Use case- and Scenario-based Approach to Represent NFRs and Architectural Policies

Claudia López, Hernán Astudillo

Universidad Técnica Federico Santa María, Departamento de Informática
Avenida España 1680, Valparaíso, Chile
clopez @inf.utfsm.cl, hernan @inf.utfsm.cl

Abstract. Software architecture decisions pay primary attention to non-functional requirements (NFRs), yet use cases normally describe functional requirements. This article presents scenario-based descriptions of Architectural Concerns to satisfy NFRs and of Architectural Policies that represent architectural choices to address such concerns. The Azimut framework combines these modeling abstractions with Architectural Mechanisms to enable iterative and traceable derivation of COTS-based software architectures. An example is provided using an inter-application communication problem, and its use in an MDA context is explored.

1 Introduction

Use cases are widely used to describe requirements to be validated by users and used by builders to drive the software development process. Unfortunately, use cases are usually employed to specify functional requirements, yet software architects need information about non-functional requirements (NFRs), such as reliability, performance, stability, etc., since they are much harder to satisfy in large and distributed systems than functional requirements are.

This article presents an approach to employ use cases and scenarios to describe NFRs, relate them to specific functional use case features, and serve as input to the architecture elaboration process. The Azimut framework supports iterative derivation of detailed component architectures from systemic *NFRs*, which are associated to specific *architectural concerns* characterized with *dimensions*; each dimension may be satisfied by some *architectural policies*; each policy may be satisfied by several *architectural mechanisms*; and mechanisms may be provided by available *COTS*. The framework supports traceability of architectural decisions from requirements through COTS-based implementation.

To associate NFRs with specific use cases, we introduce *architectural policy scenarios* to do describe architectural policies at platform-independent level; *concern scenario templates* to specify valid values for each dimension of a given concern; and *architectural policy scenario catalogs* to groups concern scenario templates for each architectural policy scenario. Thus, an architect may instantiate a concern scenario template to describe some required system NFRs, and proceed to derive architectural policies as enabled by the available catalog.

Section 2 provides an overview of related work on use cases and NFRs; Section 3 distinguishes architectural policies and mechanisms, and introduces architectural policy scenarios , concern scenario templates, and architectural policy scenario catalogs; Section 4 describes the proposed scenario-based approach with an illustrative example; Section 5 discusses some further work; and section 6 presents our conclusions.

2 Use Cases and NFRs

Use cases are means for specifying required usages of systems [1]. Typically, use cases have been used to describe functional requirements, leaving NFRs (reliability, performance, stability, etc.) out. However, recent publications have dealt with NFRs at use case level to specify this kind of requirements in a clear standard manner. The approaches vary from initial proposals to well documented techniques of recognized authors in use cases area, but all of them agree that use cases are the proper means to include specification of NFRs in a software development process.

Jacobson and Ng [6] argue that if you can define a test for a particular requirement, then you can define a use case for it. This argument allows to employ use cases to specify NFRs as long as NFRs are understood as requirements imposing behaviors on the system. Since NFRs usually need of some underlying infrastructure mechanisms, they impose some behaviors on the infrastructure; thus, the authors introduce *Infrastructure Use Cases* describing what the system does (infrastructure mechanisms) to meet NFRs (generally related to system quality requirements) to each step of an application use case. The Infrastructure Use Cases are based on teh ⟨Perform Transaction⟩ use case, which is a description of a pattern extending each step of an application use cases (describing functional requirements). With this construct, the architect can introduce infrastructure mechanisms in each step of a use case (through ⟨⟨extend⟩⟩ of Perform Transactions) describing how the system addresses to satisfy NFRs. From infrastructure use cases the approach allows to maintain traceability from infrastructure mechanisms specification to posterior decisions (detailed design, implementation and testing). This approach works properly to describe mechanisms that extend functionality in each step of an transaction in distributed systems (e.g. access control or logging mechanisms), but it doesn't describe well NFRs that specify system-level qualities (e.g. performance and reliability); they must be described as Special Requirements.

Supakkul and Chung [7] point out that NFRs can be included in a use case model, but that it is important to preserve existing principles of employing use cases to describe only functional requirements, and not to represent NFRs. With this outlook, the authors propose *NFRs Association Points*, which are specific points of use cases where NFRs can be associated. The *use case association points* associate NFRs to the described functionality (e.g. performance metrics); *actor association points* to specify NFRs related to external entities (e.g. scalability); *actor-use case association points* to represent NFRs related to interaction be-

tween external entities and a functionality (e.g. communication, user interface); and *system boundary association points* to define NFRs that are global in nature (e.g. portability).

