Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 281 (2011) 113–126

www.elsevier.com/locate/entcs

A Model Driven Approach to the Analysis of Quality Scenarios within Self-Adaptable SOA Systems

Boris Perez[1] Dario Correal[2]

Department of Systems and Computing Engineering
University of Los Andes
Bogotá, Colombia

Abstract

Self-adaptive behavior is a feature which architects needs to include in their systems in order to improve its reliability. However, despite several ways to get it, it is still hard to implement a self-adaptive system focused on non-functional properties. Difficulties to express quality attributes in the system without combining business logic with the self-adaptation logic and to include new services on runtime are some of them. In this paper we propose a model-driven analysis approach to offer a mechanism which allow the desired quality requirements to be expressed in a simple and non-intrusive manner, to find the best services available in a system and, to offer a code generation mechanism which takes the models created under the first objective and generates the necessary code for autonomously monitoring and adapting a SOA system.

Keywords: MDA, Self-Adaptation, Software Quality, Service Selection

1 Introduction

The Service-Oriented Architecture (SOA) architectural style is one of the most used in the development of distributed applications [22]. Due to this, everyday we have more and more available web services offering the same or similar functionality. It becomes a necessity to consider not only functionality but also the quality (non functional properties [7]) as an important factor for selecting the best service [11]. Bass [1] proposes the use of quality scenarios to precisely define the application quality attributes.

The quality of the service offered by web services is becoming a high priority for their suppliers and/or providers. To achieve this, there are some questions that must be answered such as what to measure, how to do so, who should do it and where it

[1] Email: br.perez41@uniandes.edu.co

[2] Email: dcorreal@uniandes.edu.co

doi:10.1016/j.entcs.2011.11.029

should be done [7]. To this, we must sum up the optimization strategy definition to find the best service from a set of quality combinations given by a client [21]. These combinations are expressed during the design stage through quality scenarios [9]. However, during runtime, it is not always possible to guarantee the compliance of these quality scenarios, due to the fact there are still externals factors such as failures in services offered by 3rd parties, or defects in legacy applications that implement the supplied services.

One way of reacting to these incompliances or failures is to stop the system to make the necessary repairs. However, this operation is not always possible, or desired, because there are some critical business processes that cannot be stopped [12]. When this happens, what we would like is that the system be capable of detecting these faults and fix them itself, i.e. replacing the problematic service with another that complies with both functional, and quality requirements, without human intervention. In other words being self-adaptable. In accordance with [5], "a self-adaptive system evaluates its own behaviour and changes behaviour when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible".

1.1 Problem

There are different alternatives offering solutions to this need. One being through the use of a Middleware with reflective capacity. According to [23], while this solution provides fundamental mechanisms successfully implementing highly adaptable systems, it is still an expensive and difficult means of making adaptation changes while the system is running, due to them not being able to respond to the "Why, When and What to do" in an adaptation. Another alternative is through the ECA (Event-Condition-Action) rules [15], although the problem with this is that said rules are general, and unqualified to take quality attributes into account. In addition to this, they're embedded in the application code and run in a specific order.

A third alternative consists of the use of exceptions. According to [9], while their use is extended, there is the problem of being highly integrated or embedded into the application at the code level. Exceptions are good for catching an error as it is detected, but are nonetheless weak when it comes to detecting subtle system anomalies, i.e. gradual performance degradation. Finally, as a fourth alternative we have UDDI (Universal Description, Discovery and Integration) [8]. A UDDI registry is a web service dealing with suppliers, implementations and metadata information about the services it hosts. This alternative, according to [16] presents two problems: 1) Unusable links and, 2) Discovery of services applicable only to the functional requirements, that said, it doesn't consider the quality attributes. All these alternatives have one thing in common, the problem that exists when it comes to performing a discovery process with respect to new services on runtime. The available alternatives are generally defined in the code.

From these limitations, we can identify the following problems: **1.** Establishment of what system component info should be recollected and analysed, with the end of putting the adaptation into place. **2.** Situation definitions needing comply

with quality requirements. **3.** System maintainability due to the mix of both self-adaptability and business logic. **4.** Finding services that comply with established functional and quality requirements. **5.** Service quality information updates, and **6.** the inclusion of new services offering a determined functionality.

