Explicit Architectural Policies to Satisfy NFRs using COTS

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 hinge more on non-functional requirements (NFRs) than on functional ones, since the architecture stipulates *which* software to build. Model-Driven Architecture (MDA) aims to automate the derivation/generation of software from high level architectural specifications, but most current MDA implementations start from software design (i.e. *how* to build a software piece) rather than software architecture. This article presents an approach to extend MDA through the concepts of *architectural policies and mechanisms.* The key ideas are representation of NFRs through architectural concerns using architectural policies, systematic reification of policies into mechanisms, and multi-dimensional description of components as implementators of mechanisms. A detailed illustrative example is provided. Azimut framework realizes these ideas, supports larger-scale work through catalogs of policies and components, and allows traceability and reuse of architecture by enabling enabling these architecture-level descriptions and reasoning.

1 Introduction

Model-driven Software Development and MDA [1] arise from the possibility to build software systems through systematic transformations of high level models. Most proposed approaches emphasize modeling and transformations that address functional requirements, but in practice Non-Functional Requirements (NFRs) such as reliability, performance and stability are much harder to satisfy, and therefore the ones that require most attention from software architects.

Some proposals [2–9] extend MDA to deal explicitly with NFRs, especially from a perspective of components more than from detailed software design. Yet remains much work to be done: there is no traceability of decisions from NFRs to architecture to design and to final implementation, nor techniques to generate hybrid solutions that combine pre-existing components and ad-hoc development.

This article presents the key ideas of a Azimut framework that extends MDA with explicit modeling of architectural policies and their derivation into implementation. The key framework ideas are representation of NFRs with architectural policies, mapping and systematic refinement of architectural policies into mechanisms, multi-dimensional description of components (COTS) and specific

products insofar as they implement architectural mechanisms, and ongoing development of a multi-dimensional catalog of components.

The reminder of this paper is structured as follows: Section 2 provides a brief overview of related work on NFRs in MDA; Section 3 introduces the concepts of *Platform-Independent Architecture Model* (PIAM), *Architecture Reification Model* (ARM), and *Policies-Specific Platform-Independent Model for Concern* v (PIM^v); Section 4 illustrates the Azimut framework and its concepts with a detailed example; and Section 5 discusses conclusions and further work.

2 MDA, NFRs, and Component-based Development

Software architects focus more on NFRs than on functional requirements because the former are much harder to satisfy in large and distributed systems. NFRs cannot be satisfied with local design decisions, but require global solutions because they correspond to systemic properties; for example, security and stability usually cannot be added ex-post-facto to an already running system without large effort and risk. Software architecture focuses on reasoning about *which* software to build or to include, and not necessarily about *how* to build it. Therefore, system NFRs are key since they determine the nature of the solution.

However, despite having architecture in their name, most proposed MDA methods and tools (e.g. [10–13]) take as starting point a description in terms of functional components at *design* level, leaving the resolution of NFRs to prior steps.

Some recent projects [2–9] have addressed explicit mechanisms to satisfy NFRs through MDA transformations. From a component-based standpoint some projects [2–4] try to satisfy system NFRs through a model-driven process for selecting components to achieve systemic NFRs, and later configuring and deploying the selected components. Other projects from a more design-oriented approach [5–8] aim to generate implementations that satisfy multiples NFRs at once, starting out from design description at platform-independent (PIM) level; thus, NFRs must be modeled and solved at prior stages.

Other researches deal with NFRs on MDA, but they have focus on one NFR [14, 15, 9], on code generation to monitor NFRs [17] or to use MDA to analyze, at design time, NFRs [16]. We focus on approaches that implement solutions to multiple NFRs.

3 Architectural Policies to Implement NFRs through MDA and COTS

Our research goal is developing tools to describe, automate and keep traceability of architectural decisions from NFRs into implementations that satisfy them, using COTS whenever possible. The key conceptual feature is descriptions using *architectural policies and mechanisms.*

3.1 Architectural Policies and Mechanisms

Architects may reason about the overall solution properties using architectural policies, and later refine them (perhaps from existing policy catalogs) into artifacts and concepts that serve as inputs to software designers and developers, such as component models, detailed code design, standards, protocols, or even code itself. Thus, architects define policies for specific architectural concerns and identify alternative mechanisms to implement such policies. For example, an availability concern may be addressed by fault-tolerance policies (such as master-slave replication or active replication) and a security concern may be addressed by access control policies (such as identification-, authorization- or authentication-based) [19].

