Supervising the Evolution of Web Service Orchestrations using Quality Requirements

Chouki Tibermacine, Tarek Zernadji

Supervising the Evolution of Web Service Orchestrations using Quality Requirements

Chouki Tibermacine[1] and Tarek Zernadji[2]

[1] LIRMM, CNRS and Montpellier-II University, France
[2] Computer Science Department, University of Biskra, Algeria
Chouki.Tibermacine@lirmm.fr , zernadji@yahoo.fr

Abstract. Since many years, Web services have confirmed their status of one of the most pertinent solutions for a given service provider, like Google, Amazon or FedEx, to open its solutions for third party software development. New business logic can be implemented through orchestrations of existing Web services. This helps development teams in capitalizing resources held by the providers of these services. Nonetheless, these service-oriented software architectures, like any other software artifact, are subject to changes during their lifecycle, and thus can undergo an evolution phenomenon. In this phenomenon, it is argued that quality can be weakened after successive changes (Lehman's 7th law of software evolution), and this is mainly due to the lack of architecture documentation and tool support to supervise architecture changes. In this paper, we present an approach to supervise the evolution of Web service orchestrations, with quality requirements considered as a support documentation. First, we show how important design decisions, like the choice of a service-oriented architecture pattern can be formalized as a documentation for the quality they implement. Then, we detail how this documentation can be used to supervise architecture changes. In this way, the impact of changes made on a software architecture are analyzed on-the-fly to determine which quality is affected.

1 Introduction: Context and Motivation

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 [14]. In this phenomenon, it is argued that quality can be weakened after successive changes (Lehman's 7th law of software evolution [14]). This is mainly due to: i) the lack of architecture documentation that can be used by developers to better understand the design decisions made on the system, and ii) the lack of tool support to supervise architecture changes.

In this paper, we present an approach to supervise the evolution of Web service orchestrations, with quality requirements considered as a support documentation. First, we show how important design decisions, like the choice of a service-oriented architecture pattern can be formalized as a documentation for the quality they implement. Then, we detail how this documentation can be used to supervise architecture changes. In this way, the impact of changes made on a software architecture are analyzed on-the-fly to determine which quality is affected.

In the following section, we show an example that illustrates the problems which we tackle in this paper. In Section 3, we expose the overall approach that we propose in this paper to solve the identified problems. Then, in Section 4, we detail how the quality documentation is specified in our approach. This documentation is used by an evolution assistance algorithm, which is described in Section 5. Section 6 illustrates the use of our approach through an example. Before concluding this paper, we present in Section 7 the related work.

2 Illustrative Example

In this section we show, through an example of a Web service orchestration – a BPEL (Business Process Execution Language [17]) process, how some evolution scenarios can have consequences on quality requirements. Let us suppose that we have an Appealed Assessments System [9] built using Web services. This system is responsible for managing and enforcing policies that pertain to private companies involved with the forestry and lumber trade. Briefly, this service is dedicated to producing a range of reports related to already assessed claims that have been successfully or unsuccessfully appealed.

The decisions made by the architects to design this service-oriented architecture involved the use of a `Data Controler` service. This one provides all the logic required to fulfill the capabilities of the assessment reports service. It allows access up to six different repositories in order to gather all of the required data depending on the requested reports.

Furthermore, the architects observe that they may have to access additional databases in the future and therefore the `Data Controler` service have to undergo some changes. This results in a portability problem in the service architecture design. Consequently, the architects decide to design a facade service (architecture decision AD1) using the service facade pattern [9], to ensure the portability requirement (quality attribute QA1) into the service architecture.

The facade service is named `Data Relayer`, and its role is to receive service consumer requests, relay them to the `Data Controler` service, and then relay the responses back to the service consumer. The facade service ensures the adaptation between the message format used by the Appealed Assessments Service and the data format managed by the `Data Controler` service. Also, it validates the reports received from the `Data Controler` service. The facade service decouples then the consumers of the Appealed Assements Service from the changes that may occur on the `Data Controler`, and compensates its behavior modifications so that the consumers are not impacted.

