Thesis Proposal

SAD: Systematic Architecture Design

A semi-automatic method

Programa de Doctorat en Computació
Departament de Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya

Student: David Ameller
Supervisor: Xavier Franch
Table of contents

Table of Figures .................................................................................................................. 3
Table of Tables ..................................................................................................................... 3
Table of annexed papers ...................................................................................................... 3

1. Introduction ......................................................................................................................... 5
   1.1. Non-Functional Requirements .................................................................................. 5
   1.2. Software architecture ............................................................................................... 6
   1.3. Systematic software development .............................................................................. 6
   1.4. Putting all together .................................................................................................... 6
   1.5. Background .............................................................................................................. 7
   1.6. Motivation ................................................................................................................ 8

2. Thesis proposal ................................................................................................................... 9
   2.1. Goals ........................................................................................................................... 9
   2.2. Considerations .......................................................................................................... 10

3. State of the art .................................................................................................................. 11
   3.1. Model-Driven Development ..................................................................................... 11
   3.2. Non-Functional Requirements ............................................................................... 12
      3.2.1. Modeling NFRs .................................................................................................. 12
      3.2.2. MDD approaches that consider NFRs ............................................................... 14
      3.2.3. SADMs that consider NFRs .............................................................................. 15
   3.3. Architectural Knowledge ......................................................................................... 18
      3.3.1. Ontologies for AK ............................................................................................. 18
      3.3.2. AK tools ............................................................................................................. 19
      3.3.3. MDD approaches that integrate AK .................................................................. 20
      3.3.4. Architectural Description Languages .............................................................. 21
   3.4. Approaches dealing with MDD, NFRs and AK ......................................................... 23

4. Working plan ..................................................................................................................... 25
   4.1. Calendar ..................................................................................................................... 25
   4.2. Methodology ............................................................................................................. 26

References ............................................................................................................................ 27

Table of acronyms ................................................................................................................. 33
Table of Figures
Figure 1: Relationships between NFRs, AK and MDD ...................................................... 7
Figure 2: Considerations ................................................................................................ 10
Figure 3: Research topics in MDD ................................................................................ 11
Figure 4: Contributions of the proposed thesis .............................................................. 24
Figure 5: Timeline ........................................................................................................... 25
Figure 6: Dedication and iterations ................................................................................ 26

Table of Tables
Table 1: Identified goals .................................................................................................... 9
Table 2: Comparison of approaches for modeling NFRs .................................................. 13
Table 3: Comparison of approaches that consider NFRs in MDD .................................. 15
Table 4: SADMs comparison ......................................................................................... 17
Table 5: Comparison of AK ontologies .......................................................................... 19
Table 6: Comparison of AK tools .................................................................................. 20
Table 7: Comparison of MDD approaches that consider architectural aspects .......... 20
Table 8: ADLs comparison ............................................................................................. 23
Table 9: Task list .............................................................................................................. 25

Table of annexed papers
David Ameller and Xavier Franch, “Assigning treatments to responsibilities in software architectures,” in EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), 2007 ................................................................. 37

David Ameller and Xavier Franch, “Asignación de tratamientos a responsabilidades en el contexto del diseño arquitectónico dirigido por modelos," in Workshop on Desarrollo de Software Dirigido por Modelos (DSDM), Jornadas de Ingeniería del Software y Bases de Datos (JISBD), 2007 ................................................................. 39

David Ameller and Xavier Franch, “Combining model-driven development and architectural design in the AR3L framework,” in a technical report of Software Department (LSI) of the Universitat Politècnica de Catalunya (UPC), 2008 ................................................................. 49

David Ameller and Xavier Franch, “Definición de una ontología para el proceso de DSDM considerando requisitos no-funcionales,” in Workshop on Desarrollo de Software Dirigido por Modelos (DSDM), Jornadas de Ingeniería del Software y Bases de Datos (JISBD), 2009 ................................................................. 65

David Ameller and Xavier Franch, “Usage of architectural styles and technologies in IT companies and organizations," in Workshop on Empirical Assessment in Software Architecture (EASA), European Conference on Software Architecture (ECSA), 2009 ................................................................. 77


David Ameller, Xavier Franch and Jordi Cabot, “Dealing with Non-Functional Requirements in Model-Driven Development,” in 18th IEEE International Requirements Engineering Conference (RE), 2010 ................................................................. 83
1. Introduction
The proposed thesis grows around three research areas: Non-Functional Requirements, Software architecture, and systematic software development. These three areas are introduced in this section. Refer to section 3 (State of the art) for the last advances in each area.

1.1. Non-Functional Requirements
Many prominent researchers have stated that it is not feasible to produce a software product that satisfies the necessities of stakeholders without taking into account Non-Functional Requirements (NFRs). To understand what a NFR is, here is one of its definitions (for other definitions see [1, 2]):

"[NFR is...] a requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. [NFR is...] a requirement that specifies physical constraints on a functional requirement", I. Jacobson et al., “Unified Software Development Process”, 1999 [3].

L. Chung et al. (2009, 2000) [1, 4] argued that the lack of integration of NFRs with functional requirements can result in long time-to-market and more expensive projects. This fact has been recurrently mentioned in previous publications such as the book “Software Requirements: Objects, functions and states”, A. M. Davis (1993) [5] and in the famous paper “No Silver Bullet”, F. P. Brooks (1987) [6]. This is the first paragraph of L. Chung et al. (2009).

"Essentially a system’s utility is determined by both its functionality and its nonfunctional characteristics, such as usability, flexibility, performance, interoperability and security. Nonetheless, there has been a lop-sided emphasis in the functionality of the system, even though the functionality is not useful or usable without the necessary non-functional characteristics", L. Chung et al., “On Non-Functional Requirements in Software Engineering”, 2009 [1].

According to M. Glinz (2007) [2], functional and non-functional requirements set the boundary of an important dimension of the software, its quality. For example, in the ISO/IEC 9126 quality standard [7] we can find quality characteristics like: functionality, reliability, usability, efficiency, maintainability, and portability (ISO/IEC 9126 was superseded since 2005 by ISO/IEC 25000, a.k.a. SQuaRE [8]). Notice that these quality characteristics are the same that were mentioned in the previous definition of the term NFR. In other words, NFRs and software quality are highly related.

"[Software quality is...] the capability of software product to satisfy stated and implied needs when used under specified conditions", ISO/IEC 25000, 2005 [8].

The “needs” mentioned in the previous definition could be any kind of software requirement: functional, or non-functional. Is important to notice that using the term quality as defined in the ISO/IEC standard, we are referring to the grade of satisfaction of both types of requirements: functional and non-functional.
1.2. Software architecture

Software architecture is the result of the architectural decisions and, by extension, technological decisions made during the software development process. These decisions are the bases of Architectural Knowledge (AK). Many definitions of AK have been published, and sometimes this concept becomes a bit fuzzy. In my opinion, the clearest definition of AK is in the next citation (for more information look at this systematic literature review on AK [9]).

Architectural Knowledge = Design Decisions + Design, “Architectural knowledge consists of architecture design as well as the design decisions, assumptions, context, and other factors that together determine why a particular solution is the way it is.”, P. Kruchten et al., “Building Up and Reasoning About Architectural Knowledge”, 2006 [10].

This knowledge is not easily accessible, normally it resides in the architect’s mind, and in few cases it is disseminated in the project documentation, but even in this case it lacks of explicit reasoning about the alternatives considered previous to the decision.

“Except for the architecture design part, most of the architectural knowledge usually remains hidden, tacit in the heads of the architects”, P. Kruchten et al., “Building Up and Reasoning About Architectural Knowledge”, 2006 [10].

1.3. Systematic software development

Many times, software engineering researchers have tried to find ways to systematize the software development processes. The last and more promising initiative in this direction is Model-Driven Development (MDD), an emergent development method that uses models as the principal artifact. This method is based on the separation of the essential specification of the system and its implementation using a specific platform. The benefits of using MDD are higher abstraction level and improved platform independence. A clear example of these benefits could be the adaptation to new technologies; this problem can be alleviated by using technology-independent models that can be transformed semi-automatically into technology-specific models to fulfill the trendy technological needs.

“Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing”, S. Mellor et al., “Model-Driven Development”, 2003 [11].

1.4. Putting all together

As we could see in the next definition, considering NFRs in a software development process (e.g. MDD) have an immediate impact on the architecture of the software system.


Common MDD processes are focused on the domain models, the implementing technologies (that are decided by the developer), and generation of code. This implies that two steps should be added to the common MDD processes, first consider NFRs from the start of the process and second design software architecture as part of the process.
Taking into account NFRs in MDD will bring the possibility to reason about them during the development processes in a semi-automatic way. Concretely, NFRs play a crucial role to make informed decisions over architectural aspects and selection of implementing technologies (see citation).

"[NFRs…] play a critical role during system development, serving as selection criteria for choosing among myriads of alternative designs and ultimate implementations", L. Chung et al., “Non-Functional Requirements in Software Engineering”, 2000 [4].

The proposed thesis will define a methodology to design software architectures. The method will use AK and will consider NFRs to make informed decisions. The method will be implemented inside a framework/tool that will use MDD as a way to semi-automate this process.

![Figure 1: Relationships between NFRs, AK and MDD](image)

1.5. Background

Previously I have done some works in this direction: the BSc final project, the MSc on computing thesis, and papers in conferences and workshops. The published papers are available at the end of this document in chronological order. The BSc project and the MSc thesis have public electronic editions:

- BSc final project: [http://upcommons.upc.edu/pfc/handle/2099.1/5302](http://upcommons.upc.edu/pfc/handle/2099.1/5302)
- MSc on computing thesis: [http://upcommons.upc.edu/pfc/handle/2099.1/7192](http://upcommons.upc.edu/pfc/handle/2099.1/7192)

The BSc final project was named Assignment of Responsibilities in 3-Layered architectures (AR3L, 2006-2007) [13, 14], this project consisted in the elicitation of several types of responsibilities from a UML class diagram that then were associated with treatments that are habitually present in 3-Layered architectures. These associations were restricted by a set of non-functional properties. As part of the project I developed a tool with the same name, AR3L [15].

AR3L evolved into a more organized and generalized framework named Responsibility Detection and Transformation (RDT, 2007-2009) [16-18]. There were two major differences, the first difference was that in RDT the responsibilities could be obtained from several diagrams that then were mixed together and the second difference was that the treatments were applied in three subsequent steps: architectural treatments, design treatments and technologic treatments. The first type of treatments allows the selection of an architectural style, so this
framework was not limited to a 3-layered architectural style. Unfortunately, this framework was too complex to be implemented due to the inherent complexity that emerges from the use of responsibilities. The concept of responsibility is too atomic.

My master thesis (2009) [18] was focused in a systematic literature review about NFR in Model-Driven Engineering (MDE). Most of the results of the master thesis are also applicable to MDD, because MDD can be understood as a subset of MDE. Apart from the knowledge acquired from the papers read, the most important conclusion of the master thesis was that the importance of NFRs is not reflected in the current MDD research. Another part of the master thesis was a survey to know the most habitual architectural styles and technologies used in industry practice. The results were presented in a workshop about empirical studies (2009) [19], and later, with deeper detail, as a poster (2010) [20]. The main motivation to drive the survey was to obtain an up-to-date perspective of the industrial practice to reorient my research. Concretely, from this survey it was determined that Service Oriented Architecture (SOA) and Java technologies are the best candidates to focus my efforts.

At the same time Xavier Franch and I were also determining accurately the role that NFRs play in MDD. We designed an ontology (2009) [21] that identifies, defines and relates the concepts that take part in MDD and NFRs. In this work architectural concepts become the most important elements in relation with NFRs. This is the reason why, at this point my research change the direction to focus on architectural aspects and NFRs. In the beginning, MDD was the principal area of research, now its role is to provide tools and mechanism to design software architectures in a semi-automatic way.

Xavier Franch, Jordi Cabot and I made a study about MDD approaches dealing with NFRs (2010) [22]. In this study we compared the existing MDD approaches and how developers deal with NFRs in practice. We also proposed two approaches to deal with NFRs from a theoretical perspective. The two proposed approaches were centered on the human interaction, one without interaction and one with full interaction. We explain in the paper that these two extreme approaches can be combined to find the most adequate development environment. The results of this work will be published in the proceedings of the 18th IEEE International Requirements Engineering Conference (RE). The annexed copy is a technical report with the same contents.

1.6. Motivation

Now that the role of NFRs in software development processes is clearer to me, I can start thinking about the ways to semi-automate the software architecture design.