Each *reification* yields more concrete artifacts; thus, architectural decisions drive a process of successive reifications of NFRs that end with implementations of mechanisms that do satisfy these NFRs.

To characterize such reifications, we use a vocabulary taken from the distributed systems community [18], duly adapted to the software architecture context:

Architectural Policies: The first reification from NFRs to architectural concepts. Architectural policies can be characterized through specific concern dimensions that allow describing NFRs with more details.

Architectural Mechanisms: The constructs that satisfy architectural policies. Different mechanisms can satisfy the same architectural policy, and the differences between mechanisms is the way in which they provide certain dimensions.

As a brief (expanded in Section 4) example, consider inter-communication among applications. An architect's concern is the communication type, which might have the dimensions of sessions, topology, sender, and integrity v/s timeliness [20]; to this we add 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 Generation of Policies-Specific PIMs

Figure 1 gives an overview of the Azimut framework. We distinguish two PIM levels:

1. *Platform-Independent Architecture Model* ($PIAM$), independent from detailed design decisions, which relates NFRs to domain components using architectural policies.

2. Policies-specific *Platform-Independent Model for Concern v* (PIM^v), independents from any specific platform, which specify the mechanisms selected to satisfies the given architectural policies that describes an derived architectural concern from a NFR. Normal PIM and PIM^v are shown in different columns of Figure 1 as complementary views of the solution description.

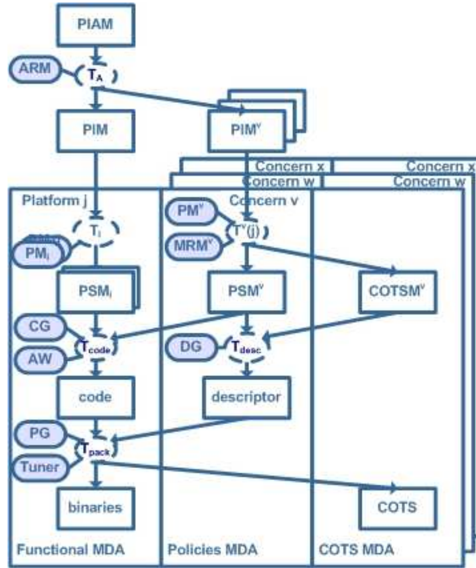

Fig. 1. Azimut Framework for NFRs and COTS

The $PIAM$'s elements are transformed to PIM's and PIM^v's using the *Architectural Reification Model* (ARM), which provides guidelines to go from architectural policies to architectural mechanisms. To support this process, the $PIAM$ characterizes platform-independent architectural policies and their dimensions, and the PIM^v characterize platform-independent mechanisms. The ARM indicates which combinations satisfy each policy, and may have rules about mechanism combinations (e.g. potential restrictions).

The transformation process determines feasible sets of mechanisms that provide specific architectural policies are proposed to the architect for validation or correction; approved mechanisms are organized according to the policies they satisfy (possibly several), and are grouped into PIM^v for each concern v, as shown in Figure 1. Thus, multiples views of a same mechanism allow maintaining the separation of concerns.

3.3 Treatment of Policies-Specific PIMs

The previous process generates platform-independent models (a PIM for functional domain components and a PIM^v for components supporting policies for each concern v) that must be implemented on specific platforms.

The PIM is transformed into a set of PSM_i for each technology i (e.g. data base engine); this is encoded by the platform mappings PM_i. The generated PSM_i can be transformed to code using conventional MDA approaches such as [10–13].

The platform-independent model PIM^v for each concern v may also be implemented with a MDA approach for NFRs [5–8], or may be mapped to selected COTS. In the former case, each PIM^v generates a PSM^v as encoded in the PM^v. Since each PSM^v gives place to code and deployment descriptors, they are weaved with each other and with the PIM-derived code using an aspect weaver (AW). Deployment descriptors are generated by a descriptor generator (DG) and guide the deployment process for the generated code.