1.2 Proposal

With the end result of offering a proposal to help resolve the previously mentioned problems, we present our proposal, a model driven approach to the analysis of quality scenarios within self-adaptable SOA systems. This proposal is a method allowing the software architect, through the use of model driven strategies, to analyse and express quality needs within a SOA system, with the purpose of finding the best service for said needs. The latter is achieved by means of a device that monitors and controls participating services with the SOA solution. Our proposal starts from the architecture model and ends with the code generation needed to run.

Our proposal is responsible for: 1) Finding the best service among a set of alternative services, those that meet our quality needs. 2) Calculating the quality information of the alternative services, and 3) allowing the inclusion of new services that provide the required functionality.

1.3 Paper Structure

This paper is organized as follows. Section 2 deals with the self-adaptability concept. Section 3 gives an introduction to the case study used to validate the given proposal. Section 4 gives our proposal, explaining the characteristics and functionality, from development to execution. Section 5 shows our experiment to sustain the proposal and its results. Section 6 presents various other proposals for the same thing based on quality attributes and their respected authors, and finally Section 7 concludes with our results and termination of our proposal.

2 Self-Adaptability

Self-adaptive systems, according to [19], [14] and [6], adapt themselves to changes in running conditions with a minimum, or no human intervention, with the means of providing reliability, robustness and availability. The running conditions refer to all elements observable by the system, such as user-inputted information, external hardware elements or program rules. The key action to this kind of systems is the fact the life cycle should not be stopped after deployment. Its life cycle should continue and be capable of constantly evaluating and responding to changes presented [18].

Keeping these systems working for a long time requires collecting information reflecting current system state, analysing said information in order to diagnose the problems or detect faults, followed by finding a resolution and then acting on it to mitigate the problem. This current system state knowledge comes thanks to feedback loop cycles [2], also known as, closed loop, which allows feedback about

the current happenings of the application, and its running environment [18].

2.1 Decision-Making Process in Self-Adaptive Software

According to the definition given by Salehie [17], the decision-making process for selecting an adaptation strategy in a self-adaptive system based on quality attributes can be defined as a set $<R,\ G,\ D,\ U>$, where Request (R) is the reason for the change being demanded (violation of a quality attribute). Goal Repository (G) corresponds to a deposit in the system containing the required quality properties (quality scenarios). Domain (D) corresponds to a deposit with structural information and information about the behaviour of the software that is to be adapted. Finally, Utility (U) is a repository with information about stakeholders' preferences for carrying out an adaptation, i.e. before two equally valid validation options, of which should one should prevail the other. This way, such as is proposed in [17], given the entries $<R,\ G,\ D,\ U>$, the problem makes a decision satisfying the objectives defined by the Goal Repository (G).

3 Case Study

In this section we will present the case study used to validate the proposal and furthermore, will be employed throughout the paper. It's based on a study made by InAlpes (Inmobiliaria Alpes), a real estate agency, who would like to implement a SOA based system. Due to the company's business objectives, an automation process strategy has been raised to be used intensively, as what they require high availability. This system is expected to be used on a large scale, and it is therefore not viable to make adaptations to the system on-line. In other words, the system should adapt itself if faults are detected, or problems arise with services.

This case study focuses on the Property Registry business process, which consists of the publication of properties that may be leased by a client. Although, we are paying specific attention to the Landlord Risk Scoring activity. The objective of this activity is to verify there are no legal impediments in order to accept someone as a provider. This activity is critical, and if faults occur, the system should adapt itself in order to find an alternative service, with a view of guaranteeing the uninterrupted operation of the system.

In Figure 1, you can see the *RiskScoringService A*, which is responsible for verifying client related legal aspects. For this figure, *RiskScoringService B* and *RiskScoringService C* both offer the required functions though with different quality characteristics, derived from this are different technologies, implementation algorithms and deployment platforms.