Preventing from unauthorised access to the ressources of the Appealed Assessments service, the architects decide to establish a service account (based on the *Trusted Subsystem Pattern* [9] (AD2)) to secure the service from direct access to the databases. The security requirement (QA2) is thus defined and implemented inside the whole service orchestration.

Let us assume now, that the maintenance team receives two changes requests. The first concerns the performance enhancement of the overall service (Appealed Assessments Service). So, the developer team proceeds by short-circuiting the authentication service account in the orchestration (removing simply the invoke activity). The architect performing this change, ignore that the usefulness of that service was also (as imposed by the Trusted Subsystem pattern) to prevent direct access to the service resources from malicious attackers. Therefore, this change breaks AD2 and thus the quality requirement that it implements (QA2).

Over the years, the company providing these services has significantly expanded, and consequently more users requested the Appealed Assessment Service. In such a highly concurrent environment, the service may manage a large amount of data and thus increasing resources consumption which may compromise the overall service performance and availability. The architects realize that the service-oriented architecture has to adapt to this scalability requirement problem keeping the performance of the service unaffected. Hence, they examine different kinds of data used by the service and find out that, the policy data remains frequently unchanged during the working days. Therefore, the architects decide (AD3) to use the Partial *State Deferral Pattern* [9] to temporarily hold a copy of data on a local database server increasing the performance of the service (QA3). This scenario implies some architectural changes: first, invoking directly the Data Controler after an authentication through the service account service; second, designing a new standardized contract for an archival service that takes the responsibility of populating the data in the database server; finally, for performance optimization purpose, moving the functionality of the Data Relayer service into the Archival service which results in removing the later from the architecture. This breaks the facade design pattern (AD1) and causes the loose of the portability quality requirement.

3 Proposed Approach

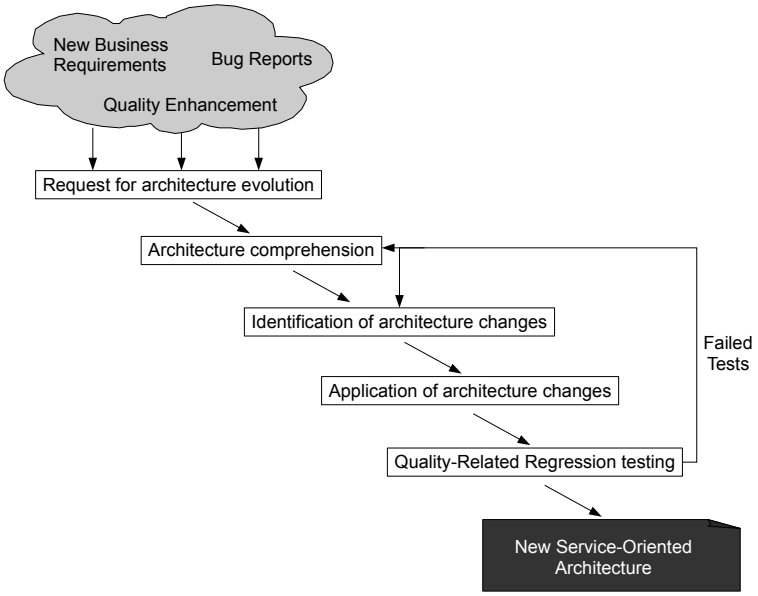

Fig. 1. A Micro-Process of Architecture Evolution

Figure 1 shows a simple micro-process of service-oriented architecture evolution[3]. 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 perform some testing to check if there is a "clean" progression (verify if the additional services, operations or activities work correctly) and no regression (existing features are not negatively impacted by the additions). In this paper, 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 approach 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 2.