This kind of specification doesn't addresses how to support software development directly (unlike the previous approach, but it supports a goal-oriented approach to selecting mechanisms that satisfy NFRs, called the NFR framework [8].

Bencomo and Matteo [9] propose *Thematic Use Cases* to describe distribution concerns and an approach to maintain traceability from the use case model through analysis, design, implementation and deployment models.

Lastly, Amaya et al. [10] apply these ideas (NFRs at use case level) to Model-Driven Architecture (MDA) [2] to model separate concerns across the development cycle, using cases for requirements (at MDA's CIM, Computation-Independent Model) and subject-orientation and composition patterns for design (at MDA's PSM, Platform-Specific Model); however, they didn't propose a specific approach to generate code that satisfies NFRs.

3 Use case- and Scenario-based Approach

Software architects focus on Non-Functional Requirements (NFRs) more than on functional requirements because the former are much harder to satisfy in large and distributed systems. Key problems of software architecture are iterative and traceable derivation of architectural decisions from NFRs.

3.1 Architectural Policies and Mechanisms

Borrowing a page from related disciplines [3], architects may reason about the overall solution properties using architectural policies, and later refine them (perhaps from existing policy catalogs) into more concrete artifacts and concepts, such as component models, detailed code design, standards, protocols, or even partial code itself. These serve as input to software designers and developers Thus, architects define policies for specific architectural concerns and identify alternative mechanisms to implement such policies.

This *reification* process encarnates architectural decisions and yields more concrete artifacts; successive reifications end with implementation (code or components) that satisfy the original NFRs.

As a brief example (expanded in Section 4), consider inter-communication among applications. Availability NFRs may relate to an availability concern, which may be addressed by fault-tolerance policies (e.g. master-slave replication or active replication) and a security concern may be addressed by access control policies (e.g. identification-, authorization- or authentication-based) [4]. Another architectural concern is the communication type might have the dimensions of sessions, topology, sender, integrity v/s timeliness [5], and synchrony. Then, the requirement *send a private report to subscribers by Internet* might be mapped in some project (in architectural terms) as requiring communication

"asynchronous, with sessions, with 1:M topology, with a push initiator mechanism, and priorizing integrity over timeliness". Based on these architectural requirements, an architect (or automated tool!) can search a catalog for any existing mechanisms or combination thereof that provides this specified policy; in our case, lacking additional restrictions, a good first fit is SMTP (the standard e-mail protocol), and this any available COTS that provides it.

3.2 Architectural Decisions Derivation

Using architectural policies and mechanisms, the architectural derivation process starts from *NFRs* specific with certain metrics for quality attributes (e.g. user-owned resources, availability = 99.9%). Quality attributes are associated to specific *architectural concerns* representing a technique subject that concern at software architect (e.g. access control, replication) which can be characterized with *dimensions* that define a discriminator factor among several policies that addresses to one (or more) NFRs (e.g. authentication kind in access control, persistent or transient state for replication; each dimension may be satisfied by some *architectural policies* that instantiates one or more concern dimensions to represent specific NFRs (e.g. something the user knows based authentication, replication with persistent state); each policy may be satisfied by several *architectural mechanisms* that addresses to satisfy policies (e.g. SMTP-AUTH for authentication based on something the user knows in Sending e-mails context, passive replication); and mechanisms may be provided by available *COTS* that package implementations of mechanisms (e.g. SendMail v8.1 and later for SMTP-AUTH, LifeKeeper for passive replication of SMTP servers on Linux). Figure 1 shows this derivation process with an example, and the base concepts that describe it.