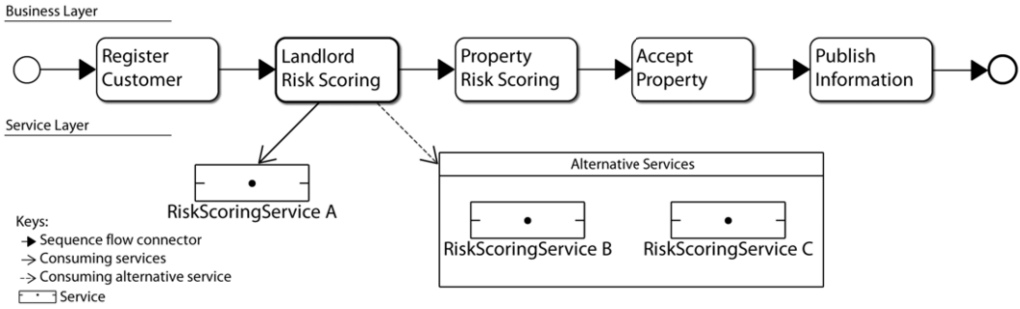

Fig. 1. Landlord Risk Scoring activity with several alternative services

4 Solution Strategy

4.1 General Overview

The aim of this proposal is so that when a service fault happens, starting from a range of services that offer the same functionality, a replacement can be found, one that not only complies with the quality scenarios but also has the best quality of service from this set of services. In general terms, this process is composed of four steps, as presented in Figure 2. In the first step the architect should model the system architecture of which wants to include self-adaptive behaviour. The second consists in defining the adaptation rules to be validated during execution. Step three consists of generating a model to relate the system to be adapted with the aforementioned rules, and finally, step four, generates the self-adaptive code.

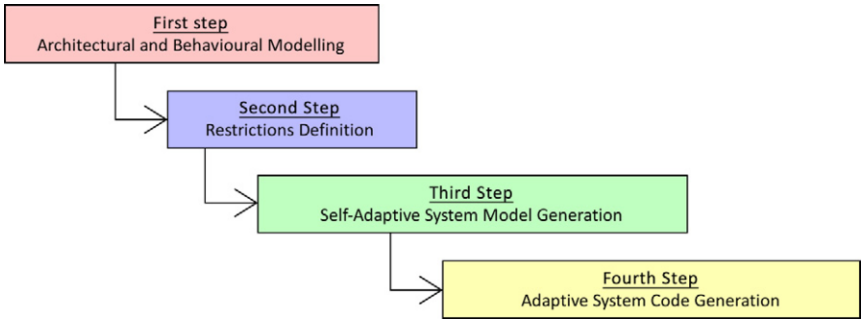

Fig. 2. Solution Strategy

4.2 First step: Architectural and Behavioural Modelling

4.2.1 Architectural Modelling–Domain Repository (D)

The *Domain Repository* contains information of the system structure to be adapted. In our case, this repository contains models representing the service architecture of the *Property Registry* business process. The service architecture representation is based on Archivol [3], a metamodel used for defining components involved in a system, the mechanisms they interact with and the architectural style specification (e.g SOA). The main metamodel package used in our proposal was *candidate architec-*

ture. This involves concepts such as candidate architecture view, model, architectural element, component, interface, capability, connector among many others.

In Section 3 we mentioned this paper focuses on the *Landlord Risk Scoring* activity within the *Property Registry* business process for the purpose of limiting its application and verification and get a better understanding of our proposal. With this in mind, Figure 3 shows an architecture model conforming to the Archivol metamodel, it also corresponds to the architecture presented in Figure 1 (Section 3), which shows a service consumer (*Landlord Risk Scoring* activity) and a service provider (*RiskScoringService A*) with its corresponding quality attributes.

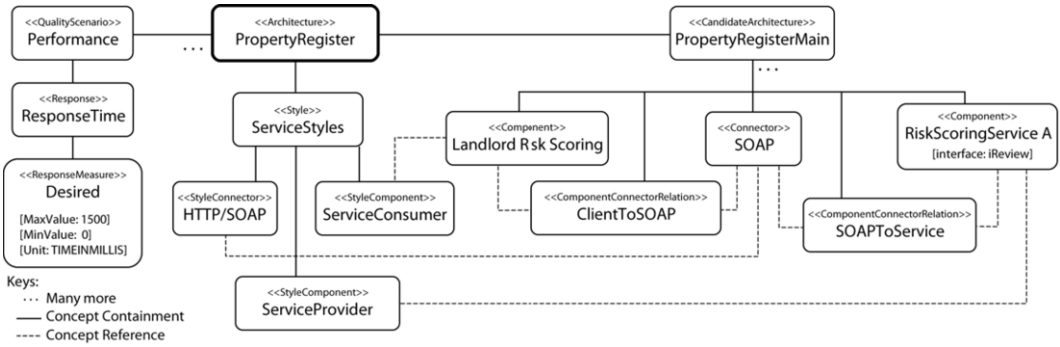

Fig. 3. Architectural Model Conforms to Archivol

4.2.2 Behavioural Modelling–Goal Repository (G)

The *Goal Repository (G)* contains the quality property information (Quality Scenarios) required in the system. These quality scenarios are a specific requirement for a quality attribute, and its definition is not tied to an architecture element. Its validation on any architecture element will be defined in the Restrictions Definition (Subsection 4.3), and carried out on runtime. The quality scenarios could also be described using Archivol. In this case, it makes use of the *architecture* and *quality* packages, of which include *quality requirement, metric, interaction and quality attribute* concepts.