Implementing a PIM^v with COTS requires an additional selection/transformation step. If one or more of the selected mechanisms are implemented by an available COTS (or set thereof), the process identifies them and the parameters they must take to implement the intended mechanism. This step is codified in a *Mechanism Reification Mappings* (MRM), which uses a COTS catalog that describes available components, the mechanism(s) that each implements, and their required parameters. In addition, the MRM contains rules about the possible combinations of COTS and the platforms where they can be implemented. The process uses several algorithms to determine the best combination of available components to implement the required mechanisms in PIM^v under the MRM constraints.

The $COTSM^v$ (COTS Model) describes the components to be assembled and deployed to satisfy policies for concern v, and is shown in Figure 1 by the third column. The $COTSM^v$ determine the generation of the deployment descriptors, similar to PSM^v. The deployment descriptors include information about the COTS parameters.

The last transformation includes a $COTStuner$ that allows to configure COTS, finally yielding deployable work sets consisting of generated code, deployment descriptors and configured components.

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 `security` and `availability`.

Without loss of generality, we will focus on the requirement *Send information* to show our derivation process for all identified architectural concerns 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.

Figure 2 shows the architectural concerns related to the requirements of this example, and the valid values for the dimensions of `communication type`, `security` [21, 22] and `availability` [21, 23] concern.

Thus, we can specify a requirement for `Communication Type` being *asynchronous*, with *1:M topology* and *Push* or *RISPush* initiator kind; also, the system must support *sessions* that privilege *integrity over timeliness*. `Security` requirements are focused on access control, and we assume that these requirements are *individual authorization* and *authentication based on something that user knows*[22], as usual. `Availability` may be reified to several architectural concerns, but we will focus on *replication* concern. To meet a high availability requirement, we specify that it needs replica with *persistent state*, *centralized consistency* and *pessimistic control of replicas*. Notice that an important aspect for replication is fault monitoring and recovery, but we don't deal with these aspects in this example lack lack of space.

Once requirements for architectural concerns are defined by specifying theirs 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 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).