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

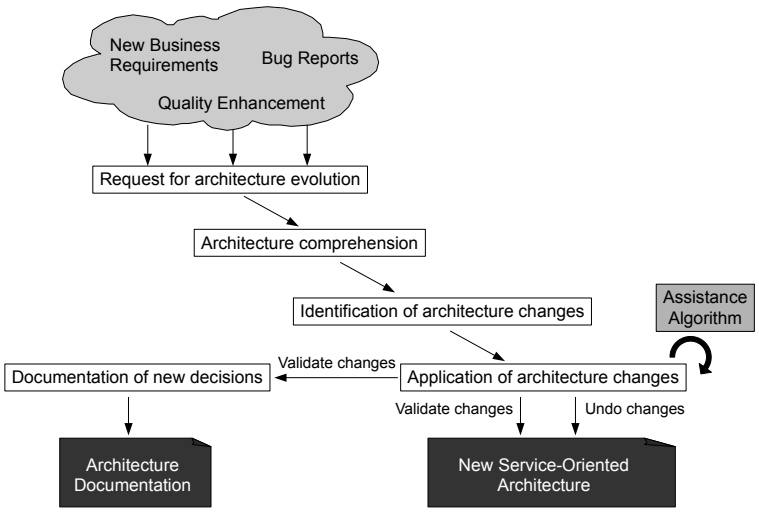

Fig. 2. The Proposed Micro-Process of Architecture Evolution

The approach introduces two concepts: an architecture documentation (bottom left of Figure 2) and an assistance algorithm. The assistance algorithm is used when developers apply changes on an architecture to notify them with the possible impact of their changes on quality requirements. 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. These two concepts are detailed in the following two sections.

4 Architecture Decision Documentation

The concept of architecture decision documentation has been firstly introduced in [19]. In this paper, we present an improvement to the old version of this documentation. 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[4]).

An architecture decision documentation abstracts the links between a given quality attribute and an architecture decision associated to this attribute. Figure 3 shows how these links are organized. We associate to a link a degree of

[4] 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).

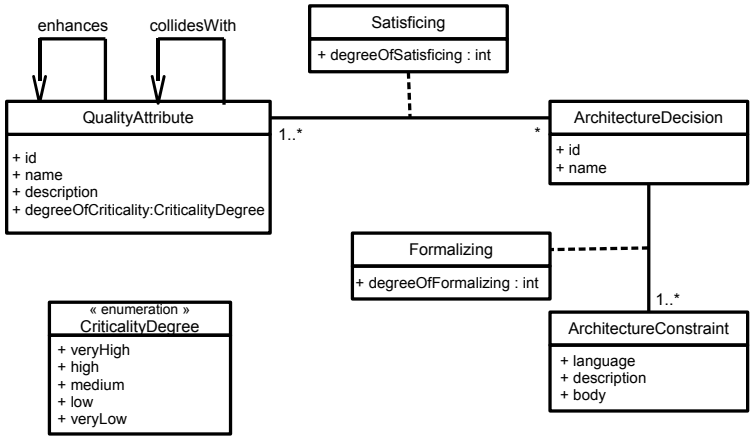

Fig. 3. Links between Architecture Decisions and Quality Attributes

satisfaction. An architecture decision in collaboration with other decisions 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 [9][5], the choice of the MVC pattern [3] 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 decision) and the last architecture decision a 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 architecture decisions to reach a quality goal, as illustrated in the example above.

We voluntarily simplify, in this documentation, the specification of architecture decisions. An architecture decision is thus formalized by an architecture constraint (see the "Related Work" section for richer specifications of architecture decisions). Here again, a formalization degree is a percentage associated to the link between an architecture decision and an architecture constraint. This score represents the extent to which the constraint formalizes the design decision. If we consider that several constraints formalize the same architecture decision, it is possible for the developer to state how the different constraints share the

[5] This pattern is originally inspired from [10].

formalization of the design decision. 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 architecture decision would be equal to 100%. The constraints written in a given documentation are defined with a predefined constraint language.

A quality attribute in this documentation is a non-functional property representing an ISO 9126[6] 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 [7]) which is specified by developers and represents the importance of this quality attribute in the architecture. Its possible values are: very high, high, medium, low and very low.

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 efficiency 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 an architecture contract.

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 4.