4.3 Second Step: Restrictions Definition

The objective of this second step is to define the self-adaptation rules governing the behaviour of the new system. The architect should think about the quality scenarios required to control every consumer-provider relationship, that is to say, determined architecture elements. The implicit action with a rule violation consists of finding a new service offering the same functionality and furthermore, one that accomplishes the same modelled quality scenarios. In the context of the model defined by [17] (see Subsection 2.1), the adaptation rules use the information contained in the Goal Repository (G).

To facilitate the self-adaptation rules description, a specific domain language was defined known as *Self-Adaptability Definition Language (SADL)*. SADL has 3

main objectives: 1) Defining operation variables of the self-adaptive system, 2) associating quality scenarios to the relation between consumer and provider and finally, 3) to facilitate the rule description for a self-adaptable system. The inspiration for the DSL design was based on the ECA (Event-Condition-Action) rules, specialised to consider the quality scenarios during the conditions evaluation. Figure 4 shows a DSL example based on the case study presented in Section 3. In this fragment of the language, we only considered one rule. This model conforms to a metamodel known as *Self-Adaptability Definition Language Metamodel*, which stores information corresponding to the association between adaptation rules, quality scenarios and previously defined architecture services.

```
1  ReliabilitySleepTime : 5000
2  BestServiceSleepTime : 200
3
4  includeScenario availabilityScenario
5  includeScenario performanceScenario
6  includeScenario reliabilityScenario
7
8  rule { rule1ForEvaluarRiesgoPropietario }
9     for serviceConsumer { EvaluarRiesgoPropietario }
10    consuming interface { iCustomer }
11    fulfilling qualityScenarios { performanceScenario ,
12    availabilityScenario } ;
```

Fig. 4. Self-Adaptability Definition Language Example

Line 1 specifies the time, in milliseconds, that the system employs to recalculate the reliability of the service in use. Line 2 defines the time needed to find and assign the best service. Lines 4 to 6 allow us to include the defined quality scenarios within the DSL during the *Behavioural Modelling (Goal Repository)*. Lines 8 to 11 define a rule. Line 8 creates and assigns a name, whereas Line 9 specifies the service consumer. Line 10 associates the interface to be employed in order to identify the relationship between service consumer and the best behaving service. Line 11 specifies the quality scenarios requiring validation for all alternative services offering the interface shown on Line 10. In this case, the services are required to comply with the *Performance and Availability Scenarios*. The idea behind including the quality scenarios on lines 4 to 6 is to offer a auto-completion characteristic when specifying said scenarios within the rule, this way the architect has no need to continuously input the same thing, rather just select it.

4.3.1 Modelling the Utility (U) Repository

The Utility Repository (U) contains information relating to adaptation-objective preferences. According to the DSL, when a failure on the quality attribute scenarios is detected, Alternative Services are evaluated in order to replace the service where the fault has occurred. This selection is made with a set of services implementing the same functional contract of the original service. Our approach gives priority, when an alternative service is being selected, to the greatest number of quality scenarios

that the particular alternative service complies with. If faced with two alternative services that comply with the same number of quality scenarios, selection will be based on the order in which the service was defined, prioritizing the older ones.

The failure to accomplish the quality scenarios of a service is calculated by comparing the quality service metrics against defined quality constraints on quality scenarios. For example, when the value of the response time for an operation is greater than defined in the quality scenario. Using the fragment of Figure 4, if two services both fulfill the *Performance and Availability scenarios*, the service best complying to the *PerformanceScenario*, as this is the defined first by the rule.

4.4 Third Step: Self-Adaptive System Model Generation

In this third step we proceed to generate a model containing the components needed for the generation of a self-adaptable system. This is an automatic activity, and henceforth doesn't require any kind of human intervention. This model is conforms to a metamodel known as *Menta Self-Adaptation Metamodel* and can be seen in Figure 5. The reason for developing this middle model between the needs of the architecture definition and the code generation is to take advantage of the independent concept of the technological-platform offering by a model based development.