There are many works that relate NFRs with architecture, for instance many software development methodologies (e.g. RUP) take into account this relationship. There are some works that deal with NFRs in MDD, but many of these works only deals with one type of NFR and in a concrete domain. There is very little research done when we take into account the three concepts altogether: MDD, NFRs, and AK (evidence about these statements can be found in the state of the art, section 3). To fill this gap I would like to continue the research line I started in my BSc project in 2006.
2. Thesis proposal

The thesis is planned as a theoretical/methodological thesis, complemented with tool support, and empirical analysis and validation. The method that will be proposed in the thesis will take into account the necessary considerations to make it at least semi-automatic method. This systematization will be provided by a MDD process that will produce the adequate software architecture for a specified set of NFRs. In order to make this thesis reusable, the theoretical part of the method will be independent of the current MDD limitations. My intention is to generalize the method and make it useful for any systematic software development process.

In short, the method will begin with some kind of business model, a model that defines the necessities and behavior of a system (e.g. BPEL, BPMN, UML, etc.), and the specification of NFRs. Then, the architect will be suggested to apply several architectural decisions (from high level decisions to technological decisions), the suggestions will have alternatives and the justification of each of them will be shown. Once the suggestions are selected the method will have enough information to produce the architectural views (representations of the architecture). As part of the thesis I will do some exploratory research to know which are the most used architectural views in industrial practice, but at least two views will be needed: one for a high level architectural representation (architectural aspects), and one for the concrete representation (technological aspects). These two views will be associated with existent architectural views (e.g. 4+1 architectural views proposed by Kruchten (1995) [23]).

2.1. Goals

The research done in this thesis will answer to this question:

*Is it possible to design the architecture of a software system in a systematic way considering a set of non-functional requirements?*

To find the answer to this question I will try to create the method that will proof its viability. The following are the principal goals I identified. They must be accomplished for the design software architectures considering NFRs.

<table>
<thead>
<tr>
<th>Table 1: Identified goals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>G1:</strong> Select the most adequate type of NFR and architectural views for an architectural style.</td>
</tr>
<tr>
<td><strong>G2:</strong> Find (or create) a good way for representing NFRs (using models)</td>
</tr>
<tr>
<td><strong>G3:</strong> Define a methodology that integrates MDD, NFR, and AK to produce software architectures.</td>
</tr>
<tr>
<td><strong>G4:</strong> Design and produce a detailed specification of the framework that will support the methodology.</td>
</tr>
<tr>
<td><strong>G5:</strong> Refine with adaptations and improvements our ontology [21] for architectural and technological aspects.</td>
</tr>
<tr>
<td><strong>G6:</strong> Build a knowledge base for architectural and technological aspects. The knowledge base will use the defined ontology.</td>
</tr>
</tbody>
</table>
2.2. Considerations

The following considerations will be used to focus the thesis investigation and eliminate possible deviations.

- Focus on the decisional aspects that are related with non-functional requirements. Functional requirements will be considered as input but will not influence the decisional part of the method.
- Focus on the architectural design. The generation of an implementation of the system-to-be is not part of the proposed thesis, but existing code generation approaches could use the work done for this thesis to improve its results.
- Focus on one architectural style, Service-Oriented Architecture (SOA). SOA is a recent architectural style that is at its height. One of the reasons to use SOA is that it has similarities to the usual business organization. This reason makes it a good candidate for the thesis because the method that will be proposed will begin from a business model.
- Focus on one kind of input model. I will use a business model as input model. It will be selected considering the most adequate business models for SOA systems and the current practice in industry.
- Focus only on few types of NFRs. I will select one or two types of NFRs, we will consider types of NFRs that have more influence in SOA systems, this information will be obtained from the experiences taken from interviews to architects.
- Focus on few architectural views, at least two (to be selected as result of G1). The selection will be based on the current practices in the industry.
- Focus on one family of technologies. There are many technological families, but we have found that Java technologies are the most used in software companies that produce SOA systems [19].

In the Figure 2 we can differentiate the parts that will be done (darken) from the parts that will not be done in the proposed thesis (lighten).
3. State of the art
The first part of the state of the art summarizes, with updated information, the work done in my MSc thesis: “Systematic Review: Non-Functional Requirements in Model-Driven Engineering” (sections 3.1 and 3.2). In this thesis proposal I added the state of the art on the research done for AK (3.3). Finally, several aspects of the relationship between MDD, NFRs and AK (3.4) are explained.

3.1. Model-Driven Development
Research done in MDD is divided into two main streamlines, on the one hand there is a research guided by industry needs, that is focused on frameworks (e.g. Eclipse), and standardization effort (e.g. UML, QVT, and other OMG standards). On the other hand, the academic research is focused on new and experimental development approaches that try to solve some problem.

Research topics in MDD are represented in Figure 3:

- **Frameworks and CASE tools** are projects developed by the industry or in cooperation with the academia. The reason is that this kind of project requires much dedication time to implement the ideas and because the project normally covers a lot of topics (see Figure 3). The most used framework in MDD research is Eclipse [24], and customized versions of it (e.g. MOSKitt [25] in the Spanish community).

- **Modeling languages** have been dominated by UML for long time, but now it seems that Domain Specific Languages (DSL) are gaining the attention of many communities because they are easy to use, and because of the normalization of metamodeling languages. This normalization allows the transformation between different modeling languages. The transformations are defined using transformation languages, and all that can be done in MDD is limited to the expressiveness of this transformation languages.

- The most well-known method in industry is MDA [26], but new methods of software development are habitual in MDD frameworks, normally to solve or alleviate a concrete problem. The development methodologies are composed by one or more theories or hypothesis, and a process definition of the development stages. The theories that support a software development methodology tend to be proven empirically (e.g. case studies).
It is not necessary to explain in depth these topics because they do not affect directly to the thesis. To obtain more detail in each research topic, please refer to the MSc thesis [18]. Electronic edition is available at: http://upcommons.upc.edu/pfc/handle/2099.1/7192

3.2. Non-Functional Requirements

As stated in the introduction of this proposal, NFRs play an important role in the design of the software architecture. This section is divided in three parts: modeling NFRs, NFRs in MDD, and Software Architecture Design Methods (SADMs) that consider NFRs (this part was not included in the MSc thesis).

3.2.1. Modeling NFRs

For the notation of NFRs in models there are two perspectives. One is to extend an existing modeling language to support NFRs representation, and the second is the use of well known modeling languages specially designed to represent NFRs.

Modeling languages extensions to represent NFRs

In the first perspective, the extended modeling language is normally UML because, nowadays, UML is the most used language for software specification and design. Many research efforts are centered on extending this language with UML profiles. Similarly, DSL aim at using a small, well-focused language that provides appropriate abstractions and notations for a particular domain. In the last years it seems to be a tendency to use DSLs instead of UML profiles, but it is not clear which modeling technique is better. Microsoft defends DSLs while OMG has done many standardization efforts for UML profiles.

UML profiles allow the customization and extension of the UML syntax and semantics to define specialized UML for a particular domain. The following are the standardized NFRs aware UML Profiles:

- Modeling and Analysis of Real-time and Embedded systems (MARTE, 2008) [27]. This profile focus on measurement sources, precision, and time expressions for NFRs (that is just a part of the profile).
- UML profile for Quality of Service and Fault Tolerance (QoS-FT, 2008) [28]. This profile uses two types of annotations: instantiation of template classes from the QoS Catalogs and annotations in UML models with QoS Constraints and QoS Values.

Other interesting profiles that take into account some form of non-functional property are:

- S. Bernardi et al. (2008) [29] present an extension to MARTE to give support for dependability analysis. Another profile that explores dependability is DMP (2003) [30].
- H. Wada et al. (2008) [31] uses a UML profile, named UP-SNFR, to support the specification of NFRs inside UML models for SOA systems. It is based on the idea of features.
- L. Zhu and I. Gorton (2007) [12] used UML profiles to specify architectural decision and NFRs, the good thing is that they can be used together. L. Zhu and Y. Lin (2009) [32] continued this work about modeling non-functional aspects and their impact on design decisions.
Another way, similar to UML profiles, is to model NFRs using a personalized metamodel that includes NFRs as a primary concept (these works can be seen also as DSLs). This practice can be found in the following works:

- L. Gönczy et al. (2009) [34] presented an approach for performance analysis in the context of SOA systems. The approach uses its own metamodel to represent NFRs.
- S. Kugele el al. (2008) [35] uses its own metamodel in the context of safety-critical real-time systems for embedded systems. In this case the treated NFR is resource usage (e.g. memory, processing time, etc.).

Modeling languages specific for NFRs

Goal-Oriented Requirements Engineering (GORE) is an approach advocating for the identification and analysis of goals as a prerequisite for writing complete and consistent requirements documents. Goal models are commonly used in GORE. These are some works that use GORE in the context of MDD:

- J. Mazón et al. (2007) [37] use goal models, but without specific mention of NFRs.
- S. Konrad et al. (2007) [38] use goal models to contemplate the existence of NFRs, but it is not clear how NFRs are used inside the proposed methodology.
- F. Alencar et al. (2009) [40] integrates GORE and MDD to fill the gap between requirements specification and the implementation of the software product. The approach is focused on functional requirements.

E. Yu published in 1997 the most used goal modeling language, $i^*$ [41]. This modeling language has many similarities with NFRs as explained in [42]. J. Cabot and E. Yu (2008) [43] have exposed their thoughts about the directions to take, and the open problems to support NFRs with goal modeling in MDD. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos have defined the NFR Framework (2000) [4]. This framework describes a notation to specify NFRs based on goals and soft-goals, and it has a wide acceptance in the requirements engineering community. The NFR framework has many common elements with $i^*$ language. L. M. Cysneiros (2001, 2004) [44, 45] has published several works to use $i^*$ notation in conceptual models.

Analysis

The following table summarizes the works about notation of NFRs in models.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Types of NFRs</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>[27]</td>
<td>Measurable constraints</td>
<td>Real-time systems</td>
</tr>
<tr>
<td>[28]</td>
<td>Quality of Service</td>
<td>Any</td>
</tr>
<tr>
<td>[29]</td>
<td>Dependability</td>
<td>Real-time systems</td>
</tr>
<tr>
<td>[31]</td>
<td>Security, Fault Tolerance</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>[12, 32]</td>
<td>Any</td>
<td>Any</td>
</tr>
<tr>
<td>[33]</td>
<td>Security</td>
<td>Any</td>
</tr>
</tbody>
</table>
We can see that UML profiles and DSL are designed for specific purposes while GORE approaches try to embrace all possible NFRs in a generic way. The work I pretend to do in this thesis is in similar direction to the works of L. Zhu and Y. Lin [32], were NFRs are related to design decisions.

3.2.2. MDD approaches that consider NFRs

NFRs may play two different non-exclusive roles inside MDD. The first is to use NFRs for driving the transformations and the second is to use NFRs to validate the results.

When I say that NFRs play the driver role in transformations, I am saying that the selection or the behavior of the transformations will depend on the specified NFRs. I found only two works that are advocating for this idea:

- A. Solberg et al. (2004) [46] presented an approach that will use the most adequate transformations depending on the QoS requirements. The authors had their own UML profile to specify QoS. The approach presented in the papers is based in gradually resolve QoS requirements when a model transformation is performed. These transformations use patterns that improve some aspect of QoS. They also considered a platform to resolve QoS requirements at run-time.

- A. Sterritt and V. Cahill (2008) [47] presented an approach to make model to model (M2M) transformations that take user priorities as NFRs, and select between trade-offs in the transformation. The target model in this case is an architectural model.

Concretely, in this paper they treat with distribution issues of architectural styles.

Similar works are presented as quality-driven transformations. At the beginning of this Thesis Proposal it was said that software quality is composed by functional and non-functional factors. This is especially true when we look at quality-driven transformations:

- J. I. Panach et al. (2008) [48] presented an approach to tackle with usability, where only the functional part of the usability is used to drive the transformations.

- E. Insfran et al. (2008) [49] presented an approach to tackle with understandability aspects. It is not mentioned in the paper if they take into account the non-functional factors.

The approaches that validate the conformance to specified NFRs normally use MDD techniques to generate a model that has some specific formalism where a concrete type of NFR can be analyzed. These are the most representative works in this direction:


- G. Rodrigues et al. (2005) [51] proposed a method to validate the reliability by using prediction models.
V. Cortellessa (2007) [52] presented a framework that uses NFRs to validate the resulting models. This approach proposes a separation of NFRs in different levels of abstraction to do the validation in each stage of the development process.

A. Fernandez et al. (2009) [53] proposed an approach to include a usability model as part of the MDD process for Web development. This model is evaluated using some metrics to produce system usability reports. A similar work has been done by F. Molina and A. Toval [36].