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 efficiency 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 efficiency 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,

[6] 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

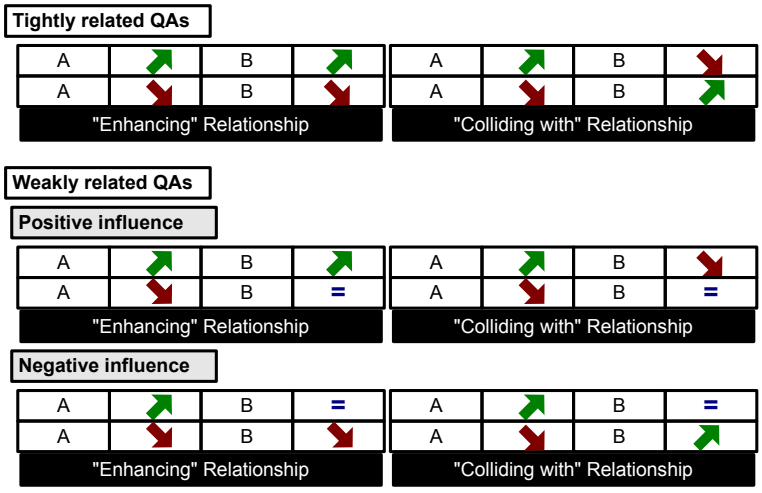

Fig. 4. Relationships between Quality Attributes

in the presence of a unique 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 influences on the second one; however in the other case, it is its weakness which produces an effect on the second. This is shown on Figure 4

In the current implementation of architecture decision documentation, architecture constraints are specified using a modified version of OMG's OCL [18]. An architecture constraint in this language navigates in a metamodel of BPEL Web service orchestrations, but apply to only one instance of that metamodel (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

Once this table validated by the project manager, the assistance algorithm can exploit it in accompanying developers in the architecture change step.

5 Change Assistance Algorithm

During architecture change, the information encapsulated in the architecture documentation is exploited by an assistance algorithm in order to assist developers. The main purpose is to drive software architecture evolution to a situation where the initially required quality is minimally affected. This algorithm is presented throughout this section as several functions.

```
(01)  algorithm ArchitectureChangeAssistance {
(02)    let AE := Architectural Element
(03)    and AD := Architectural Decision
(04)    and AC := Architectural Constraint
(05)    and QA := Quality Attribute
(06)    and AT := Architecture Tactic //a couple composed of a QA and an AD
(07)    and Doc := architecture documentation associated to changed AE
(08)    and affectedQAs := { } //an empty set
(09)    function main() {
(10)      on RequestForAssistance {// an event listener
(11)        for-each (AT in Doc) {
(12)          QA := QA in AT
(13)          AD := AD in AT
(14)          checkArchitecturalConstraint (AD)
(15)        }
(16)        let newAD := ask for AD associated to the new architecture, if any
(17)        if(newAD != null) let newQA := ask for the QA associated to newAD
(18)        addNewArchitecturalTactic (newAD,newQA)
(19)      }
(20)      checkAffectedQAs ()
(21)    }
(22)  }
```

During the step of architecture change application (Figure 2) the developer can ask for an assistance. This triggers the listener on Line 10 in the algorithm above. The algorithm starts first by looking for the architecture documentation associated to the architectural element (the orchestration or the Web service description) which has been changed. Then, the algorithm checks each constraint in the documentation (by calling a function which is detailed in the following paragraphs). After that, the developer is asked to pinpoint the architecture decision and the quality attribute associated to the changes, if any (Lines 16 and 17). At last, if the changes generate a new architecture decision, the algorithm try to

add, to the documentation, the couple composed of this new decision associated to its quality attribute, which is called in this work an architectural tactic (Line 18). In addition the algorithm tries to infer the quality attributes affected by this new tactic (Line 20).