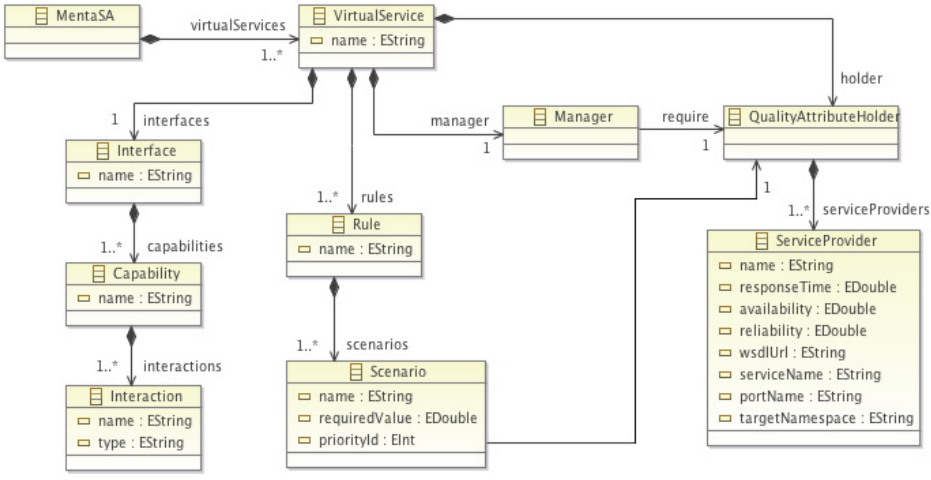

Fig. 5. Menta Self-Adaptation Metamodel

The Self-Adaptive system model performed a join process between both Domain Repository and Goal Repository components. This model introduces one element that is fundamental to facilitating system self-adaptation, namely *Virtual Service (VS)*. A VS can look like a membrane to cover services used to define an adaptation rule. The VS seeks to free the consumer from a specific service. Additionally, the VS handles the search for better services during execution and constantly monitors the compliance of the quality scenarios of said service. If there is a violation, the VS validates the adaptation rules and replaces the violating service with a viable alternative.

4.5 Fourth Step: Adaptive System Code Generation

Our fourth and final step consists of taking the model generated in step three and generating the application source code to a particular platform, in our case Java.

The model's most important component the code generation process is the Virtual Service. Each VS element is implemented in the form of dual components, called Membrane and Validator. The Membrane is generated as a web service that has two objectives. The first is to deal with service user requests, while the second is to ask the Validator component for the reference of a specific service to use in order to answer to a request. When the Validator component returns the service reference, the Membrane is responsible for routing the request to this service through HTTP. Thus, becoming transparent to the consumer on the service he is using. The Validator component is responsible for selecting the service which best adheres to the quality scenarios at a given moment and besides, it best meets those requirements from the set of alternative services. To execute the service selection, the Validator saves information about the status of each available service that implements a particular business interface. The Validator performs a periodic scan of each service in order to determine their respective statuses - for example, if the service is alive and responds within the time limits established. The periodical scan done by this component is shown in Figure 4 Line 2 (*BestServiceSleepTime : 200*). The Validator allows the inclusion of a new service that implements a business interface. This business interface will define the operations required to support a feature. Thus, all alternative services must offer at least the operations described in the interface. This verification is done through signing the methods. During execution, it may add new services on condition that they provide at least the operations in the interface. In Figure 6 we can see the functionality of our proposal whilst running.

The VS is deployed as a web service and acts as a service directory although decentralized, since it already creates one per interface required and defined by the system architecture. The VS encapsulates all the necessary logic in order to deliver an answer within the required quality conditions to the client, it also comprises of the operations they need. These same operations should be included in the alternative services required to implement the same VS interface.

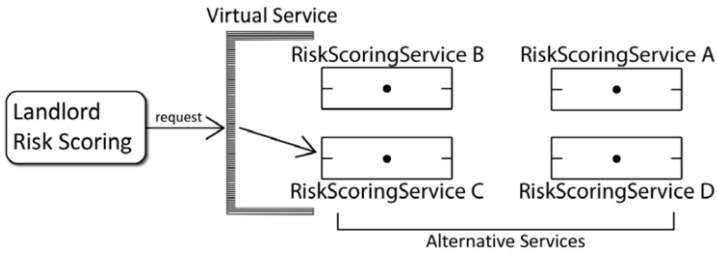

Fig. 6. Menta Operation

4.5.1 Quality Attributes

The process of ensuring a service fulfils the quality scenarios consists of the resulting value calculate from its quality attributes. This proposal concentrates on the uncertain quality attributes, those that are constantly changing. These uncertain attributes are *Response time*, *Availability* and *Reliability*.