S. Gallotti et al. (2008) [56] proposed an approach that checks the QoS in SOA context.

### Analysis

The following table summarizes the MDD approaches that consider NFRs.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Types of NFRs</th>
<th>Domain</th>
<th>Instrument</th>
</tr>
</thead>
<tbody>
<tr>
<td>[46]</td>
<td>Quality of Service</td>
<td>Any</td>
<td>Patterns</td>
</tr>
<tr>
<td>[47]</td>
<td>Any</td>
<td>Any</td>
<td>Patterns</td>
</tr>
<tr>
<td>[48]</td>
<td>Usability</td>
<td>Any</td>
<td>OO-Method</td>
</tr>
<tr>
<td>[49]</td>
<td>Understandability</td>
<td>Any</td>
<td>Empirical work</td>
</tr>
<tr>
<td>[50]</td>
<td>Quality of Service</td>
<td>Any</td>
<td>Measurable models</td>
</tr>
<tr>
<td>[51]</td>
<td>Reliability</td>
<td>Any</td>
<td>LTSA*</td>
</tr>
<tr>
<td>[52]</td>
<td>Any</td>
<td>Any</td>
<td>Independent</td>
</tr>
<tr>
<td>[55]</td>
<td>Performance, reliability</td>
<td>Any</td>
<td>PRIMA** &amp; COBRA*** methods</td>
</tr>
<tr>
<td>[36, 53]</td>
<td>Usability</td>
<td>Web IS</td>
<td>Quality metrics</td>
</tr>
<tr>
<td>[54]</td>
<td>Performance, reliability</td>
<td>Any</td>
<td>Markov chains models</td>
</tr>
<tr>
<td>[56]</td>
<td>Performance, reliability</td>
<td>SOA</td>
<td>Probabilistic models</td>
</tr>
</tbody>
</table>

* Labeled Transition Systems Analyzer  ** PeRformance Inc reMental vAlidation  *** COmponent-Based Reliability Assessment

First, NFR-aware transformations group are in the same direction as the thesis I’m proposing. In particular, I want to mention the approach proposed by A. Sterritt and V. Cahill [47] because they consider architectural aspects in the transformations. Next, quality-driven works found are more oriented to fulfill functional aspects of the quality than the non-functional. Finally, the group of analysis and validation is where I found more works. These works are complementary to what I’m proposing for my thesis. My opinion is that integrating both points of view (NFR-aware transformations, and analysis and validation) could lead to better results.

### 3.2.3. SADMs that consider NFRs

We have seen above that most approaches only deal with one type of NFRs and/or one kind of architectural style or domain. A possible cause for this situation would be that each type of NFR deserves specific analysis, validation, and reasoning techniques. For example, availability is a type of NFR that cannot be validated until the system is implemented or simulated, while usability or accessibility can be validated before the whole system is implemented. In this section, it does not matter where the NFRs are validated or analyzed, because in any case NFRs major impact will be on the architecture of the system, and more concretely during the architectural decision making. Following the previous example, availability problems
will be detected on the testing phase but solutions (probably) would change the deployment of the architecture.

The Software Engineering Institute (SEI) has created many SADMs: SAAM [57], ATAM [58], CBAM, QAWs, QUASAR, ADD [59], ARID. Documentation for all of them can be found in SEI website [60]. I will comment only the two most influential for the proposed thesis.

- **Architecture Tradeoff Analysis Method (ATAM, R. Kazman et al., 1998) [58]** is a methodology that evolved from Software Architecture Analysis Method (SAAM, 1996). It is a method to understand the tradeoffs of the architectures of software-intensive systems. This method analyzes the architecture for several quality attributes (e.g. security, performance, etc.). The method guides the design decisions that have an impact on quality attributes. It is a spiral method, consisting in four phases: requirements elicitation including constraints, architectural views, modeling and analysis, and identification of tradeoffs.

- **Attribute-Driven Design Method (ADD, F. Bachmann and L. Bass, 2001) [59]**. A second version of this method was published in 2007 (it is only accessible in the SEI website). ADD is a method to design the software architecture of a system based on quality goals for the system. The method is extensible for any quality attributes but has been particularly elaborated for the attributes of performance, modifiability, security, reliability, availability and usability. The method considers three architectural views: module view, component and connector view, and deployment view. The method consist in decomposing the system recursively into subsystems and then into components.

Other SADMs found in the literature are:

- **Quality Attribute-oriented Software ARchitecture (QASAR, J. Bosch, 2000) [61]** is a method performed in three steps. First, the functional requirements are implemented in components, then the architecture is analyzed to decide whether the quality requirements (NFRs) are fulfilled or not, and in the third step the architecture is adapted to be in conformance with the quality requirements. The second and third steps are repeated till the whole system is in conformance.

- **Quality-driven Architecture Design and quality Analysis (QADA, M. Matinlassi et al., 2002) [62]** is a set of methods that include a method for selecting an appropriate family architecture approach, a method for quality-driven architecture design, a method for evaluating the maturity and quality of the family architecture, and a technique for representing variation points in the family architecture.

- **Quality Achievement at the Architectural Level (AQUA, H. Choi et al., 2006) [63]** is a method that provides software architects means for achieving quality attributes at the architectural level. AQUA involves two kinds of activities, which are architectural evaluation and transformation.

- A. Bertolino et al. (2005) [64] presented an approach to automate the architecture design and implementation. The method starts from requirements in Natural Language (NL). The authors say that they want to integrate several existing tools to accomplish the task: QuARS (Quality Analyzer for Requirements Specifications, tool to obtain requirements from NL specifications), ModTest (a model-checking tool), and Cow Suite (a testing tool).

- T. Al-Naeem et al. (2005) [65] proposed a method centered on the decision making process, but not on generating the architecture. The method uses Analytic Hierarchy
Process (AHP) to score each alternative decision. The author says that other Multiple Attribute Decision Making (MADM) methods could be used instead of AHP.

- L. Chung et al. (2003) [66] proposed a framework, Proteus, to develop software architectures considering NFRs in goal-oriented notation, using NFR Framework [4].

Many other SADMs are just improvements or specializations of the previous ones, some examples are:

- S. Bode et al. (2009) [67] presented a method based on QASAR to design the system’s security architecture. The authors state that they considered methods form software engineering and security engineering to deal with security requirements.

- S. Kim et al. (2009) [68] presented a method that is based on architectural tactics. Architectural tactics are explained in L. Bass et al. book (2003) “Software architecture in practice, second edition” [69], they are basically reusable pieces of the architecture. This method uses feature models to generate the architecture automatically. It is very similar to a product line for architectures.

- D. Perovich et al. (2009) [70] presented a method to design software architectures, ATRIUM, using MDD considering quality aspects (based on ADD method). In this case they use a “megamodel” (a model composed of models) to represent the software architecture. The method uses feature models to construct the architecture.

- E. Niemelä and A. Immonen (2007) [71] presented the QRF method, this method extends QADA by providing a systematic method for eliciting and defining quality requirements, tracing and mapping these requirements to architectural models and for enabling quality evaluation.

- F. Bachmann et al. (2005) [72] proposed an improved reasoning framework for ADD method (first version). The authors distinguish between architectural model and quality attribute model and characterize the actions that a reasoning framework undertakes as basic architectural transformations.

### Analysis

The following table summarizes the SADMs that consider NFRs.

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Name</th>
<th>Kind of NFR</th>
<th>Automation</th>
<th>Based on…</th>
</tr>
</thead>
<tbody>
<tr>
<td>[58]</td>
<td>ATAM</td>
<td>Quality aspects</td>
<td>No, thought for human</td>
<td>SAAM evol.</td>
</tr>
<tr>
<td>[59]</td>
<td>ADD</td>
<td>Quality aspects</td>
<td>No, thought for human</td>
<td>Nothing</td>
</tr>
<tr>
<td>[60]</td>
<td>QASAR</td>
<td>Quality aspects</td>
<td>No, thought for human</td>
<td>Nothing</td>
</tr>
<tr>
<td>[62]</td>
<td>QADA</td>
<td>Quality aspects</td>
<td>Yes, but as product lines</td>
<td>Nothing</td>
</tr>
<tr>
<td>[63]</td>
<td>AQUA</td>
<td>Quality aspects</td>
<td>Somehow, transformations</td>
<td>Nothing</td>
</tr>
<tr>
<td>[64]</td>
<td>No name</td>
<td>Requirements</td>
<td>Yes, but no details are given</td>
<td>Nothing</td>
</tr>
<tr>
<td>[65]</td>
<td>ArchDesigner</td>
<td>Quality aspects</td>
<td>Yes, but limited to decisions</td>
<td>Nothing</td>
</tr>
<tr>
<td>[66]</td>
<td>Proteus</td>
<td>Any NFRs</td>
<td>Not mentioned</td>
<td>NFR Framework</td>
</tr>
<tr>
<td>[67]</td>
<td>No name</td>
<td>Security</td>
<td>No, thought for human</td>
<td>QASAR</td>
</tr>
<tr>
<td>[68]</td>
<td>No name</td>
<td>Any NFR</td>
<td>Yes, but as product lines</td>
<td>[69]</td>
</tr>
<tr>
<td>[70]</td>
<td>ATRIUM</td>
<td>Any NFR</td>
<td>Yes, MDD method, features</td>
<td>ADD</td>
</tr>
<tr>
<td>[71]</td>
<td>QRF</td>
<td>Quality aspects</td>
<td>Yes, but as product lines</td>
<td>QADA</td>
</tr>
<tr>
<td>[72]</td>
<td>No name</td>
<td>Quality aspects</td>
<td>Somehow, decision making</td>
<td>ADDv1</td>
</tr>
</tbody>
</table>

There are many SADMs that use the concept of product line to design new architectures depending on the stated quality requirements. It is interesting to see that almost all are capable to treat with any quality aspect or NFR. It seems to be more common to speak about quality
aspects than NFRs in this area. SADMs that are based on other SADMs are more specific and are more oriented to facilitate the automation of the method.

There are many SADMs that consider NFRs and even some of them are able to generate an architecture in a semi-automatic way. In this situation it seems more feasible to extend or modify an existing SADM than designing a new one. My focus will be on semi-automatic capabilities and decision-making improvements.

3.3. Architectural Knowledge

In the last decade, the essential role that Architectural Knowledge (AK) plays in any software development process has been recognized. More concretely, the concept of Architectural Design Decisions (ADDs, a.k.a. architectural decisions) has been identified as a key concept of AK. The importance of ADD concept is reflected in [73-75].

"ADDs…] are defined as those Decisions that are assumed to influence the Architectural Design and can be enforced upon this Architectural Design, possibly leading to new Concerns that result in a need for taking subsequent decisions. […] Note that architectural design decisions need not necessarily be ‘invented’ by the architect himself; architectural patterns, styles, and tactics are examples of architectural design decisions (or, more precisely, alternatives) that are readily available from other sources", R. de Boer et al., “Architectural Knowledge: Getting to the Core”, 2007 [76].

This recognition has triggered several actions in the research community like having dedicated tracks in software architecture conferences, special issues in journals, and dedicated events as the workshop on SHAring and Reusing architectural Knowledge (SHARK).

This section is divided in three parts: AK ontologies, AK tools, and AK in MDD.

3.3.1. Ontologies for AK

Several ontologies have been published to reflect the organization of AK, each of them with different nuances, but all of them making a special emphasis on the notion of ADDs. In my opinion, a good taxonomy of ADDs was proposed by P. Kruchten et al. (2006) [10] (it was previously published in 2004 [74]). In this taxonomy ADDs are classified into: existence decisions (e.g. need, depend, incompatibility of some component of the system), property decisions (e.g. a condition over a property of the system), and the executive decisions (e.g. the selection of one or more components to be used).

Examples of the mentioned ontologies can be found in [73, 76-78]:

- A. Jansen et al. (2005) [73]: This ontology is based in composing the software architecture. Decisions imply the use of fragments that compose components. Other concepts that appear are connectors, interfaces and ports.
- R. de Boer et al. (2007) [76]: In this case the AK ontology includes development behavior (e.g. stakeholders, activities, artifacts, etc.). Another interesting thing is that decisions are treated as alternatives.
- P. Avgeriou et al. (2007) [77]: The ontology proposed treats decisions as options. Also, decisions have rationale and concerns associated.
- R. Capilla et al. (2007) [78]: In this case the AK ontology is divided in three big groups. The first group is project concepts, where are represented stakeholders, iterations, requirements, and views of the architecture. Second is architecture that is related with variation points, patterns and styles. And the third group is decisions; in
this case the types of decisions are variation points, patterns and styles. Also, for each decision there could be some constraints and rationale associated.

- C. López et al. (2008) [79]: The proposed ontology is focused on NFRs. The ontology is thought to be used with Soft-goal Interdependencies Graphs (SIGs). It is not clear where in this ontology the architectural concepts are defined, but the authors state that it is intended to this use.

- A. Akerman and J. Tyree (2006) [80]: The ontology have architectural concepts that are called "architectural-asset", these concepts are clearly separated from decisional concepts. The detail of the architectural concepts is slightly better than the previous ontologies. Concerns (soft-goals) are addressed by architectural decisions, these, in turn, are implemented by “road-maps” and ADDs transform the architecture.

**Analysis**

The following table summarizes the AK ontologies mentioned in this section.