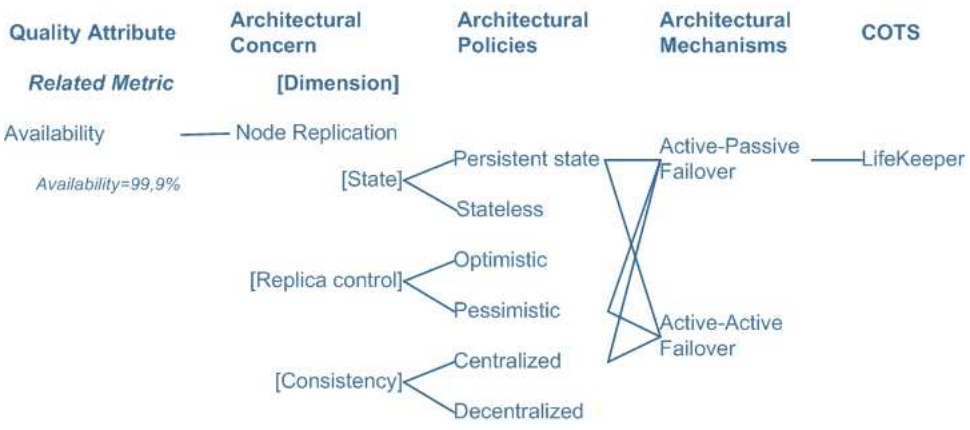

Fig. 1. Example of Architectural Decisions Derivation Process

The Azimut framework supports these concepts and the derivation process of architectural decisions maintaining traceability of such decisions from NFRs (metrics for quality properties) to COTS-based implementation.

3.3 Use case- and Scenario-based Approach

Using the concepts introduced in subsections 3.1. and 3.2., we employ an two-level specification approach, the first level describing NFRs in the use cases model, and the second level representing architectural policies in scenario specifications.

We use the Supakkul and Chung [7] notion of including new elements in use case models, describing NFR- or functionality-related architectural concerns. These models explicitly describe NFRs and architectural concerns at use cases level, allowing their manipulation in subsequent steps in the derivation process.

We then introduce *Architectural Policy Scenarios* to describe architectural policies at platform-independent level. A *concern* can be identified with a set of architectural policies, and each of these can be described using specific dimensions that specify with more details the NFRs in architectural terms in each use case. For this, we define a *Concern Scenario Template* for each concern that allows specifying a valid value for each of its dimensions; a concern scenario template describes one concern dimension for each of a series of steps.

Thus, the architect that needs to specify an NFR includes it in the use case model and its related architectural concerns, instantiates a concern scenario template related to the architectural concerns at hand, and specifies a valid value for each dimension described in each step. Valid values for concern dimensions must have been previously defined (e.g. "persistent-" or "transient state values" for the "state" dimension of the "replication" concern); the null values is a valid value too. If an architect doesn't need to specify a concern dimension in a given case, this step can be eliminated from the scenario instance.

Enabling the full power of use cases and scenarios as artifacts that drive software development process requires that they describe concern dimensions and requirements in a platform-independent manner, eschewing features related to specific technologies or platforms that implement mechanisms that satisfy them.

Current work in course includes the development of a *Concern Scenario Catalog*, as a set of concern use case templates to characterize open source COTS that are available to our own installation. This catalog will allow architects to reason about architecture through architectural policies, reuse concern specifications, and their resolutions. Concern Scenario Catalog is currently based on the middleware dimensions proposed by Britton [5], the metamodel of Fault Tolerance Group Properties of the UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms [11], the tactics to achieve quality attributes proposed by Bass et al. [12], and two domain-specific classifications for security requirements [4] and for security mechanisms [13].

4 Example

Let's explore an example with a requirement about extraction and propagation of information on stocks behavior. A requirement might be:

The system must obtain and synthesize information about a client's stocks, and propagate this summaries to the client. The system extracts this information from several sources according to the client's portfolio, summarizes it into a report, and sends the report to the client. The service must have availability = 99,9%, and must provide security through access control

This requirement can be decomposed into functional and non-functional requirements. The former can be *Extract information*, *Synthesize information* and *Send information*. The service *Send information* has the NFRs of *availability=99,9%* and *security by access control*.

From these requirements it is possible to identify *architectural concerns* as the `communication type` architectural concern relates to *the system must extract information from different sources* and *the system must send such reports to the client*; other architectural concerns are `access control related to security` and `concerns related to availability`. Figure 1 describes these architectural concerns and its related quality requirements (either NFRs or functionality) in the use case model. We use NFRs association points [7] to include new elements describing which NFRs are related to certain NFRs association point, but we also include in such elements the architectural concerns that refine the NFRs description. Also, it is possible that an architectural concern be derived from functional requirements, in which case we use functionality terms instead of the NFRs to maintain the traceability from requirements (quality attributes) to architectural concerns.

Now, we can use the Catalog Scenario Concern to select the Concern Scenario Templates of the architectural concerns identified in use case model. Examples of Concern Scenario Templates for access control, communication type and replication are shown below; each template describes the dimensions that allow discriminating among alternative mechanisms that address each given concern.