The function detailed in the listing below, checks the constraints associated to a given architecture decision 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 developer. The displayed information includes the architecture decision, the precise 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 11 to 14 in the algorithm below). In addition this function shows to the developer the list of quality attributes which are eventually impacted by this change (Lines 15 and 16). For doing so, it uses the table of relationships between quality attributes presented in the previous section. It limits the selected quality attributes to the ones which are tightly coupled with an "enhacing" relationship. This ensures the selection of the most pertinent quality attributes in this situation.

```
(01)  function checkArchitecturalConstraint (AD) {
(02)    for-each(AC associated to AD)
(03)      let result := check AC
(04)      if (result == false) {
(05)        AE := AE in the context of AC
(06)        QA := QA associated to AD
(07)        warn "The following architecture decision " +AD+" is affected."
(08)        warn "This concerns the architectural element: "+AE
(09)        warn "The affected architecture decision is formalized
(10)        by the constraint up to " + degreeOfFormalization (AD,AC)+ "%"
(11)        warn "The affected architecture decision is satisficing "+QA
(12)        + " up to " +degreeOfSatisficing(AD,QA)+"%"
(13)        warn "The degree of criticality of this QA is: "
(14)        + degreeOfCriticality(QA)
(15)        warn "Other QAs may be affected. This concerns: " +
(16)        QA_Relationships (QA,"enhances", "tight")
(17)        ask to validate the new architecture or undo changes
(18)        according to the warnings above
(19)        if(new architecture maintained) {
(20)          affectedQAs := affectedQAs + QA
(21)          + QA_Relationships(QA, "enhances", "tight")
(22)          warn "Architecture documentation will be changed ..."
(23)          Doc := Doc - AT(AD,QA)
(24)          ask to review satisficing degrees of ATs related to
(25)          QA_Relationships(QA, "enhances", "tight")
(26)          ask to review Non-Functional Requirements specification
(27)        }
(28)      }
(29)    }
(30)  }
```

Then, the developer is asked to validate the new architecture fully aware with the possible consequences of her/his changes, or to undo changes. In this last case, the architecture documentation should be updated by the algorithm (this is the second important role of this assistance algorithm). The affected decisions and their associated quality attributes are removed from the documentation (Line 23 in the algorithm above). 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(...)` 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 04 in the listing below). 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 (Lines 06 – 07). At last, the algorithm asks the developer to change the quality requirements specification.

```
(01)  function addNewArchitecturalTactic (AD,QA) {
(02)    newAT := new AT(AD,QA)
(03)    if (QA is in affectedQAs)
(04)      affectedQAs := affectedQAs - QA
(05)    else {
(06)      warn "Other QAs may be in conflict with "+QA+": "
(07)      + QA_Relationships (QA,"collidesWith","both")
(08)    }
(09)    warn "Architecture documentation will be changed ..."
(10)    Doc := Doc + newAT
(11)    ask to change Non-Functional Requirements specification
(12)  }
```

The last function (see below) 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.

```
(01)  function checkAffectedQAs () {
(02)    if (affectedQAs <> {} ) {
(03)      for-each (QA in affectedQAs) {
(04)        warn QA + "is still affected by your changes"
(05)        ask to review satisficing degrees of ATs implying QA
(06)      }
(07)      ask to change Non-Functional Requirements specification
(08)    }
(09)  }
```

The overall goal of this algorithm is twofold. 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 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.

6 The Proposed Approach in Practice

In this section, we show an example of an architecture documentation and its use by the evolution assistance algorithm. Let us take the example of Section 2. 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 guarantees the Portability quality requirement by using
    a Service facade pattern
    - Quality-Attribute name="Portability" degreeOfCriticality="high"
    - Related-Quality name="Performance" relationship="CollidesWith"
                              relationType="tight"
    - Architecture-Decision name="Service facade pattern"
                                       degreeOfSatisficing="90"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="80"

2. Architecture-Tactic :
    This tactic ensures the Security quality requirement by using
    a Trusted subsystem pattern
    - Quality-Attribute name="Security" degreeOfCriticality="very high"
    - Related-Quality name="Availability" relationship="Enhances"
                              relationType="weak" influence="negative"
    - Architecture-Decision name="Trusted subsystem pattern"
                                        degreeOfSatisficing="70"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="90"