The *Response time* (Q_{rt}) measures the time (in milliseconds) from the moment a VS receives a request up to when the result is sent to the client. The *Availability* (Q_{av}) measures the probability of the service being accessed within the given time period. Finally, the *Reliability* (Q_{re}) measures the probability of the requests being answered within the given time period. These values are calculated and updated per service. This way, the *Validator* component revises n services using $<$*ServiceId,* Q_{rt}, Q_{av}, Q_{re} $>$ in order to determine the service is complying with the quality scenarios described in the rules, and furthermore, it is the best possible alternative.

4.5.2 Internal State Monitoring

Our proposal includes a monitoring element designed to inform us of the quality characteristics of alternative services. This allows the user to know what they are connecting to, what quality attribute values it has and what services are available for use and also allowing them to activate and deactivate the *Validator* and finally add or remove new or undesired services.

This user interface allows us to: 1) View the list of rules defined by the VS, 2) view the list of scenarios associated with each rule, 3) view the encapsulated services list along with their respected quality attributes, 4) add or Remove Services and 5) allow the activation/deactivation of the Validator component.

5 Experimentation

For our experiment we used the case study presented in Section 3. The first step consists of modelling the business process using Archivol, together with the specification of the two quality scenarios. These are represented in Figure 7.

For our validation, we only identified the *Landlord Risk Scoring* activity as self-adaptable. The second step, restrictions definition, was developed using the DSL described in Subsection 4.3 and can be seen in Figure 8. These scenarios will affect all alternative services to be evaluated. Once the first two steps have been achieved, we can then proceed to generate the self-adaptive system model and its later transformation to code.

This experiment was done in order to demonstrate the fact the VS is capable of handling the system reliability required to involve more than one quality scenario in order to choose the best service. The experiment specifically implies that for t time, services RiskScoringService A, RiskScoringService B and RiskScoringService C fulfil just as much functionality as the quality scenarios AvailabilityScenario and PerformanceScenario. RiskScoringServices A is in charge of resolving requests. In time $t + 1$, all the services equally fulfil the AvailabilityScenario requirements, and should hence employ the second quality scenario, PerformanceScenario, in order to

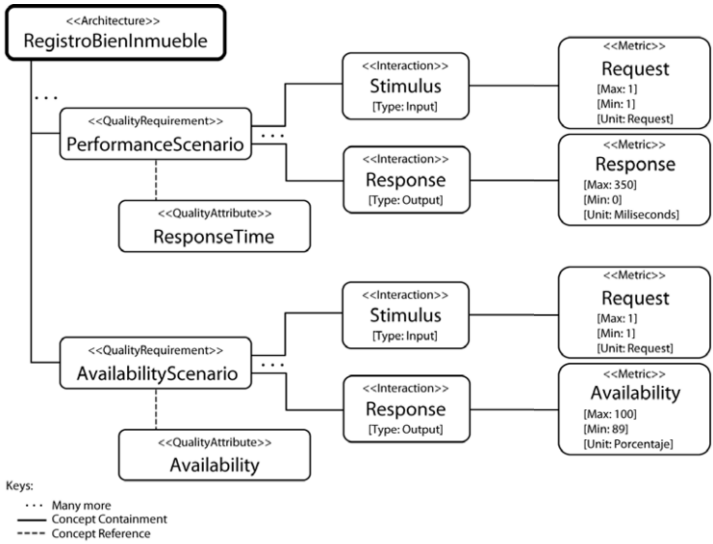

Fig. 7. Quality Scenarios

```
1  includeScenario availabilityScenario
2  includeScenario performanceScenario
3
4  rule { rule1ForEvaluarRiesgoPropietario }
5     for serviceConsumer { EvaluarRiesgoPropietario }
6     consuming interface { iCustomer }
7     fulfilling qualityScenarios { performanceScenario ,
8        availabilityScenario } ;
```

Fig. 8. Self-Adaptability Definition Language Fragment

determine which service best fulfils the needs.

The evaluation consists of a process running in the Bonita BPM process engine. For this process we created two instances, of which each instance launched 55 requests. Those 55 requests correspond to 55 process executions, translating into 110 summonses to the VS in charge of administrating the Landlord Risk Scoring activity's alternative services.