Access Control Concern Scenario Template
Actors The involved actors in this scenario
Quality Attributes The quality attributes related to Acces Control (eg. Security)
Dimensions for access control policyies
- **Authentication** The actor(s) can access the system through *something the users is* or *something the users knows* or *something the users have*
- **Authorization** and he/she(they) can have *individual-* or *group-* or *roles-*based authorization to access a certain system resources.

Communication Type Concern Scenario Template
Actors The involved actors in this scenario
Quality Attributes The quality attributes related to Communication Type
Dimensions for communication type policies

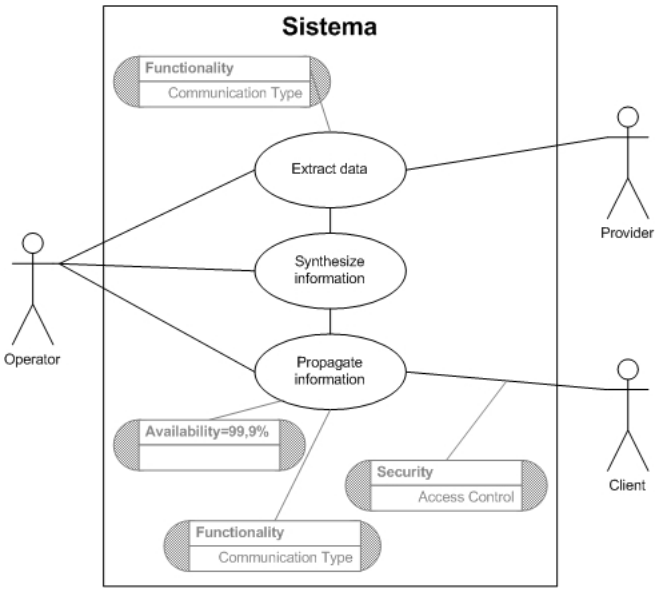

Fig. 2. Use Cases Model describing NFRs and Architectural Concerns

Initiator The actor(s) (or system) initiate(s) a *push* or *pull* or *RISPush* or *RISPull* communication type

Topology The *1* or *M* initiator(s) communicate(s) with *1* or *M* other(s) actor(s) (or systems)

Synchrony emphSynchronously or *Asynchoronously*,

Sessions *with sessions* or *without sessions*,

Integrity/Timeliness and privileging *integrity over timeliness* or *timeliness over integrity*

Replication Concern Scenario Template

Actors The involved actors in this scenario

Quality Attributes The quality attributes related to Replication (eg. Availability, Reliability, Performance)

Dimensions for replication policies

State The system (or service) must have replicas with a *persistent state* or a *transient state*.

Control of Replicas It must control such replicas in a *optimistic* or *pessimistic* manner

Consistency and the consistency style among replicas must be emphcentralized or *decentralized*

Without loss of generality, we will focus on the requirement *Send information* to show our scenario-based description second level and the subsequent derivation process for all identified architectural concerns kind in this example. *Extract*

information can be dealt with a similar process considering only `communication type` concern, and *Synthesize information* can be used to guide software development in traditional MDA manner.

Thus, we can specify each architectural concern specifying *Architectural Policies Scenarios* for `Communication Type`, `Access Control` and the architectural concerns related to availability requirement. In regard to availability we may relate some architectural concerns as replication, fault monitoring and recovery, but we will deal with `Replication` concern in this example due to lack of space.

Access Control Policies Scenario
Actors Clients
Quality Attributes Security
Dimensions for access control policies

- **Authentication** The clients can access to the system through *something the users knows*
- **Authorization** and he/she(they) can have *individual*-based authorization to access a certain system resources.

Communication Type Policies Scenario
Actors Operator, Clients
Quality Attributes Functionality
Dimensions for communication type policies

- **Initiator** The operator initiates a *push* or the client initiates a *RISPush* communication type
- **Topology** The operator communicates with M clients
- **Synchrony** *Asynchoronously*
- **Sessions** *with sessions*,
- **Integrity/Timeliness** and privileging *integrity over timeliness*

Replication Concern Policies Scenario
Actors
Quality Attributes Availability=99,9%
Dimensions for replication policy

- **State** The service must have replicas with a *persistent state*.
- **Control of Replicas** It must control such replicas in a *pessimistic* manner
- **Consistency** and the consistency style among replicas must be *decentralized*