<table>
<thead>
<tr>
<th>ADDs are alternatives</th>
<th>Development process</th>
<th>ADDs are related with</th>
</tr>
</thead>
<tbody>
<tr>
<td>[73] No</td>
<td>No</td>
<td>Fragments</td>
</tr>
<tr>
<td>[76] Yes</td>
<td>Yes</td>
<td>Nothing</td>
</tr>
<tr>
<td>[77] Yes</td>
<td>No</td>
<td>Rationale and concerns</td>
</tr>
<tr>
<td>[78] No</td>
<td>Yes</td>
<td>Rationale and constraints</td>
</tr>
<tr>
<td>[79] No</td>
<td>No</td>
<td>ADDs are not represented</td>
</tr>
<tr>
<td>[80] Are related</td>
<td>No</td>
<td>Alternatives, concerns and assumptions</td>
</tr>
</tbody>
</table>

With the exception of the ontology proposed by A. Jansen et al. and the ontology proposed by C. López et al. all other ontologies have many similarities, but none of the studied ontologies fulfills the two underlying needs of a semi-automatic process, a formalism to be understood by a computer and enough detail to design the architecture. The intention of the proposed thesis is to be able to semi-automate the decision making process, and even the design of some parts of the architecture. This is the reason why I have already started to develop my own ontology, Arteon [21] (see annexed papers). Arteon takes ideas from many of the studied ontologies and adds the needed detail for a semi-automatic process.

### 3.3.2. AK tools

There are, already, many tools to manage AK, these tools have the objective of codifying, storing and reusing AK. A. Tang et al. published a comparative of five AK tools, and the name they use for architectural concepts [81]. Another comparison of AK tools but only for Open Source Software (OSS) was presented by K. Henttonen an M. Matinlassi [82]. There is also another publication [83] that compares architectural design decision tools and ways to model the decisions.

The tools that appear in these comparisons are: AEvol [84], Ontology-Driven Visualization [85], Archium [73], ADDSS [86], AREL [87], Knowledge Architect [88], PAKME [89], Web of Patterns [90], Stylebase for Eclipse [91], and Morpheus [92]. Some interesting facts of some of these tools are:

- **Ontology-Driven Visualization**: Uses the ISO/IEC 9126 to classify NFRs.
- **AREL**: takes in consideration NFRs as one of the elements of the architecture design. This tool helps in the design of the architecture using UML models and views.
- **PAKME**: NFRs can be specified as keywords of architectural patterns that then can be reused for other projects. This tool is limited to textual knowledge.
• Stylebase for Eclipse: Is a plug-in for Eclipse, that is capable to generate code for some architectural patterns, each pattern have a model associated (an image not a real model) and the principal NFRs that are improved (but in a very limited way).
• Morpheus: Uses NFRs as constraints over functional requirements that then conditions the software architecture. It is presented as a MDD method that starts from requirements using goal oriented notations.

Analysis
The following table summarizes the mentioned AK tools in this section.

<table>
<thead>
<tr>
<th>Table 6: Comparison of AK tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
</tr>
<tr>
<td>AEvol</td>
</tr>
<tr>
<td>ODV</td>
</tr>
<tr>
<td>Archium</td>
</tr>
<tr>
<td>ADDSS</td>
</tr>
<tr>
<td>AREL</td>
</tr>
<tr>
<td>Knowledge Architect</td>
</tr>
<tr>
<td>PAKME</td>
</tr>
<tr>
<td>Web of Patterns</td>
</tr>
<tr>
<td>Stylebase for Eclipse</td>
</tr>
<tr>
<td>Morpheus</td>
</tr>
</tbody>
</table>

Most of these tools are discontinued or used just a proof of concept of a published paper. All of them are the result of an academic research.

3.3.3. MDD approaches that integrate AK
As we have seen in the previously that only one tool is integrated in a MDD process, but there at least some works that considers theoretically this integration. The approaches I found are:

• A. Mattsson et al. (2009) [93] made an empirical study on a real-world project using MDD and found several deficiencies on the ability of MDD to treat with architectural design rules.
• D. Perovich et al. (2009) [70] (see section 3.2.3).
• O. Zimmermann et al. (2008) [94] proposed an approach to model architectural decisions for SOA systems following the MDA process.
• E. Navarro and C. E. Cuesta (2008) [95] treated issues of traceability of the architectural design decisions inside of MDD process. This approach uses the previous mentioned AK tool that is integrated in a MDD process, Morpheus [92].

Analysis
The following table summarizes the MDD approaches that integrate AK mentioned in this section.

<table>
<thead>
<tr>
<th>Table 7: Comparison of MDD approaches that consider architectural aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref.</td>
</tr>
<tr>
<td>[93]</td>
</tr>
<tr>
<td>[70]</td>
</tr>
<tr>
<td>[94]</td>
</tr>
<tr>
<td>[95]</td>
</tr>
</tbody>
</table>
First thing to notice is that all these works are recent in time, an indication that this research topic is just in its beginning. Second, the same issues mentioned before are present here.

- MDD needs more detailed knowledge. The knowledge treated in this approaches is limited to high level concepts such as style or pattern. This makes difficult to combine and adapt solutions. The needed level of detail is not present in the mentioned ontologies (see section 3.4.1), probably because it adds a complexity that is not necessary if the knowledge is not planned to be reused in a MDD context.
- MDD needs some kind of formalism for AK. Most of the previous tools (see section 3.4.2) store knowledge as text thought to be understandable by humans. This makes the knowledge almost useless in a MDD process.

These issues should be resolved in order to be able to effectively guide selection of architectural transformations in a semi-automatic MDD process.

3.3.4. Architectural Description Languages

We have seen many tools and ontologies to share and reuse AK, but if we look back to the definition of AK, we found that in most cases ADDs are covered but the architecture design itself is not always part of this knowledge.

**Architectural Knowledge = Design Decisions + Design**

Architectural Description Languages (ADLs) are Domain Specific Languages (DSLs) that are thought to represent a concrete architecture of a software product. This is the formalization needed in a semi-automatic software development process such as MDD.

Well-known ADLs in software architecture community are:

- **ACME (D. Garlan, 1997)** [96], ACME started in 1995 and the last update dates from 2009, three major versions of the ADL have been released. The intention of the language is to provide means to interchange architectural information. Components, connectors, and ports are the principal elements of this language.
- **Rapide (D. C. Luckham, 2001)** [97] is an Executable ADL (EADL) that is designed to support component-based development of large, multi-language systems. This ADL uses an event-based execution model.
- **Architecture Analysis & Design Language (AADL, P. H. Felier, 2003)** [98]. “[AADL] is a textual and graphical language used to design and analyze the software and hardware architecture of real-time systems and their performance-critical characteristics. It is aimed at supporting the avionics, aerospace, and automotive industry. The language is used to describe the structure of such systems as an assembly of software components mapped onto an execution platform. The language can describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing)”. AADL is based on another ADL called MetaH.
- **Chiron-2 (C2, R. N. Taylor, 1995)** [99] is a software architecture style for user interface intensive systems. The ADL used to define this architectural style is called C2SADEL, but it is commonly referred as C2. This style consists of software components and connectors. Connectors transmit messages between components. Components maintain state, perform operations, and exchange messages with other components through its top and bottom interfaces.
State of the art

- Darwin (J. Magee et al., 1995) \[100\] is a notation for the high-level organization of computational elements and the interactions between those elements. “Darwin is in essence a declarative binding language which can be used to define hierarchic compositions of interconnected components. Distribution is dealt with orthogonally to system structuring. The language supports the specification of both static structures and dynamic structures which may evolve during execution. The central abstractions managed by Darwin are components and services. Services are the means by which components interact”. The \(\pi\)-calculus formalization is used in semantics of Darwin.

- \(\pi\)-ADL (a.k.a. pi-ADL, F. Oquendo, 2004) \[101\] “[\(\pi\)-ADL] is a formal, well-founded theoretically language \(\pi\)-calculus. While most ADLs focus on describing software architectures from a structural viewpoint, \(\pi\)-ADL focuses on formally describing architectures encompassing both the structural and behavioral viewpoints.”

- Wright (R. Allen and D. Garlan, 1997) \[102, 103\] is a formal and textual ADL used to analyze and specify architectures. Its first-class elements are components and connectors. In Wright semantic of architectural connections are formalized.

Some of these and other ADLs were classified and compared by N. Medvidovic and R. N. Taylor (2000) \[104\]. This work is a bit old, but to my knowledge there is not any newer comparison of ADLs. The full list of compared ADLs is: ACME, Aesop, C2, Darwin, MetaH, Rapide, SADL, UniCon, Weaves, Wright. AADL and \(\pi\)-ADL are not present because they were published in a later publication.

In the recent literature I found several works related to ADLs and Aspect-Oriented principles, examples can be found in \[105-109\]. J. Perez et al. (2006) \[109\] presented an ADL that combines Component-Based Software Engineering (CBSE) and Aspect-Oriented Software Development (AOSD). B. Wang et al. (2008) \[106\] extended the Wright ADL to support Aspect Orientation (AO). W. Jing et al. (2008) \[107\] extended C2 ADL to support AO. A. Navassa et al. (2009) \[108\] presented an architectural framework, and an ADL called AspectLEDA, the paper justifies the need of a new ADL for AO because extending the existing ADLs does not respect the obliviousness principle of AO. W. Duan et al. (2009) \[105\] proposes an ADL to specify AO in the connector element of the ADL.

I found two works that tackle about SOA specific description languages, T. Zhang et al. (2008) \[110\] define SO-ADL that allows to service components, connectors and compositions having in mind the inherent reusability and dynamic nature of SOA systems. M. Lopez-Sanz et al. (2008) \[111\] present an ADL based on \(\pi\)-ADL for SOA systems, in this case the proposed ADL is related to the PIM level of MDA.

Last, but not least important, UML became the de facto standard for software architecture specifications in industrial practice. N. Medvidovic et al. (2002) \[112\] proposed and compared two strategies to use UML as ADL, first is to use UML as is, and second is to use UML with extensions that integrate the expressivity of the existing ADLs. UML 2.0 (2003) was made public one year later, It included several new facilities to specify the architecture of a software system. These facilities were explored in the same year by M. Björkander and C. Kobryn (2003) \[113\]. Nowadays there is some controversy about the use of UML as an all-in-one language.
Analysis

The following table summarizes the ADLs mentioned in this section.

**Table 8: ADLs comparison**

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Name</th>
<th>Scope</th>
<th>NFRs</th>
<th>Last Pub.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[96]</td>
<td>ACME</td>
<td>Architectural interchange</td>
<td>Properties</td>
<td>2009 *</td>
</tr>
<tr>
<td>[97]</td>
<td>Rapide</td>
<td>Simulating dynamic behavior</td>
<td>None</td>
<td>2002</td>
</tr>
<tr>
<td>[98]</td>
<td>AADL</td>
<td>Analysis and Design</td>
<td>Quality</td>
<td>2010 *</td>
</tr>
<tr>
<td>[99]</td>
<td>C2</td>
<td>Distributed architectures</td>
<td>None</td>
<td>1999</td>
</tr>
<tr>
<td>[100]</td>
<td>Darwin</td>
<td>Distributed architectures</td>
<td>None</td>
<td>1997</td>
</tr>
<tr>
<td>[101]</td>
<td>π-ADL</td>
<td>Dynamic architectures</td>
<td>None</td>
<td>2006</td>
</tr>
<tr>
<td>[102]</td>
<td>Wright</td>
<td>Formal specifications</td>
<td>None</td>
<td>1998</td>
</tr>
<tr>
<td>[108]</td>
<td>AspectLEDA</td>
<td>Aspect-Oriented Architecture</td>
<td>As aspects</td>
<td>2009</td>
</tr>
<tr>
<td>[110]</td>
<td>SO-ADL</td>
<td>Service-Oriented Architecture</td>
<td>None</td>
<td>2008</td>
</tr>
<tr>
<td>[112]</td>
<td>UML</td>
<td>Generic</td>
<td>Profiles</td>
<td>2009 *</td>
</tr>
</tbody>
</table>

* Active ADL, the year is the last publication about the ADL.

This is an overview of the most significant (but not all) ADLs. The currently active ADLs have been studied: ACME, AADL, and UML. ACME is used as interchange language for architectural specifications. AADL is used for large and complex architectures that need analysis. In the last years some new ADLs emerged because UML does not cover all its necessities (e.g. AspectLEDA for AO necessities and SO-ADL for SOA necessities).

3.4. Approaches dealing with MDD, NFRs and AK

As a summary of the whole state of the art presented in this thesis proposal, I would like to mention again the ideas that have a relation with the proposed thesis: Systematic Architectural Design (SAD). In Figure 4 are represented the contributions of the proposed thesis, the diagrams also show the relationship of each contribution with the research areas studied in the state of the art: MDD, NFRs and AK.