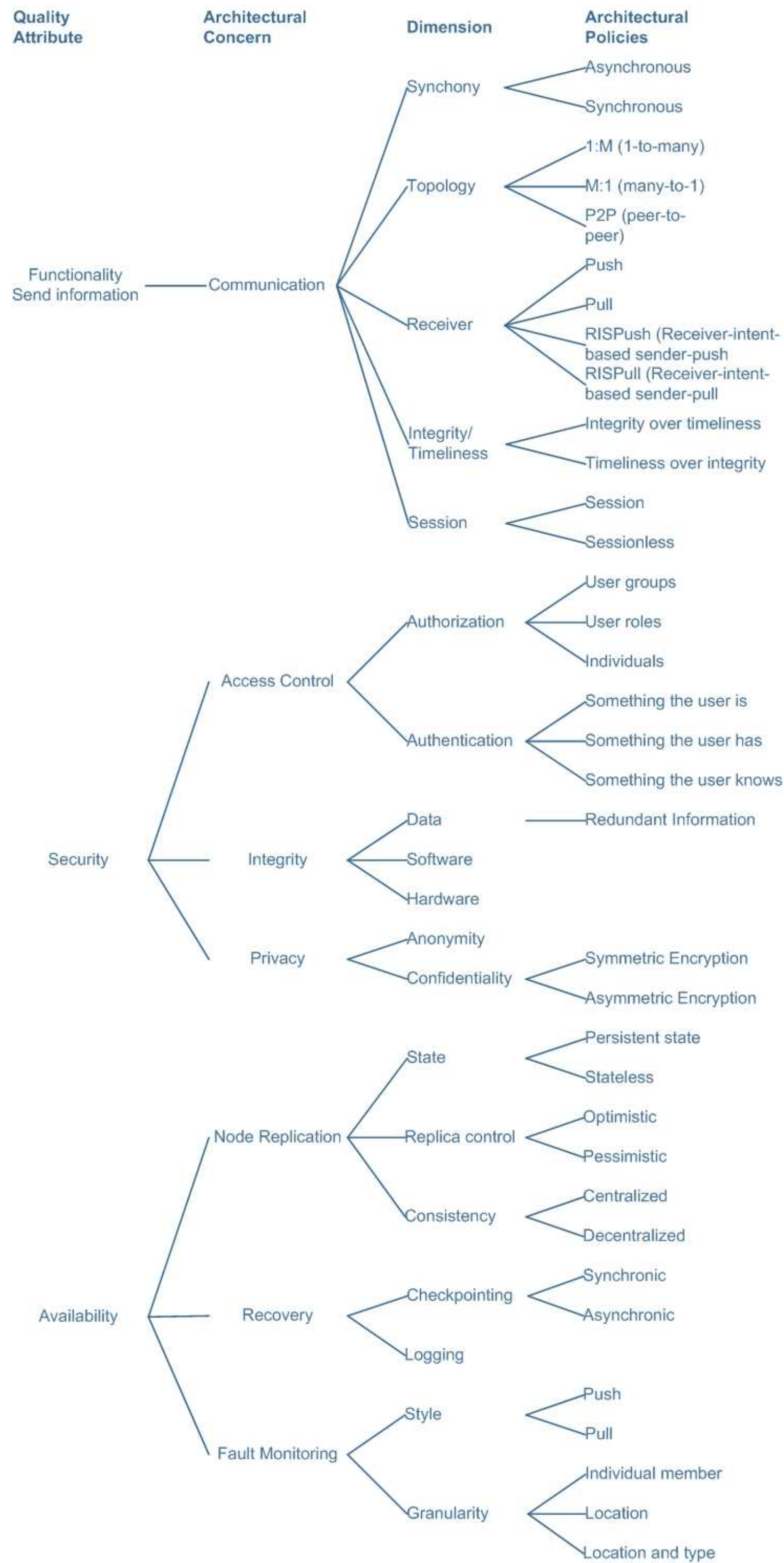

Fig. 2. Partial content of the Architecture Reification Mappings (ARM)

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

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 allow to record 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.

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

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.

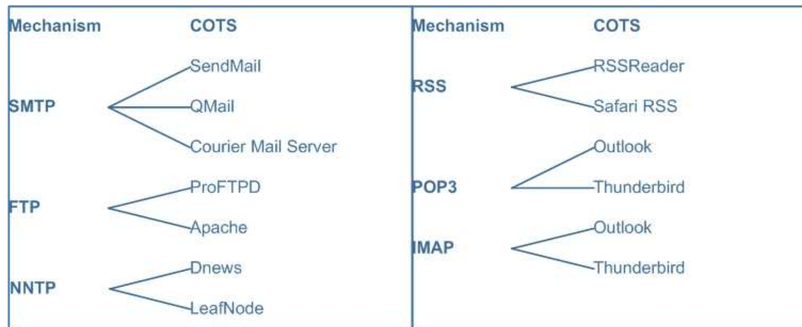

Fig. 3. Partial content of COTS Catalog in the Mechanism Reification Model (MRM)

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 select implementation for selected *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 Work

We are working to define model transformations from PIAM to COTSM in order to support an MDA process to describe architectural decisions in different phases of development and support COTS selection process, and to incarnate this process in a MDA tool.

Work in progress includes expanding the policies catalog by adding more concerns and their dimensions, extending the framework to allow generation and comparison de alternative combinations of mechanism to satisfy a given problem, and identifying constraints on mechanism combinations. A hard problem that is being jointly explored with artificial intelligence researchers is the generation and selection of mechanism combinations that are optimal according to some

non-technical criteria, such as purchase cost, deployment risk, and development complexity.

6 Conclusions

This article presents the conceptual model of the Azimut framework to extend MDA for reasoning about architectural policies related to NFRs, preserving traceability of architectural decisions, and generating hybrid solutions with COTS and ad-hoc development. The key framework ideas are representation of NFRs with architectural policies, multi-dimensional description of components as implementations of specific architectural mechanisms, systematic refinement and mapping from architectural policies into (component) mechanisms, and development of a component catalog on these lines.