3. Architecture-Tactic :
    This tactic ensures the Performance quality requirement by using
    a Partial state deferral pattern
    - Quality-Attribute name="Performance" degreeOfCriticality="high"
    - Related-Quality name="Security" relationship="CollidesWith"
                               relationType="tight"
    - Architecture-Decision name="Trusted subsystem pattern"
                                        degreeOfSatisficing="60"
    - Architecture-Constraint profile="BPEL" degreeOfFormalizing="70"
```

The architecture documentation contains three architectural tactics. They document the links between architectural decisions (AD1, AD2, AD3) presented in Section 2 and their corresponding quality attributes (QA1, QA2, QA3). 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 first tactic[7], the `Related-Quality` element shows that the portability and performance quality attributes are colliding and are tightly coupled.

[7] We recall that a tactic is the couple composed of an architecture decision and its quality attribute.

Let us see now the use of the assistance algorithm, given the evolution scenario described in the example of Section 2: short-circuiting the authentication service. The assistance algorithm checks the constraints[8] formalizing the architectural decisions for each architectural tactic, and detects that the constraint formalizing the decision AD2 (The Authentication service implementing the Trusted subsystem pattern) was violated. Therefore, the security quality attribute (QA2) is affected. It then notifies the developer that the violated constraint formalizes AD2 up to 90% (an important constraint in the formalization of this decision), the affected architecture decision satisfies Security up to 70% with a very high degree of criticality, and that the security is weakly coupled to the availability quality attribute by an enhancement relation with a negative effect. This means that the availability quality attribute is directly affected by this change. Based on this notification, the developer decides to abort the change she/he made, aware that short-circuiting the authentication service makes the service not secured.

The second change to the orchestration consists of adding a new functionality service namely the Archival service which leads to remove the Data Relayer service. This change aims to improve the performance and the availability of the service. The algorithm detects that the constraint representing the decision AD1 (service facade pattern) does not hold any more, which means that the portability quality attribute (QA1) is affected. Hence, it informs the developer that the violated constraint formalizes AD1 with a degree of 80%, satisfies QA1 up to 90% which has a high degree of criticality, and that QA1 and QA3 (performance) are colliding but tightly coupled. The developer concludes that wanting to improve the performance of the service she/he will probably loose the portability quality attribute, and therefore, decides that performance is more important than portability and validates the change. The corresponding tactic of the affected decision is removed from the documentation and a new tactic is added. The developer is invited to update the NFRs specification and to review the satisficing degrees for the affected qualities. These information serve for possible changes that may occur on the service architecture in the future.

7 Related Work

In the literature, there are many works on the documentation of architeure design decisions. Clements et al. in [6] present an approach which provides a framework for documenting different views of a software architecture. The authors propose a template for architecture description encompassing the documentation of architectural decisions. In [20], Tyree and Akerman discuss the importance of documenting architecture decisions and their specification as first-class entities in architecture description. They present a template specifically designed for architecture decision documentation, which embeds interesting information characterizing architecture design decisions (`status`, `assumptions`, `implications`,

[8] For reasons of space limitations, constraints are not presented here. They are defined using a modified version of OCL and navigate in a metamodel of BPEL.

related artifacts, constraints, ...). Philippe Kruchten introduced a taxonomy of design decisions [13]. He presents a model for describing architecture decisions, including rationale, scope, state, history of changes, categories, cost and risk. He identifies in this ontlogy the different possible relationships between design decisions and links between design decisions and design artifacts. In [11], Jansen and Bosch present a new way of building software architectures. They propose to define these design models as a composition of architecture design decisions. The authors introduce a model for architecture design decisions, including a description, the rationale, the design rules, the design constraints, the consequences, the pros and cons. In [4], the authors proposed a way to characterize architectural decisions. They defined attributes to describe architectural decisions by separating mandatory and optional attributes according to their degree of importance. The first class introduces information associated to architectural decisions that must be defined throughout the system life cycle, including a decision name and description, the constraints, the dependencies, the status, the rationale, the design patterns, the architectural solution, and the requirements. The second class provides additional information that can be choosen 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.

As in these approaches, our work proposed a new way to document architecture decisions. But contrarily to these works, we focused on the use of this documentation in the architecture change assistance. These works can complement our proposed documentation in order to make it richer.