For the notation of NFRs (Figure 4-a) I embrace two possibilities, using my own metamodel that will contemplate also architectural aspects or use some GORE notation such as $i^*$. The first possibility could be based on Arteon ontology.

The MDD approaches based on the application of patterns are on the same direction as my proposed thesis. But it seems that the most recent publications are based on analysis of the NFRs instead of using them as a driver of the system development. The MDD approach will act as the supporting framework for the proposed methodology (Figure 4-b).

Almost all SADMs are considering NFRs, the difference is that I’m trying to define a method able to be used systematically and in a semi-automatic way. Studied SADMs with semi-automatic approach are based on the ideas of product-lines. These methods are very limited in its application because of the allowed variability in product-lines. I would like to give more attention to the semi-automatic and assisted decision-making process instead of the production process (Figure 4-c).

In order to semi-automate the decision making process for architectural design it is necessary to have a well founded AK structured on a well defined ontology (Figure 4-d). This ontology will need a support tool to add or modify the knowledge.

Having selected SOA as the focus for the proposed thesis will have an impact to the whole thesis, for example in the adoption of an ADL for the representation of the generated architecture will must be an adequate ADL for SOA. Another possibility is to adhere to UML and its extensions.
Figure 4: Contributions of the proposed thesis
4. Working plan

I have identified 9 tasks, 6 of them stem from the stated goals of the thesis (see section 2.1), and the rest of tasks are the habitual tasks on every thesis (e.g. validation, writing the thesis, etc.).

Table 9: Task list

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Empirical research based on interviews to architects. This will help to make a final decision over the restrictions of the thesis including the selection of NFRs (G1), architectural views (G1), and the initial business model. Obtained information of this work will be also used to partially fill the knowledge base (G6).</td>
</tr>
<tr>
<td>F1</td>
<td>Design and produce a complete specification of the framework (G2 and G4).</td>
</tr>
<tr>
<td>F2</td>
<td>Develop the framework.</td>
</tr>
<tr>
<td>O</td>
<td>Refine the ontology with improvements and necessary adaptations (G5).</td>
</tr>
<tr>
<td>KB</td>
<td>Fill knowledge base of architectural and technological aspects (G6).</td>
</tr>
<tr>
<td>M</td>
<td>Define the methodology to produce software architectures (G3).</td>
</tr>
<tr>
<td>SLR</td>
<td>Maintain an up-to-date state of the art running a continuous systematic literature review.</td>
</tr>
<tr>
<td>V</td>
<td>Validation of the methodology with a use case in some IT company.</td>
</tr>
<tr>
<td>T1</td>
<td>Write the thesis.</td>
</tr>
<tr>
<td>T2</td>
<td>Design the presentation of the thesis.</td>
</tr>
</tbody>
</table>

4.1. Calendar

All the tasks are represented in the timeline (Figure 5). The thesis is planned for two years but administrative issues could cause at the most six month deviation at the end of the planning. Also, the timeline does not consider holidays or external factors that could delay the thesis (e.g. stays on foreign universities, non-related work for the department, etc.). It should be seen as two years of full dedication.
4.2. Methodology
An iterative methodology will be used for the thesis. The methodology will consist in four iterations: inception, incubation, implementation, and validation.

- The first iteration, inception, is already done with the Master Thesis [18] and this Thesis Proposal. This iteration globalizes the work done for the state of the art, and the work done to find gaps in the current research and find possible ways of research in these areas.
- The second iteration, incubation, refers to the work that will be done in design and specification, and also the work to obtain information for the knowledge base. The second iteration is already partially done.
- The third iteration, implementation, is centered in obtaining results. It is a sum of implementing the proposed framework plus the introduction of the knowledge base obtained in the previous iteration. In this iteration the theoretical part of the method should be done.
- The fourth iteration, validation, will focus on proofing that the results obtained are consistent. To do that, the method will be used in some company project as an empirical validation mechanism. The implementation efforts in this iteration will be related to the possible detected flaws during the validation.

In the following figure are shown the three resting iterations and the estimated dedication (in percentage) to each task.

![Figure 6: Dedication and iterations](image)
References


References:


## Table of acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADL</td>
<td>Architectural Description Language</td>
</tr>
<tr>
<td>AHP</td>
<td>Analytic Hierarchy Process</td>
</tr>
<tr>
<td>AK</td>
<td>Architectural Knowledge</td>
</tr>
<tr>
<td>AO</td>
<td>Aspect Orientation</td>
</tr>
<tr>
<td>AOSD</td>
<td>Aspect-Oriented Software Development</td>
</tr>
<tr>
<td>BPEL</td>
<td>Business Process Execution Language</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Modeling Notation</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>GORE</td>
<td>Goal-Oriented Requirements Engineering</td>
</tr>
<tr>
<td>M2M</td>
<td>Model to Model Transformation</td>
</tr>
<tr>
<td>M2T</td>
<td>Model to Text Transformation</td>
</tr>
<tr>
<td>MDA</td>
<td>Model-Driven Architecture</td>
</tr>
<tr>
<td>MDD</td>
<td>Model-Driven Development</td>
</tr>
<tr>
<td>MDE</td>
<td>Model-Driven Engineering</td>
</tr>
<tr>
<td>NFR</td>
<td>Non-Functional Requirement</td>
</tr>
<tr>
<td>NL</td>
<td>Natural Language</td>
</tr>
<tr>
<td>OSS</td>
<td>Open Source Software</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QVT</td>
<td>Query/View/Transformation</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>SADM</td>
<td>Software Architecture Design Method</td>
</tr>
<tr>
<td>SIG</td>
<td>Soft-goal Interdependency Graph</td>
</tr>
<tr>
<td>SLR</td>
<td>Systematic Lecture Review</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architecture</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
Thesis Proposal

Annexed papers
Assigning Treatments to Responsibilities in Software Architectures

David Ameller, Xavier Franch
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
nail2001@gmail.com, franch@lsi.upc.edu

1. Introduction

According to the methodology proposed by Craig Larman in the context of Object-Oriented (OO) specification and design [1], designing a software system architecture basically means to assign treatments to responsibilities. Responsibilities are derived from the specification of the system, examples are: obtaining the element identified by some key; updating an attribute from a class; or ensuring some consistency constraint. Treatments are technological solutions to fulfill responsibilities. For example, to update an attribute, we may use pure OO code, or a trigger or a stored procedure in the data base. Assigning a particular treatment to a responsibility is made on the basis of some non-functional properties, such as efficiency, portability or maintainability.

In this work we present a framework aimed at supporting the assignment of treatments to responsibilities. In section 2 we present the theoretical work, Responsibility Detection and Transformation (RDT); whilst in section 3 we give details of our current implemented work based on the AndroMDA tool, Assignment of Responsibilities in a 3-Layered architecture (AR3L).

2. The RDT framework (theoretical work)

Fig. 1 shows the RDT framework. The main components are the Responsibility Detection and Responsibility Transformation tools. Besides, we include different editors to make the framework highly customizable. The framework may be composed to any tool able to generate code from an architectural composition; this kind of tools are gaining importance in the last years in the context of Model-Driven Development (MDD) [2]. Note that our framework is open in the sense that it is designed to be interoperable with as much of them as possible.

The Responsibility Detection tool is able to extract responsibilities from one or more departing models. In our framework we do not fix the kind of model, it may be a heavyweight UML model [3] with class, sequence and state diagrams; just a description of user interface in an agile development context; or an IEEE-830-compliant system requirements document. Depending on the type of model, responsibility detection may be more or less automated.

Fig. 1: The RDT framework

The Responsibility Transformation tool is able to find the most appropriate treatments, according to the non-functional requirements of the project, to be assigned to the identified responsibilities. For instance, if efficiency is a high-priority requirement, then the stored procedure may be the better treatment for the attribute updating responsibility, but if portability is preferred, then pure OO code is better.

The editors make possible to maintain the responsibilities and treatments repositories. This is utterly important especially in the second case, because treatments are highly dependant on the technology state-of-the-art. Also we allow defining several architectural patterns as target. Examples would be typical patterns.
such as layered, pipe and filters or blackboard; or other emerging ones, such as service-oriented architectures.

3. The AR3L framework (implemented work)

Our first, current implementation of the RDT is AR3L [4], it makes the framework concrete because of:

- A technological choice. We have selected one existing tool, AndroMDA [5], for implementing the framework. This is a powerful, fully customizable tool that is currently used in several projects.

- Some decisions on the models. Since this is a proof of concept, we have decided to restrict this first prototype to: UML class diagrams (including operations) for writing specifications; 3-layer architectures as architecture model.

The 3-layer architectural pattern [6] is one of the most, if not the most, used patterns in information systems nowadays, therefore it seems a good candidate for this first prototype. It structures the system into the following layers: presentation (in charge of communicating with the user), domain (with all the business logic) and data (taking care of data management and persistence issues). Therefore, the assignment of treatments to responsibilities means: extract responsibilities from UML class diagrams, and decide how these responsibilities are split into the layers as a consequence of the application of the treatments that best fulfill the non-functional requirements stated in requirements document. Afterwards, a code generator may be used to obtain final software but in the current status of the work this step must be done manually mapping the list of responsibilities with treatment and layer to the initial UML model. Fig. 2 shows the result of this approach.

As an example, consider one operation in the UML specification called deleteBook(ISBN: String) which is in charge of deleting a book. A precondition of this operation is that the book exists. In other words, the responsibility is “A book identified by an ISBN must exist”. This responsibility belongs to a type of responsibilities from the repository (we have more than 100 types), the type “operation->pre->deletion->elementsExists”. For that type of responsibility, we have several types of treatments, among them “PresentationProtectedDeletion”, “DomainProtectedDeletion” and “DataProtectedDeletion”. In the first case, the presentation layer shows a combo box with all the books’ ISBN that are in the system and the user chooses one; in the second case, the domain layer checks if the ISBN exists before deleting; in the third case, the data layer raises an exception if the deletion is executed (by SQL) on a non-existing book. Again, if we have efficiency as a non-functional requirement, the third option will be most likely chosen by the system.

![Fig. 2: The AR3L framework](image)

4. References


Asignación de Tratamientos a Responsabilidades en el contexto del Diseño Arquitectónico Dirigido por Modelos

David Ameller, Xavier Franch
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
c/ Jordi Girona, 1-3
08034 Barcelona
{dameller, franch}@lsi.upc.edu

Resumen
Una de las principales actividades en el desarrollo de Sistemas de Información (SI) es la asignación de tratamientos a responsabilidades durante la etapa de diseño. En este artículo, presentamos el marco Responsibility Detection and Transformation (RDT) para la semi-automatización de este proceso aplicable al contexto del desarrollo dirigido por modelos (MDD). RDT ofrece la posibilidad de partir de diversos tipos de artefactos (específicaciones OO, interfaces gráficas, etc.) para describir el sistema software. Para llevar a cabo el proceso de asignación de tratamientos a responsabilidades, RDT selecciona el tratamiento más adecuado de un repositorio de tratamientos, en base a los requisitos no funcionales establecidos sobre el SI en desarrollo. A continuación, presentamos el prototipo actual de la implementación de RDT, llamado Assignment of Responsibilities in a 3-Layer architecture (AR3L), en el que usamos una plataforma de código abierto llamada AndroMDA para implementar el proceso, especificaciones en UML como artefacto a partir del que se extraen las responsabilidades, y arquitectura en tres capas como patrón arquitectónico del SI.

1. Introducción
La fase de diseño de un SI puede contemplarse como una actividad de asignación de tratamientos tecnológicos a las responsabilidades identificadas en el SI. Según [6], “una responsabilidad contiene uno o más propósitos u obligaciones de un elemento”, por ejemplo, la responsabilidad de asegurar que no pueden existir en el sistema dos instancias de una clase con un mismo identificador. Por otro lado, un tratamiento es una acción asignada a una responsabilidad con el fin de que ésta sea satisfecha, p.e., definir una clave primaria en el esquema de base de datos nos asegura que la responsabilidad anterior queda satisfecha. Los tipos de responsabilidades son bastante estábles y pueden deducirse de diversos tipos de artefactos (especificaciones OO, interfaz de usuario, etc.). En cambio, el repertorio de tratamientos disponible viene dado por la oferta actual de tecnologías (Hibernate, EJBs, Swing, etc.), lenguajes de programación (Java, C++, C#, etc.), estándares (XML, etc.), ..., por lo que tiene un carácter más volátil.