The systematic use of architectural policies and mechanisms allows describing and reasoning architectural decisions at an architecture level, i.e. determining *which* software to build to provide certain required systemic properties (NFRs). The Azimut framework allows describing, encapsulating, automating and reusing architectural decisions from architecture to implementation, and maintains traceability of such decisions.

The Azimut framework extends MDA with several models: *Platform-Independent Architecture Models* ($PIAM$) to relate NFRs to domain components using architectural policies; *Architecture Reification Models* (ARM) to describe which combinations of architecture mechanisms satisfies each $PIAM$-described policy; and *Mechanism Reification Models* (MRM) to indicate which COTS (and with which parameters) implement each architecture mechanism.

References

1. *Object Management Group: MDA Guide Version 1.0.1* (June 2003). http://www.omg.org/cgi-bin/doc?omg/03-06-01
2. Gokhale, A., Balasubramanian, K., and Lu, T. *CoSMIC: Addressing Crosscutting Deployment and Configuration Concerns of Distributed Real-Time and Embedded Systems.* OOPSLA 2004, ACM Press, p. 218-219.
3. Solberg A., Huusa K. E., Aagedal J. ., Abrahamsen E: *QoS-aware MDA.* Workshop SIVEOS-MDA 2003, published in the ENTCS Journal.
4. Cao, F., Bryant, B., Raje, R., Auguston, M., Olson, A., Burt. C: *A Component Assembly Approach Based on Aspect-Oriented Generative Domain Modeling.* ENTCS 2005, pp.119-136.
5. Burt, C., Bryant, B., Raje, R., Olson, A., Auguston, M.: *Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models.* Proc. EDOC 2002, pp.212-223.
6. Silaghi, R., Fondement, F., Strohmeier, A.: *Towards an MDA-Oriented UML Profile for Distribution.* Proc. EDOC 2004, pp.227-239.
7. Simmonds, D., Solberg, A., Reddy, R., France, R., Ghosh, S.: *An Aspect Oriented Model Driven Framework.* Proc. EDOC 2005, to appear.
8. Weis, T., Ulbrich, A., Geihs, K., Becker, C.: *Quality of Service in Middleware and Applications: A Model-Driven Approach.* Proc. EDOC 2004, pp.160-171.

9. Almeida, J.P.A., van Sinderen, M.J., Ferreira Pires, L. and Wegdam, M.: *Handling QoS in MDA: a discussion on availability and dynamic Reconfiguration.* Workshop MDAFA 2003, TR-CTIT-03-27, pp. 91-96.
10. AndroMDA website. http://www.andromda.org/
11. OptimalJ website. http://www.compuware.com/products/optimalj/
12. ArcStyler website. http://www.interactive-objects.com/
13. SosyInc Modeler and Transformation Engine website. http://www.sosyinc.com/
14. Basin, D., Doser, J., and Lodderstedt, T.: *Model driven security for process-oriented systems.* Proc. SACMAT 2003, pp.100-109.
15. Lang, U., and Schreiner, R.: OpenPMF: *A Model-Driven Security Framework for Distributed Systems.* Presented at ISSE 2004 http://www.objectsecurity.com/ISSE04.pdf
16. Skene, J., and Emmerich, W.: *A Model Driven Architecture Approach to Analysis of Non-Functional Properties of Software Architectures.* Proc. ACE 2003, pp.236-239.
17. Pignaton, R., Villagra, V., Asensio, J., Berrocal, J.: *Developing QoS-aware Component-Based Applications Using MDA Principles.* Proc. EDOC 2004, pp.172-183.
18. Policy and Mechanism Definitions. http://wiki.cs.uiuc.edu/MFA/Policy+and+Mechanism
19. Firesmith, D.: *Specifying Reusable Security Requirements.* Journal of Object Technology, Vol. 3, N° 1, (Jan-Feb 2004), pp.61-75. http://www.jot.fm/issues/issue_2004_01/column6
20. Britton, C.: *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems.* Addison-Wesley Professional (Dec 2000).
21. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice, Second Edition* Addison-Wesley Professional (Apr 2003).
22. http://sarwiki.informatik.hu-berlin.de/Authentication_Mechanisms
23. 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