Situation A

Service	TR (ms)	D (%)	C (%)
RiskScoringService A	70	90	100
RiskScoringService B	80	98	100
RiskScoringService C	30	89	100

Situation B

Service	TR (ms)	D (%)	C (%)
RiskScoringService A	70	99	100
RiskScoringService B	80	70	100
RiskScoringService C	30	98	100

Situation C

Service	TR (ms)	D (%)	C (%)
RiskScoringService A	70	100	100
RiskScoringService B	80	100	100
RiskScoringService C	30	100	100

Fig. 9. Changes in Quality Attributes

For the execution of our case case we adjusted the quality attribute values for

services according to Figure 9 (Situation A). The first column (Service) corresponds to the service name; the second (TR) represents the calculated Response Time value; the third (D) saves the calculated value for Availability; the fourth and final column (C) saves the value calculated for the Reliability attribute.

In this situation, our proposal found the three services all fulfil both quality scenarios, and consider them the priority as well as putting the RiskScoringService B in charge of dealing with requests, due to it having the best Availability value. We then simulated the Service B down in order to alter the Availability calculation, and after various scans by the Validator, the values became as seen in Figure 9 (Situation B).

Given this situation, we tested the RiskScoringService B, which initially fulfilled the quality scenario, and now does not. We now have found that RiskScoringService A has become the best, fulfilling the quality scenario priority.

Finally, several runs were made by Validator component to ensure that Availability values for all services was the same, i.e. 100%, just as shown in Figure 9 (Situation C). Given this situation, the VS considers all services as good services for highest priority quality scenario (AvailabilityScenario) and makes the decision conforming to the priorities, that is to say, analysing the Response Time attributes in order to find the best; in this case, RiskScoringService C.

This experiment has achieved the validation of the behaviour against various quality scenarios and changing factors within the quality characteristics of the involved services. The final objective is to give a client the response in the best quality conditions possible.

6 Related Work

This section explores distinct papers, similar to our own.

The work carried out in [13] and [20] uses the ECA style rules [4] to facilitate the implementation of context sensitive applications. These works are focussed on the distribution of responsibilities between context sensitive service platforms, such as broadband or number of clients connected. Put another way, they don't take the quality attributes into account.

The ACRM (Architectural Runtime Configuration Management) approach [10] creates a model capturing the adaptable system configurations and corresponding behaviour, and organises them into a historical graph of configurations. To complement this model along with the adaptable processes metadata, ARCM creates a historic perspective of process adaptations. ARCM also provides active controls to undo an operation or activate a saved configuration. This work is focussed on the administration and visualization of said running adaptations.

In [23] a method based on quality attribute scenarios to find and analyse potential points of self-adaptation in software architecture during the design stage is proposed. Extend an ADL called ABC / ADL, to store the architecture information. Information is used directly by a reflective-based middleware architecture, called PKUAS, for making self-adjustments in implementation. Some limitations in

this proposal are using EJB components. While the use of these components is not really a limitation at the functionality level, it certainly is at a interoperability and scalability one. In a word, this proposal is tied to implementation of technology. This proposal is not clear about the possibilities of including new components. The system design is separate from the implementation, so the implementation may be different from the solutions proposed in the architecture. In practice this should not happen, but it is a risk taken to manage this separation.

7 Conclusions

Here, we have introduced a way of modelling a self-adaptable system inside a service-based architecture. The contribution of our approach is working with models, allowing platform independence; also that of taking quality attributes into account in order to accomplish dynamic adaptations and source code generation. This in other words translates to a shorter response time in order to be implemented.

To achieve this, the architect must model the current architecture, describing the self-adaptation rules that govern the new system relying in the quality scenarios. For this the architect needs to use a DSL equipped for this proposal, to be able to express the quality conditions required during execution. This proposal takes the information provided by the architect and proceeds to generate new system architecture with self-adaptable characteristics, which will then transform for code to generate a ready to use application.

This work is only the first step in achieving self-adaptable systems that take quality attributes into account, without worrying about a pre-existent, and working, system. This is a proposal orientated to SOA systems within an organization, of which the context characteristics, such as bandwidth, channel usage; connection kind and CPU use are irrelevant.

References

[1] Bass, L., P. Clements and R. Kazman, "Software Architecture in Practice, Second Edition," Addison Wesley, 2003.