Once requirements for architectural concerns are defined by specifying their dimensions, we need to reify these architectural policies to mechanisms. Table 1, 2 and 3 show several architectural mechanisms that satisfy some of the architectural policies for the `communication type`, `security` and `availability` concerns, respectively. Notice that in this example, architectural mechanisms are specifications of communication protocols, security mechanisms and tactics to meet availability goals, and therefore they are platform-independent just like architectural policies, although at a lower abstraction level. These mechanisms

Table 1. Partial content of the Architecture Reification Model (*ARM*) for Communication Type

Mechanism	Synchrony	Topology	Initiator	Integrity/Timeliness	Sessions
SMTP	Asynchronous	1:M	Push	Integrity	Yes
IM	Asynchronous	P2P	Push	Integrity	Yes
SOAP	Synchronous	M:1	Pull	Integrity	Yes
NNTP	Asynchronous	1:M	RISPush	Integrity	Yes
RSS	Asynchronous	M:1	RISPush	Integrity	Yes
SIP	Synchronous	P2P	RISPull	Timeliness	No
POP3	Synchronous	M:1	RISPull	Integrity	Yes
IMAP	Synchronous	M:1	RISPull	Integrity	Yes

are available as targets for the *ARM*-guided reification process that maps architectural policies for the each concern into specific mechanisms.

With the available *ARM* information (shown in Table 1, 2 and 3), the framework can recommend to the architect several possible mechanisms to satisfy the specified architectural policies. For example, the policies related to `Communication Type` for *Send information* can be reified to the protocols NNTP (used for client-initiated subscription-based articles reading) or SMTP (used to send e-mail); on the client side, IMAP (used for read news), POP3 and IMAP (both widely used for e-mail reading) or RSS (used for client-initiated subscription-based articles reading of RSS files).

Table 1 doesn't allow us to choose among these proposed protocols, but in practice the actual choice among alternative mechanisms is taken using information not available in the *ARM* (such as cost or simplicity). However, the framework allows recording this rationale and history of decisions to provide traceability and support the selection process.

Table 2. Partial content of the Architecture Reification Model (*ARM*) for Access Control

Mechanism	Authorization	Authentication
Personal Password	Individual	Something the user knows
ID Card	Individual	Something the user has
Fingerprint	Individual	Something the user is

On the other concerns, requirements for access control policies can be addressed with a password mechanisms, and `availability` requirements with passive replication of servers.

Once mechanisms are chosen, they are reified by choosing specific components that implement them. Figure 3 shows a (part of the) *MRM*'s catalog of Commercial Off-The-Shelf (COTS) components that describes available options to implement these particular communication mechanisms.

Table 3. Partial content of the Architecture Reification Model (ARM) for Node Replication

Mechanism	State	Replica Control	Consistency
Active Replication	Persistent State	Pessimistic	Decentralized
Passive Replication	Persistent State	Pessimistic	Centralized
Voting	Stateless	Pessimistic	
Spare	Persistent	Pessimistic	Centralized

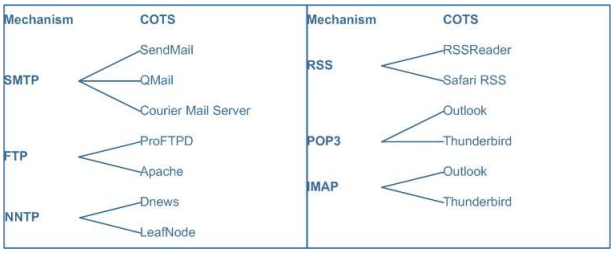

Fig. 3. Partial content of COTS Catalog in the Mechanism Reification Model (MRM)

If SMTP and IMAP or POP3 were chosen, the MRM-known available COTS alternatives are SendMail, QMail and Courier Mail Server (for SMTP) and Outlook and Thunderbird (for POP3 and IMAP).

Also, we need to select implementations for the chosen *access control* and *replication* mechanisms. For instance, SMTP-AUTH protocol can implement access control for SMTP, and therefore we need to identify COTS that implement SMTP-AUTH, such as SendMail (8.1 and later), Qmail (with qmail-smtpd-auth patch) and Courier Mail Server. Also, Outlook and Thunderbird implements POP-AUTH and IMAP-AUTH to support a access control mechanisms for sending mail. Regarding replication, there are several possibilities as well: realizing passive replication maintaining SMTP server replication and related policies with ad-hoc development; purchasing/adquiring COTS with this capabilities (e.g. LifeKeeper for Linux, SMTP.NET for Windows); or outsourcing this service to third parties defining SLA (availability=99,9%).