Many works have been proposed on Non-Functional Requirements capture and specification. One of the major works in the literature is that of Mylopoulos et al. [16]. 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. The authors consider Non-functional requirements as goals to be achieved by validating the right design decisions and their rationale, considered in turn as goals. In [8] Cyneirios et al. 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. Bass et al. [1], proposed ADD method (*Attribute-Driven Design*) that follows an architectural design process guided by quality requirements. It uses the concept of attribute primitives, which are collections of components and connectors collaborating to satisfy some quality attributes. These attributes are documented as general scenarios. In [2], the authors proposed Ar-

chitectural tactics, in the same spirit as the primitive attributes to guarantee quality characteristics in software architectural design. Kim et al. [12] presented an approach for representing NFRs in software architecture using architectural tactics as reusable architectural building blocks. The later and their relationship are represented as *Feature Models* and their semantics is defined with the RBML language (*Role-Based Metamodeling Language*). Architectural tactics satisfying quality attributes are selected and composed into one tactic encompssing all the desired qualities. The resulting tactic is then instantiated to create a software architecture that incorporates NFRs for the system under development. In [15], the authors present an approach inspired from [16, 5]. 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.

All these approaches focus on the design stage of software development. Our work is complementary to these approaches, since it addresses a stage which is situated downstream in the development process, the evolution stage.

8 Conclusion and Future Work

Since some years, architecture decision and design rationale are two topics which received a lot of attention from the software architecture research community. In this paper, we proposed an approach which provides: i) a language to document a basic form of architecture decisions as architecture constraints, and the rationale of these decisions, which are quality attributes, together with some fine-grained information about the relationships between these two concepts; ii) a method which makes operational this documentation through its use during architecture evolution; and iii) 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 and consequently identify the affected quality requirements.

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 Web service orchestrations.

On the conceptual aspect, we plan in the near future to separate functional from non-functional evolution in the assistance algorithm. Indeed, these two imply different considerations. Non-functional (or quality) evolution have direct impact on existing decisions and quality, and the developer has some knowledge about the existing quality requirements. In the case of functional evolution, the developer has a different profile, and should be assisted differently.

On the tool and experimental aspect, we will conduct the creation of a catalog of predefined service-oriented architecture design decisions in order to help the developers in the initial documentation of their architectures. This will be based on existing works on patterns or quality models for service-oriented architectures, among others.

References

1. L. Bass, F. Bachmann, and 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.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.
3. F. Buschmann, M. R., H. Rohnert, P. Sommerlad, and M. Stal. *Pattern Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.
4. R. Capilla, F. Nava, and J. 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.
5. L. Chung, B. A. Nixon, E. Yu, and M. J. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 1999.
6. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures, Views and Beyond*. Addison-Wesley, 2003.
7. P. Clements, R. Kazman, and M. Klein. *Evaluating Software Architectures, Methods and Case Studies*. Addison-Wesley, 2002.
8. L. M. Cysneiros and J. C. Sampaio do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. *IEEE TSE*, 30(5):328–350, 2004.
9. T. Erl. *SOA Design Patterns*. Prentice Hall, 2009.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Sofware*. Addison-Wesley Professional Computing Series. Addison Wesley Longman, Inc., 1995.
11. A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of of the 5th IEEE/IFIP WICSA'05*, 2005.
12. S. Kim, D.-K. Kim, L. Lu, and S. Park. Quality-driven architecture development using architectural tactics. *Elsevier JSS*, 82(8):1211–1231, August 2009.
13. P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *Proceedings of the 2nd Groningen Workshop Software Variability*, pages 54–61, 2004.
14. M. Lehman and J. F. Ramil. Software evolution. *Marciniak J. (ed.), Encyclopedia of Software Engineering, 2nd Ed, Wiley*, 2002.
15. T. Marew, J.-S. Lee, and D.-H. Bae. Tactics based approach for integrating nonfunctional requirements in object-oriented analysis and design. *Journal of Systems and Software*, 82(10):1642–1656, 2009.
16. J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE TSE*, 18(6):483–497, June 1992.
17. OASIS. Web services business process execution language version 2.0. Website of the Organization for the Advancement of Structured Information Standards: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html, 2006.
18. OMG. Object constraint language specification, version 2.0, document formal/2006-05-01. Object Management Group Web Site: http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, 2006.
19. C. Tibermacine, R. Fleurquin, and S. 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.
20. J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, March/April 2005.