] Huiyuan Zheng, Weiliang Zhao, Jian Yang, et Athman Bouguettaya. Qos analysis for web service compositions with complex structures. *IEEE T. Services Computing*, 6(3):373–386, 2013.