[2] Brun, Y., G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè and M. Shaw, *Software engineering for self-adaptive systems*, Springer-Verlag, Berlin, Heidelberg, 2009 pp. 48–70.
URL http://dx.doi.org/10.1007/978-3-642-02161-9_3

[3] Correal, D., *Model oriented software architectures* (2011).
URL http://moosas.uniandes.edu.co

[4] Costa, P. D., L. F. Pires and M. V. Sinderen, *Architectural patterns for context-aware services platforms*, in: *in Proceedings of the Second International Workshop on Ubiquitous Computing (IWUC 2005*, 2005, pp. 3–19.

[5] DARPA, *Self adaptive software*, DARPA, BAA 98-12, Proposer Information Pamphlet (December, 1997).

[6] Di Nitto, E., C. Ghezzi, A. Metzger, M. Papazoglou and K. Pohl, *A journey to highly dynamic, self-adaptive service-based applications*, Automated Software Engg. **15** (2008), pp. 313–341.

[7] Diamadopoulou, V., C. Makris, Y. Panagis and E. Sakkopoulos, *Techniques to support web service selection and consumption with qos characteristics*, J. Netw. Comput. Appl. **31** (2008), pp. 108–130.
URL http://portal.acm.org/citation.cfm?id=1351191.1351418

[8] for the Advancement of Structured Information Standards (OASIS), O., *Universal description, discovery and integration (uddi)* (2005).
URL http://oasis-open.org/committees/uddi-spec

[9] Garlan, D. and B. Schmerl, *Model-based adaptation for self-healing systems*, in: *Proceedings of the first workshop on Self-healing systems*, WOSS '02 (2002), pp. 27–32.

[10] Georgas, J. C., A. v. d. Hoek and R. N. Taylor, *Using architectural models to manage and visualize runtime adaptation*, Computer **42** (2009), pp. 52–60.

[11] Kulnarattana, L. and S. Rongviriyapanish, *A client perceived qos model for web services selection*, , **02**, 2009, pp. 731 –734.

[12] Lin, K.-J., J. Zhang, Y. Zhai and B. Xu, *The design and implementation of service process reconfiguration with end-to-end qos constraints in soa*, Service Oriented Computing and Applications **4** (2010), pp. 157–168, 10.1007/s11761-010-0063-6.
URL http://dx.doi.org/10.1007/s11761-010-0063-6

[13] Maatjes, N., "Automated Transformations from ECA Rules to Jess: An MDA Approach," VDM Verlag, 2008.

[14] Oreizy, P., M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, *An architecture-based approach to self-adaptive software*, IEEE Intelligent Systems **14** (1999), pp. 54–62.
URL http://dx.doi.org/10.1109/5254.769885

[15] Oriol Hilari, M., J. Marco Gomez, J. Franch Gutirrez and D. Ameller, *Monitoring adaptable soa-systems using salmon* (2008), pp. 19–28.

[16] Ran, S., *A model for web services discovery with qos*, SIGecom Exch. **4** (2003), pp. 1–10.
URL http://doi.acm.org/10.1145/844357.844360

[17] Salehie, M. and L. Tahvildari, *A quality-driven approach to enable decision-making in self-adaptive software*, in: *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07 (2007), pp. 103–104.
URL http://dx.doi.org/10.1109/ICSECOMPANION.2007.13

[18] Salehie, M. and L. Tahvildari, *Self-adaptive software: Landscape and research challenges*, ACM Trans. Auton. Adapt. Syst. **4** (2009), pp. 14:1–14:42.
URL http://doi.acm.org/10.1145/1516533.1516538

[19] Wang, Q., *Towards a rule model for self-adaptive software*, SIGSOFT Softw. Eng. Notes **30** (2005), p. 8.

[20] Wang, Q., *Towards a rule model for self-adaptive software*, SIGSOFT Softw. Eng. Notes **30** (2005), p. 8.

[21] Yu, Q., M. Rege, A. Bouguettaya, B. Medjahed and M. Ouzzani, *A two-phase framework for quality-aware web service selection*, Service Oriented Computing and Applications (2010).
URL http://dx.doi.org/10.1007/s11761-010-0055-6

[22] Yu, T., Y. Zhang and K.-J. Lin, *Efficient algorithms for web services selection with end-to-end qos constraints*, ACM Trans. Web **1** (2007).
URL http://doi.acm.org/10.1145/1232722.1232728

[23] Zhu, Y., G. Huang and H. Mei, *Quality attribute scenario based architectural modeling for self-adaptation supported by architecture-based reflective middleware*, in: *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2004, pp. 2 – 9.