El diseño basado en aplicar tratamientos a responsabilidades fue propuesto a finales de los 80–principios de los 90 [11] y posteriormente adoptado de una manera u otra por diversas metodologías de gran divulgación, especialmente en el ámbito OO, que se basan en esta idea para construir la arquitectura de los SI a partir de su especificación: RUP [5], Larman [6], Fowler [3], etc. En este artículo formulamos una propuesta para semi-automatizar el proceso de asignación de tratamientos a responsabilidades en el marco del desarrollo dirigido por modelos (MDD). El MDD es un paradigma de desarrollo de software que ha ido ganando aceptación en los últimos años. Según [8], “El desarrollo por modelos es simplemente la capacidad de crear un modelo del sistema y convertirlo en algo real”. La detección y asignación de responsabilidades se puede considerar un caso del MDD, siendo necesario definir qué es el modelo del sistema, qué es algo real y cómo se ha implementado la capacidad:

• El modelo del sistema es un conjunto de artefactos (eventualmente con cierto solapamiento) que describen el comportamiento del sistema. Un ejemplo típico sería una especificación OO que, según
[6], está compuesta de un diagrama de clases con restricciones de integridad, un modelo de comportamiento y un modelo de estado. Pero también la interfaz del usuario, donde la existencia de elementos como desplegables, campos numéricos, etc., determina ciertas responsabilidades a cubrir.

- **El algo real** es el conjunto de tratamientos aplicados a cada responsabilidad detectada en los artefactos de partida. Estos tratamientos pueden ser eventualmente transformados en código pero no trataremos esta problemática en este artículo ya que, como veremos en la próxima sección, nuestro objetivo es obtener un modelo preparado para ser usado por herramientas de MDD capaces de generar código.

- **La capacidad** se ha implementado como un proceso de dos pasos. El primer paso identifica las responsabilidades existentes en los artefactos de partida y el segundo selecciona los tratamientos más adecuados para cumplir los requerimientos no funcionales del sistema. Ambos pasos son semi-automáticos. En el caso de la detección, será más automático mientras más preciso sea el modelo de partida. En el caso de la selección, mientras más detallados sean los requisitos no funcionales.

El resto del artículo se estructura de la manera siguiente. En la sección 2 presentamos el marco genérico RDT. En la sección 3 particularizamos RDT para un caso particular, dando lugar a AR3L. En la sección 4 mostramos un escenario de uso para ilustrar los conceptos principales. En las secciones 5 y 6 presentamos la forma en que trabajamos con responsabilidades y tratamientos. En la sección 7 sumarizamos los detalles de implementación del prototipo actual. Finalmente, en la sección 8 damos un resumen y exponemos las vías de trabajo futuras.

2. **El marco RDT**

En esta sección presentamos el marco genérico *Responsibility Detection and Transformation* (RDT) para la asignación de tratamientos a responsabilidades. Para ello, vamos a motivar su utilidad en el contexto del MDD y, a continuación, daremos algunos detalles de su arquitectura.

En la figura 1, parte superior, presentamos la arquitectura habitual de los métodos MDD actuales. En resumen, dichas propuestas proponen la generación de código en un lenguaje determinado a partir de un cierto tipo de artefacto de entrada para un patrón arquitectónico concreto [2]. Por ejemplo, AndroMDA o ArcStyler permiten la transformación entre modelos PIM y PSM y la generación de código, limitándose a UML [10] como entrada y Java o .NET como salida. En [7], se propone usar el estándar UsiXML como PIM para diseñar un modelo abstracto y pasando a un modelo concreto, pueden generarse interfaces de usuario orientadas a la Web.

En la figura 1, parte inferior, mostramos el encuadre de RDT en dichas propuestas. Destacamos dos características principales. Primero, el marco RDT tiene como objetivo reconocer diversos tipos de artefactos de entrada, tanto modelos como otros artefactos menos formales. En todos los casos, RDT va a absorber todas las responsabilidades que detecte en los artefactos de entrada (que pueden ser uno o varios, redundantes o no, etc.), y va a permitir añadir manualmente aquéllas que no aparezcan en los mismos. Segundo, el marco RDT está diseñado para generar diversos tipos de artefactos de salida (PSM) y así permitir conectarse con diversos generadores de código. Este enfoque posibilita plantearse el uso de las propuestas MDD existentes en más situaciones, por ejemplo usar UsiXML tal como se propone en [7] como artefacto de entrada a partir del que se pueden identificar responsabilidades usadas para generar un PSM UML usable por otras herramientas.

![Figura 1: Propuestas MDD sin el marco RDT (parte superior) y con el marco RDT (parte inferior)](image-url)
Dados los objetivos, queda claro que la arquitectura de RDT debe enfrentarse a dos retos: primero, identificar las responsabilidades inherentes al artefacto o artefactos de partida; segundo, decidir qué tratamientos son más adecuados para estas responsabilidades.

En la figura 2 (evolución de la figura 1, parte inferior) presentamos la arquitectura de RDT. El primer proceso, la detección, depende del tipo de los artefactos usados como punto de partida del diseño. El proceso de detección está diseñado para permitir extraer las responsabilidades de un único tipo de artefacto, p.e. un documento de especificación de requisitos o un diseño de interfaz de usuario, o de varios, por ejemplo una especificación UML que conste de diagrama de clases, modelo de casos de uso y contratos de operaciones [6]. La herramienta Responsibility Detection es la encargada de identificar las responsabilidades de dichos artefactos y clasificarlas según los tipos de responsabilidades almacenados en el Responsibility Repository.

Puede pasar (como veremos en este artículo) que los artefactos deban cumplir ciertas convenciones (p.e., notacionales) para explotar al máximo el proceso de detección automática. En todo caso, se puede prever cierto grado de interacción del diseñador para proporcionar información adicional que permita detectar y/o clasificar las responsabilidades (herramienta Responsibility Editor). El resultado del proceso de detección es pues un conjunto de responsabilidades del SI clasificadas por su tipo.

El segundo proceso, la transformación, parte de dicho conjunto de responsabilidades detectadas y las transforma con el fin de que puedan ser gestionadas automáticamente. Esta transformación será efectuada mediante una selección automática de tratamientos a partir de los requisitos no funcionales del sistema software (herramienta Responsibility Transformation). Los tratamientos a aplicar residen en un Treatments Repository y su aplicación dependerá asimismo del patrón arquitectónico que rige el SI (p.e., tres capas, orientado a servicios, etc.). El modelo resultante puede ser de distintos tipos (UML, i* etc), aunque el habitual sería UML, ya que existen más herramientas capaces de transformar modelos en dicho formato. En el supuesto de MDD, las herramientas de generación de código tendrían suficiente información para que el trabajo del desarrollador se redujera considerablemente.

Por último, citemos que la arquitectura de RDT favorece la extensibilidad según diversos criterios: reconocimiento de nuevos tipos de artefactos de entrada o salida, añadido de nuevos tipos de responsabilidades y (especialmente) tratamientos, consideración de nuevos patrones arquitectónicos, etc. Desde un punto de vista conceptual, esta extensibilidad se basa en el uso del concepto de responsabilidad como elemento central en el proceso de diseño.

3. El proyecto AR3L

En esta sección presentamos el proyecto AR3L, que se puede considerar una prueba de concepto del marco RDT para un contexto determinado. Concretamente, AR3L fija: el tipo de artefacto de entrada (especificación UML), el tipo de artefacto de salida (una descripción textual de los tratamientos a aplicar), los tipos de requisitos no

---

1 Los recuadros blancos son herramientas que forman parte del marco RDT, mientras que las sombreadas son aquéllas que ya existen o que simplemente están fuera de los objetivos principales del proyecto. Las elipses sombreadas son artefactos de entrada mientras que las blancas son información transformada.
funcionales considerados (cuatro, relacionados con la usabilidad y la tecnología) y sus posibles valores, y el patrón arquitectónico del SI (arquitectura en tres capas [1] en el marco del diseño OO [6]). La elección de estas variantes se debe a su gran predominancia en el estado del arte actual.

El proyecto AR3L se basa en una herramienta del mismo nombre. Nuestro objetivo ha sido adaptar una plataforma MDD de las muchas existentes [2], AndroMDA, un generador de código basado en una arquitectura dirigida por modelos (MDA) [9]. AndroMDA es una plataforma muy versátil y adaptable; la versión actual, AndroMDA 3.2, da soporte a UML 2.0, está integrada con Eclipse y contiene un conjunto de extensiones que permiten generar código para distintas tecnologías.

En la figura 3, presentamos el prototipo actual de AR3L. La figura refleja que las herramientas de detección y de transformación de responsabilidades se han integrado en una única herramienta, AR3L. En concreto, la detección de responsabilidades se ha implementado usando las facilidades de AndroMDA (en la sección 7 presentamos algún detalle adicional) mientras que la transformación de responsabilidades se ha desarrollado mediante código Java invocado desde AndroMDA. La arquitectura interna de AR3L se ha diseñado lo más desacoplada posible, para que el módulo Java de transformación de responsabilidades pueda reutilizarse en otros contextos.

4. Escenario: Gestión de conferencias

En el resto del artículo usaremos un escenario de uso como ejemplo, un sistema gestor de conferencias. Las conferencias están organizadas en sesiones, los artículos se envían a las conferencias en una fecha. Cada artículo enviado se evalúa como aceptado o rechazado, los aceptados se asignan a sesiones de la conferencia a la cual fueron enviados. La figura 4 muestra el diagrama de clases que representa esta información, como es usual algunas restricciones aparecen de forma gráfica (p.e., cardinalidades), mientras que otras como los identificadores de clase deben expresarse textualmente, en este caso los identificadores son: Paper (title), Congress (name o acronym) y Date (date). Además, no puede haber dos sesiones con el mismo nombre en el mismo congreso. Estas restricciones comentadas y otras pertenecientes a las operaciones forman parte de la especificación y del modelo de comportamiento, ya sea con lenguaje natural, OCL u otro mecanismo de especificación.

5. Responsabilidades

En el proyecto AR3L, clasificamos las responsabilidades en dos categorías que se derivan de los tipos habituales de restricciones en diagramas de clase UML [4].
Las restricciones conllevan un problema fundamental ya que existen múltiples estrategias de detección (notas, estereotipos, valores etiquetados, etc.), especialmente en el caso de las restricciones textuales. De hecho se pueden usar estrategias distintas dependiendo de la responsabilidad, en este artículo nos hemos basado en las siguientes características para decidir qué estrategia es la más adecuada en cada caso:

- **Esfuerzo del especificador.** Trabajo extra llevado a cabo por el especificador, el cual puede ser medido en forma de cantidad y complejidad de la información añadida.
- **Comprensibilidad.** Cada estrategia requiere una información extra que requiere ser comprendida.
- **Expresividad.** Con esta característica tenemos en cuenta si la estrategia provoca pérdida de información y si se contemplan todas las posibilidades.
- **Estandarización.** Con el fin de que nuestra solución sea aplicable a más ámbitos toman preferencia las estrategias basadas en conceptos estandarizados.
- **Dificultad de implementación.** Estimación del trabajo que se tendrá que hacer al implementar la estrategia en AndroMDA.

### 5.1. Identificador de clase

La regla de integridad de identificador de clase permite establecer que no puede haber dos instancias de una clase con el mismo valor para un conjunto de atributos (identificador).

Existen tres tipos de identificador:

- **Key.** El identificador esta compuesto por uno o más atributos de la clase.
- **Weak key.** El identificador esta compuesto por uno o más atributos de la clase conjuntamente con el identificador de otra clase, esta segunda clase estaría unida con la primera mediante composición.
- **Alternative key.** Además de tener una key, la clase puede tener otro conjunto de atributos que identifican dos instancias distintas.

En el ejemplo (Figura 4) la clase Congress tiene un Key y un Alternative key (queda al albur del especificador indicar cual de los dos es el alternativo). Por otra parte la clase Session tiene una Weak key, su identificador está compuesto por el Weak key y el identificador de la clase Congress conjuntamente.