At this point, the architect makes the first decision about platform, in this case picking one on which both products run; the known choices are Windows (a gamut of choices itself) and Linux. We leave that last leg of the exercise to the reader.

5 Further and Related Work

We are employing the concept of *architectural policy use case catalog* in an Azimut framework (shown schematically in Figure 5). The Azimut framework aims to package, automate and reuse architectural decisions along the software development process, by means of some newly introduced MDA transformations.

Given the nature of the reification and description of architecture policies, the framework also provides traceability of architectural decisions.

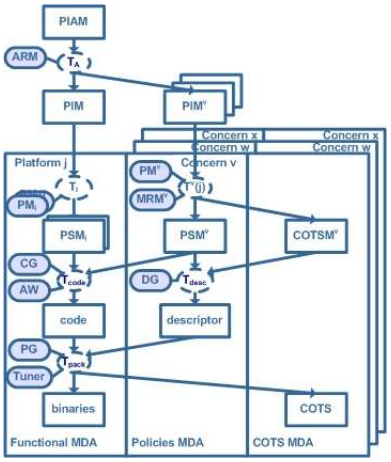

Fig. 4. Azimut Framework to NFR y COTS

This framework includes a Platform-Independent Architecture Model (PIAM) that combines description of domain components (from functional requirements) and architectural policies (from NFRs) at a platform-independent level. A *catalog* of architectural policy use cases is employed to characterize architectural policies and componentes, whose use was illustrated in the Example section. The *Architecture Reification Model* (ARM) holds the information to support derivation of componentes from architectural decisions, and the *Mechanism Reification Model* (MRM) guides transformations from architectural mechanisms into specific componentes and COTS. Framework details are available in [14].

Work in progress adds more concerns and dimensions to the Azimut framework supports multiple selection that satisfy different categories with a same solution (set of mechanisms), and resolve constraints of mechanism combinations.

6 Conclusions

We have presented a use case-based approach to describe NFRs. This approach is based on the concepts of *architectural policies* as means to describe concern-specific NFRs in architectural terms, and *architectural mechanisms* as constructs (more concrete) that are known to satisfy these architectural policies.

These concepts are notationally embedded in use case models describing NFRs and architectural concerns at platform-independent level, and in *concern scenario templates* that specify the valid values for each dimension of a concern.

An *concern scenario catalog* under development describes policies using concern use case templates.

References

1. *Object Management Group: UML 2.0 Superstructure Specification.* (Oct 2004). http://www.omg.org/cgi-bin/doc?ptc/2004-10-02
2. *Object Management Group: MDA Guide Version 1.0.1* (June 2003). http://www.omg.org/cgi-bin/doc?omg/03-06-01
3. Policy and Mechanism Definitions. http://wiki.cs.uiuc.edu/MFA/Policy+and+Mechanism
4. Firesmith, D.: *Specifying Reusable Security Requirements.* Journal of Object Technology, Vol. 3, N 1, Jan-Feb 2004 (2004), pp.61-75. http://www.jot.fm/issues/issue_2004_01/column6
5. Britton, C.: *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems.* Addison-Wesley Professional (Dec 2000).
6. Jacobson, I., Ng, P.: *Aspect-Oriented Software Development with Use Cases.* Addison Wesley Professional (Dec 2004).
7. Supakkul, S. and Chung, L.: *Integrating FRs and NFRs: A Use Case and Goal Driven Approach* Proc. SERA 2004, pp. 30-37.
8. Mylopoulos, J., Chung, L. and Nixon, B.: *Representing and Using Non-Functional Requirements: A process-Oriented Approach*, IEEE Transactions on Software Engineering, June 1992, pp. 483-497.
9. Bencomo, N., Matteo, A.: *Traceability Management through Use Cases when Developing Distributed Object Applications.* International Workshop Open Issues in Industrial Use Case Modeling 2004.
10. Amaya, P., Gonzalez, C., Murillo, J.: *MDA and Separation of Aspects: An approach based on multiple views and Subject Oriented Design.* AOM 2005.
11. OMG Specification : *UML^{TM} Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, (Jun 2004) http://www.omg.org/docs/ptc/04-06-01.pdf
12. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice, Second Edition* Addison-Wesley Professional (Apr 2003).
13. http://sarwiki.informatik.hu-berlin.de/Authentication_Mechanisms
14. López, C., Astudillo, H.: *Explicit Architectural Policies to Satisfy NFRs using COTS.* Technical Report DI-2005/09, Departamento de Informática, Universidad Técnica Federico Santa María, Valparaíso, Chile (2005).