<table>
<thead>
<tr>
<th>Responsibiltades gráficas.</th>
<th>Detectadas en elementos gráficos, como la cardinalidad (de asociaciones, atributos, etc.) y otro tipo de propiedades (herencia, constantes, etc.).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Responsabilidades textuales.</strong></td>
<td>No aparecen en el modelo pero están expresadas textualmente con lenguaje natural o formal (p.e., OCL). De estas últimas destacamos dos tipos:</td>
</tr>
<tr>
<td>Deben cumplirse en cualquier estado válido del sistema (p.e., identificadores de clase, restricciones en atributos de clase). Son representados por invariantes de clase.</td>
<td></td>
</tr>
<tr>
<td>Deben cumplirse cuando se invoca una operación. Existen dos tipos, precondiciones y poscondiciones.</td>
<td></td>
</tr>
</tbody>
</table>

En esta prueba de concepto, hemos implementado en AR3L un ejemplo representativo de cada tipo de responsabilidad: cardinalidad de asociaciones (gráfica), identificadores de clase (textual permanente) y precondiciones y poscondiciones de las operaciones (textual eventual). La tabla 1 ofrece las responsabilidades de estos tres tipos que pueden deducirse del modelo UML de la fig. 4.

| Tabla 1: Lista de responsabilidades del sistema gestor de conferencias |
|---------------------------------|-------------------------------|
| **Responsibility** | **Model Element Name** | **Responsibility description** |
| Paper | An object ‘Paper’ may have at most 1 object ‘Congress’ bound. |
| SentPaper | An object ‘SentPaper’ must have exactly 1 object ‘Paper’ bound. |
| | An object ‘SentPaper’ must have exactly 1 object ‘Congress’ bound. |
| Session | An object ‘Session’ must have exactly 1 object ‘Congress’ bound. |
| | An object ‘Session’ may have at most 5 objects ‘Accepted’ bound. |
| Accepted | An object ‘Accepted’ must have exactly 1 object ‘Session’ bound. |
| Evaluation | An object ‘Evaluation’ must have exactly 1 object ‘Paper’ bound. |
| | An object ‘Evaluation’ must have exactly 1 object ‘Congress’ bound. |
| | An object ‘Evaluation’ must have exactly 1 object ‘SentPaper’ bound. |
| | An object ‘evaluation’ must have exactly 1 object ‘Person’ bound. |
| Cardinality | **Identifier** | **Pre / post** |
| Paper | Title is identifier. |
| Congress | Name and acronym are both identifier. |
| Date | Date is identifier. |
| Session | There cannot be two sessions with the same name in the same congress. |
| Pre: | insertElement: post: Create a SentPaper. |
| insertElement: post: Create a SentPaper. |
| post: | Delete a SentPaper. |
| post: | The Congress must have some Sessions bound. |
| post: | Return the set of all the Accepted Papers. |
| post: | Delete a SentPaper. |
| Accepted | The Congress must have some Accepted Person. |
Para mantener una coherencia los tres tipos de identificadores deben cumplir una serie de restricciones, por ejemplo, una clase puede tener un Key o un Weak key, pero no ambos al mismo tiempo. Estas restricciones deben cumplirse en el modelo de especificación para que éste pueda ser evaluado correctamente.

Existen diversas alternativas para declarar identificadores en un modelo UML. A continuación las presentamos y analizamos. Se basan en tres posibilidades: usar mecanismos de extensión UML (estereotipos o valores etiquetados); incluir información textual en notas o en la clase propiamente (puede ser en texto o en OCL); usar alguna nomenclatura para detectar el tipo de identificador. La figura 5 muestra las diferentes estrategias y la tabla 2 resume nuestro análisis de estas siete estrategias.

- **Class stereotype.** Esta estrategia es muy cómoda para el especificador pero bastante mala en cuanto a expresividad porque no tenemos información de qué atributos componen el identificador. Esto también afecta a la comprensibilidad del modelo.
- **Attribute stereotype.** La clase contiene más información que antes aunque ésta deberá ser introducida por el especificador.
- **Tagged values.** Requiere especificar una sintaxis y obliga al especificador a escribir el nombre de los atributos lo cual puede inducir errores. Además esta estrategia requiere algo más de trabajo en la implementación ya que AndroMDA nos ofrece más facilidades usando estereotipos.
- **Notes.** La primera consecuencia que detectamos es el incremento del tamaño del modelo, por otro lado las notas no son un elemento muy manejable en AndroMDA.
- **Class invariants.** Como ya se ha comentado antes esto supone una serie de problemas en cuanto a la implementación.
- **Attribute naming convention.** Esta estrategia no ofrece una buena comprensibilidad y añadiría otros problemas cuando un atributo forma parte de más de un identificador o cuando hay más de una Alternative key.
- **Class invariant with naming convention.** Procesar las expresiones OCL no es tan costoso como antes aunque aún requiere más trabajo que el resto de estrategias.

![Figura 5: Estrategias alternativas de detección de identificador de clase](image.png)

<table>
<thead>
<tr>
<th>Estrategia</th>
<th>Facilidad</th>
<th>Expresividad</th>
<th>Comprensibilidad</th>
<th>Estandarización</th>
<th>Implementación</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Class stereotype</td>
<td>Alta</td>
<td>Media</td>
<td>Baja</td>
<td>Alta</td>
<td>Alta</td>
</tr>
<tr>
<td>(b) Attribute stereotype</td>
<td>Media</td>
<td>Alta</td>
<td>Alta</td>
<td>Media</td>
<td>Media</td>
</tr>
<tr>
<td>(c) Tagged values</td>
<td>Media</td>
<td>Alta</td>
<td>Media</td>
<td>Alta</td>
<td>Baja</td>
</tr>
<tr>
<td>(d) Notes</td>
<td>Media</td>
<td>Baja</td>
<td>Alta</td>
<td>Baja</td>
<td>Baja</td>
</tr>
<tr>
<td>(e) Class invariant</td>
<td>Baja</td>
<td>Alta</td>
<td>Alta</td>
<td>Baja</td>
<td>Media</td>
</tr>
<tr>
<td>(f) Attribute naming</td>
<td>Media</td>
<td>Baja</td>
<td>Alta</td>
<td>Baja</td>
<td>Baja</td>
</tr>
<tr>
<td>(g) Class invariant with</td>
<td>Baja</td>
<td>Alta</td>
<td>Baja</td>
<td>Altas</td>
<td>Media</td>
</tr>
<tr>
<td>naming convention</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tras este análisis finalmente hemos escogido la estrategia Attribute Stereotype porque nos ofrece un buen balance en los cinco criterios evaluados.

### 5.2. Precondiciones y poscondiciones

Las operaciones que aparecen en el diagrama de clases están descritas por medio de un contrato con restricciones de tipo precondición y poscondición (en este artículo asumimos que los resultados se expresan mediante una poscondición del estilo “post: result = ...”).

Dada la diversidad de tipos de precondiciones y poscondiciones, para los objetivos del proyecto AR3L nos centramos en un subconjunto suficientemente representativo para la experimentación. Este grupo de restricciones nos permiten analizar cuál es la mejor alternativa en este caso. Las condiciones soportadas (ver ejemplos en la tabla 1) son:
- **Poscondiciones:** Inserción y eliminación de elementos y obtener toda la población de una clase.
- **Precondiciones:** Comprobar la existencia de un elemento y comprobar que existe población.

Igual que en el caso del identificador, el problema principal es escoger qué estrategia usar para detectar las responsabilidades. Básicamente tenemos las mismas opciones que en el caso del identificador teniendo en cuenta que el elemento responsable en este caso es la operación. Otra diferencia importante es que en el caso del identificador teníamos sólo tres subtipos, pero en este caso tenemos muchos más, lo cual dificulta el uso de estereotipos o valores etiquetados. Por ello hemos decidido usar una nomenclatura. Los identificadores usados son: `insertElement`, `deleteElement`, `listAll`, `existsElement` y `notEmptyPopulation`.

5.3. **Cardinalidad de asociaciones**

En este caso el resto de cardinalidades que aparecen gráficamente podrían tratarse de la misma forma. Antes de continuar cabe remarcar que en nuestro caso sólo tenemos en cuenta las consecuencias de la cardinalidad propiamente sin considerar navegabilidades ya que consideramos que éstas no pertenecen al modelo de especificación.

Cada extremo de una asociación puede inducir una responsabilidad. Dependiendo de la cardinalidad podemos deducir que las cardinalidades "*" y "0" no inducen responsabilidad mientras que las numéricas añaden un límite superior o inferior (o ambos). (p.e., una cardinalidad de tipo "m..n" Induce una responsabilidad de mínimo m y máximo n instancias asociadas)

La cardinalidad de las asociaciones es un ejemplo típico de responsabilidad gráfica que no necesita elementos adicionales ni nomenclaturas, ya que el estándar UML nos ofrece suficiente información. Este tipo de responsabilidades se pueden tratar fácilmente desde AndroMDA y no requieren ningún esfuerzo extra por parte del especificador.

6. **Tratamientos**

En esta sección mostraremos el comportamiento de AR3L en distintos contextos de uso. Para ello necesitaremos definir algunos tratamientos y criterios no funcionales.

Los tratamientos pueden ser muy diversos. Distinguiendo dos tipos, tratamientos estáticos y tratamientos de procesos, los primeros normalmente aparecen en los esquemas de la base de datos, o en tecnologías emergentes como EJB mientras que los tratamientos de procesos son habituales de la capa de dominio aunque también puede aparecer, por ejemplo, en la capa de gestión de datos, por ejemplo, disparadores. En RTD, no prevemos limitación alguna en cuanto al tipo de tratamientos disponibles, pero en AR3L nos hemos focalizado en algunos tratamientos asociados a cada capa para obtener un grupo representativo de una arquitectura en 3 capas (ver tabla 3).

### Tabla 3: Lista de tratamientos

<table>
<thead>
<tr>
<th>Tratamiento</th>
<th>Capa</th>
<th>Descripción</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputSizeControl</td>
<td>x</td>
<td>Asegura que un conjunto no supera un número determinado de elementos (formulario)</td>
</tr>
<tr>
<td>InputFilter</td>
<td>x</td>
<td>Asegura que el elemento gestionado existe (combo box)</td>
</tr>
<tr>
<td>CardinalityControl</td>
<td>x</td>
<td>Asegura que el rol de una asociación tiene un número válido de elementos</td>
</tr>
<tr>
<td>Dictionary</td>
<td>x</td>
<td>Controla la población de una clase</td>
</tr>
<tr>
<td>Hibernate</td>
<td>x</td>
<td>Tecnología que provee persistencia de forma transparente (patrón Data Mapper)</td>
</tr>
<tr>
<td>SQL</td>
<td>x</td>
<td>Permite manipular directamente la base de datos</td>
</tr>
<tr>
<td>Trigger</td>
<td>x</td>
<td>Responde cuando se viola alguna restricción de integridad</td>
</tr>
<tr>
<td>DBSchema</td>
<td>x</td>
<td>Declara propiedades en las tablas (primary key)</td>
</tr>
<tr>
<td>OnCascade</td>
<td>x</td>
<td>Elimina elementos usando borrado en cascada</td>
</tr>
<tr>
<td>StoredProcedure</td>
<td>x</td>
<td>Permite añadir código en la base de datos</td>
</tr>
</tbody>
</table>

Los tratamientos se pueden aplicar a varias de las responsabilidades detectadas, por ejemplo `InputFilter` para la responsabilidad `ExistsElement`, u `OnCascade` para `DeleteElement`. Cada tratamiento tiene algún efecto sobre los criterios no funcionales, por ejemplo eficiencia, complejidad, etc. Actualmente AR3L asigna los tratamientos dependiendo de los criterios no funcionales, y no es posible aplicar tratamientos distintos a responsabilidades del mismo tipo.

En esta sección tomamos los siguientes criterios no funcionales para la selección de tratamientos:
- **Dependencia tecnológica** (alta, media, baja).
- **Lenguaje de desarrollo** (Java, C++, .NET).
- **Database** (ninguna, Oracle, PostgreSQL).
- **Complejidad de la interficie** (alta, media, baja).
Para establecer la relación entre los criterios no funcionales, los tratamientos y las responsabilidades hemos creado un conjunto de tablas (una por cada tipo de responsabilidad). Uno de los tratamientos de cada tabla será asignado por defecto, lo que nos permite asegurar que todas las responsabilidades tienen como mínimo un tratamiento asignado, que no requieren asumir criterios no funcionales sobre el sistema resultante. La tabla 4 es un ejemplo para el tipo de responsabilidad de cardinalidad: el tratamiento en cursiva es el tratamiento por defecto, las casillas sombreadas son aquéllas que no tienen efecto en el proceso de selección, las "X" impiden el tratamiento y los "√" lo permiten. Los contenidos de la tabla reflejan algunas heurísticas (p.e., un tratamiento que requiere Database padece al menos una dependencia tecnológica media).

Tabla 4: Tabla de tratamientos para la responsabilidad de cardinalidad

<table>
<thead>
<tr>
<th>Dependencias tecnológicas</th>
<th>Lenguaje de desarrollo</th>
<th>Complejidad de la interficie</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Como ejemplos de uso, consideremos los siguientes (basados en el escenario de uso expuesto en la sección 3):

- **Ejemplo 1**: Estamos interesados en diseñar un subsistema que obtiene la información de los artículos y automáticamente genera el programa de la conferencia (p.e., la distribución de los artículos en sesiones). No necesitamos base de datos ya que los datos se cargan en memoria. La complejidad de la interficie es mínima ya que no pretendemos interacción con el usuario. Queremos usar el lenguaje C++ y la dependencia tecnología no es relevante.

- **Ejemplo 2**: Estamos interesados en añadir una interfaz gráfica al subsistema que permita introducir artículos manualmente. Ya que esta interfaz funcionara bajo el sistema operativo Windows, hemos decidido cambiar el lenguaje a .NET.

- **Ejemplo 3**: Queremos hacer el subsistema persistente, añadiendo una base de datos que mantenga la información de las sesiones y que permita futuras modificaciones.

La tabla 5 muestra un resumen de los criterios no funcionales seleccionados en cada caso.

Tabla 5: Resumen de requisitos no funcionales en los ejemplos vistos

<table>
<thead>
<tr>
<th>Ejemplo</th>
<th>Media</th>
<th>C++</th>
<th>Ninguna</th>
<th>Baja</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ejemplo 1</td>
<td>Media</td>
<td>C++</td>
<td>Ninguna</td>
<td>Baja</td>
</tr>
<tr>
<td>Ejemplo 2</td>
<td>Alta</td>
<td>.NET</td>
<td>Oracle</td>
<td>Baja</td>
</tr>
<tr>
<td>Ejemplo 3</td>
<td>Alta</td>
<td>.NET</td>
<td>Oracle</td>
<td>Alta</td>
</tr>
</tbody>
</table>

La tabla 6 muestra el resultado obtenido de ejecutar AR3L en los distintos contextos descritos. Como se puede ver, en el primer ejemplo AR3L selecciona sólo los tratamientos por defecto para todas las responsabilidades. En el ejemplo 2, se añaden todos los tratamientos de interficie y finalmente en el ejemplo 3, se añaden los tratamientos que requieren una base de datos. Nótese que con otro conjunto de tratamientos podría darse el caso en el que con los mismos ejemplos obtuviéramos un resultado con más tratamientos disponibles para el primer ejemplo y menos para el último, al contrario del resultado expuesto en la tabla 6.

Tabla 6: Resultados de AR3L en los ejemplos expuestos

<table>
<thead>
<tr>
<th>Trementimentos</th>
<th>Responsabilidades</th>
<th>Identificador de clase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ejemplo 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentación</td>
<td>Dominio</td>
<td>Dictionary</td>
</tr>
</tbody>
</table>

| Ejemplo 2 | | |
| Presentación | | |

| Ejemplo 3 | | |
| Presentación | | |

La tabla 6 muestra el resultado obtenido de ejecutar AR3L en los distintos contextos descritos. Como se puede ver, en el primer ejemplo AR3L selecciona sólo los tratamientos por defecto para todas las responsabilidades. En el ejemplo 2, se añaden todos los tratamientos de interficie y finalmente en el ejemplo 3, se añaden los tratamientos que requieren una base de datos. Nótese que con otro conjunto de tratamientos podría darse el caso en el que con los mismos ejemplos obtuviéramos un resultado con más tratamientos disponibles para el primer ejemplo y menos para el último, al contrario del resultado expuesto en la tabla 6.
7. Implementación

Como ya hemos comentado, la implementación de AR3L está basada en una plataforma MDD, AndroMDA. Ésta usa unos componentes llamados cartuchos para extender sus funcionalidades. En nuestro cartucho, siguiendo las directrices de desarrollo de esta plataforma, hemos extendido el metamodelo de AndroMDA para facilitar la detección de responsabilidades (ver figura 6) y también hemos definido los estereotipos para la detección de identificadores (ver figura 7).

La decisión de extender el metamodelo para definir los estereotipos, como se ve en la figura 7, fue por motivos prácticos. De esta forma AndroMDA nos permite incrustar las restricciones sobre los modelos de entrada en forma de restricciones OCL dentro del propio metamodelo.

Para mejorar la mantenibilidad y extensibilidad del código hemos seguido un criterio de desarrollo: las responsabilidades se detectan desde el elemento responsable, que a su vez es quien almacena sus responsabilidades. Como se puede ver en la figura 6, todas las responsabilidades que pertenecen a una clase quedan almacenadas dentro de su extensión del metamodelo.

En algunos casos no parece la mejor opción determinar las responsabilidades desde el elemento responsable, como en el caso de la responsabilidad gráfica de cardinalidad de asociaciones ya que sería más fácil obtener todas las asociaciones del modelo y ver cuál genera una responsabilidad, en cambio lo que hacemos es tomar todas las clases del modelo (ya que este es el elemento responsable), obtener las terminaciones opuestas de cada una de sus asociaciones y ver si la terminación genera una responsabilidad para dicha clase.

El código fuente de la herramienta AR3L y algunos ejemplos de uso están disponibles en: http://www.lsi.upc.edu/~gessi/AR3L

8. Conclusiones y trabajo futuro

En este artículo hemos presentado las ideas genéricas de RDT y el estado actual del proyecto AR3L, un sistema para detectar responsabilidades a partir de la descripción del comportamiento de un sistema y seleccionar tratamientos para éstas acordes con los requisitos no funcionales del sistema. Como resultado, se dispone de una descripción de las tres capas que forman la arquitectura del sistema, a partir de la cual puede aplicarse algún generador de código.

Consideramos que las contribuciones más importantes de RDT son:

- Proponemos un marco de trabajo altamente genérico que permite definir diversos tipos de artefactos de entrada, formatos de generación de salida, y catálogos de responsabilidades y tratamientos. Como resultado, y creemos que es la mayor contribución, el marco de trabajo puede adaptarse a una gran variedad de procesos de desarrollo de software (más rigidos, más ágiles, etc.) y a diversas opciones tecnológicas subyacentes.

- Desacoplamos el diseño arquitectónico propiamente dicho con la generación de código. De esta manera, el diseñador puede experimentar diversas alternativas arquitectónicas y una vez tomada una
decisión, generar código con su herramienta MDD favorita.

- No nos limitamos a una tecnología en concreto (Hibernate, EJBs, etc.), como contrapartida la generación de código es más compleja en esta situación.
- En la implementación actual, AR3L, usamos una plataforma consolidada (AndroMDA) lo cual nos permite sacar partido de los progresos en dicha plataforma.

Esta propuesta puede ser analizada desde distintos puntos de vista:

- Desde el punto de vista del especificador, éste debe incluir más información para obtener buenos resultados con el uso de esta herramienta. Esta cantidad de trabajo extra es relativamente poca y puede ayudar a que la especificación sea más completa.
- Desde el punto de vista del diseñador, éste puede gestionar su catálogo de tratamientos y de criterios no funcionales para estudiar los efectos de sus decisiones, podría interactuar en cada paso refinando los resultados. El diseñador reduciría su volumen de trabajo y además al ser un proceso altamente automatizado se podría cambiar la especificación sin que esto repercutiera muy negativamente en el tiempo de desarrollo del sistema.
- Desde el punto de vista de un desarrollador, éste tiene a su disposición una arquitectura altamente extensible (p.e., posibilidad de añadir nuevas responsabilidades al proceso de detección), y en la versión actual, mantenible (gracias a la arquitectura de AndroMDA es fácil cambiar un componente por otro).

La dedicación futura a este proyecto se centrará en completar la implementación del RDT Framework (figura 2) y a analizar y solucionar los problemas que no se hayan detectado hasta el momento. Esto incluye la creación de las herramientas complementarias, el soporte de más artefactos de entrada (por ejemplo, en los modelos UML, incluir diagramas de secuencia que declaren las operaciones del sistema), y la generación de diversos tipos de modelos (inicialmente nos centraremos modelos UML por ser estos los más usados) para posibilitar la generación de código automáticamente. Igualmente la definición de requisitos no funcionales y su evaluación jugará un papel importante en el futuro del sistema.

Agradecimientos
Este trabajo se ha desarrollado en el marco del proyecto TIN2004-07461-C02-01.

Referencias
Combining Model-Driven Development and Architectural Design in the AR3L Framework

David Ameller, Xavier Franch
Universitat Politècnica de Catalunya
{dameller, franch}@lsi.upc.edu

Abstract. Model-Driven Development (MDD) has the ultimate goal of producing code from some kind of analysis model, aiming at maximizing the automation of software development. However, it is difficult to inject in the MDD transformation process all the knowledge, skills and experiences that software architects have and which may be crucial when considering the satisfaction of system properties and constraints over the system-to-be. In this paper we propose the AR3L framework that integrates the software architect role into the MDD transformation process by recognizing the need of human interaction in an intermediate design phase that produces an architecture model. We focus on a particular domain, namely the development of Information Systems which, considering the current state of practice, determines the type of analysis model (object-oriented specification) and the target architecture pattern (3-layer architecture). We use responsibility-driven design as methodology driver. We present a proof-of-concept prototype implemented over the AndroMDA tool that shows the feasibility of our approach.

1 Introduction

Model-Driven Development (MDD) is a software development paradigm that has gained acceptance in the last years. According to [1], “Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing”. Usually, the “real thing” is understood as the code, therefore most MDD approaches aim at producing some executable code as final result of a transformation process from some starting model of the system.

This development paradigm partly collides with the vision that software architects have about software system design. Apart from the functionalities described in the analysis model, software architects are concerned with the analysis of fundamental system properties (like performance, reliability, ease-of-use, etc.) and the satisfaction of constraints (referring for instance to the use of particular databases or programming languages, or licensing issues) in order to choose the most adequate technologies to implement the functionalities of the system. Software architects want to play with some architectural model, understanding the consequences of an architecture over those properties, and exploring what-if questions (what if this response time requirement is relaxed? what if the programming language changes from Java to C#?), eventually giving feedback to analysts about feasibility of the analysis model.
They are not really interested in code but in more abstract questions, e.g. whether to use some enterprise pattern or other (e.g., Data Mapper [2] vs. Transaction Script [2]), which product better satisfies the needs implied by a pattern in the system (e.g., Hibernate\(^1\) vs. .NET\(^2\) DataMapper), to what extent database-related features (e.g., stored procedures, triggers, etc.) are preferable to writing code in some programming language, etc. Once an architecture model has been built, gene-rating code is a subordinated (although of course still very complex) task to perform.

The natural answer to this conflict is to explicitly distinguish the architecture model and the associated architecture design phase in the MDD transformation process. Therefore, we may reconcile both of the worlds: still we do have the advantages of MDD, whilst providing the possibility to software architects to take informed decisions according to system properties and constraints, building an architecture model from which code is derived in the rest of the MDD transformation process.

The goal of this paper is to present such a framework in the particular case of development of Information Systems (IS), which are one of the most typical types of software application nowadays (in different forms: client/server applications, Web-based systems, etc.). This type of systems are currently specified mostly using an object-oriented (OO) analysis methodology [3], and usually adopt a 3-layer architecture pattern [4], being the layers: presentation layer; domain, also named business, layer; and data, also named persistence, layer. In this scenario, if we refer again to the definition given in [1], we may say that:

- **Model of a system.** It is an OO analysis model that describes the functionality of the IS (i.e., what the system does) and a set of properties (e.g., efficiency, security, ...) and constraints (e.g., which database to use).
- **Real thing:** It is an architecture model generated from the analysis model, which basically declares which available technologies are used to structure the system and tackle the points behind the analysis model.
- **Transformation.** It is an assignment of elements from the functional part to the architecture, which satisfies the stated properties and constraints.

In this context, we have formulated the AR3L (Assignment of Responsibilities into a 3-Layered architecture model) framework. In AR3L, we take as functional specification a UML model [5] (although in fact other OO options like OMT [6] would be possible) and we describe both properties and constraints as a list of requirements in the form of restrictions stated over measurable factors. Concerning the architecture model, we focus on technology features (e.g., declaration of primary key in a database schema, use of a combo box in a GUI, use of a dictionary data structure in the domain layer, etc.) and how they solve the functionality of the system. We consider this assignment, from functionality to technology features, as a simplified architecture model. In other words, the goal of this paper is to present a MDD framework that generates, from an OO analysis model, a 3-layer architecture model that satisfies a list of requirements about system properties and additional constraints.

Therefore, the transformation is a process that takes the following form (see fig. 1):

\(^1\) http://www.hibernate.org
\(^2\) http://msdn.microsoft.com/netframework
1. Infer the atomic elements that describe the system’s functionality. Following the terminology introduced by [7], we call them responsibilities.
2. For each responsibility, select some technology feature that supports the stated requirements. These technology features are called treatments in the paper.
3. As a third step that is out of our current work, the transformation process would end by generating the source code from the architecture model.

In the first two steps, the architect constantly receives information from the transformation process and provides feedback, makes decisions, experiments, etc.

AR3L is a tooled framework. Among the several platforms that provide support for MDD (see [8, 9] for a survey), we have chosen AndroMDA3, an open source MDA [10] platform because it is highly customizable and it seems to be stable (in the sense of reliability and likelihood of existing over some time). As it is the usual case in this type of tools, AndroMDA is normally used for code generation but due to its customizability, we have been able to use it to create the architecture model as described above. With AndroMDA we have built a proof of concept of the AR3L framework that is also described in the paper.

The rest of the paper is organized as follows. In sections 2 and 3, we introduce the two processes that take part in the AR3L framework, responsibility identification and treatment assignment. In section 4, we give some details about the AndroMDA-based implementation. Last in section 5 we give the conclusions of our work.

Fig. 1. The AR3L framework.

2 AR3L: Responsibility Identification

In the context of OO analysis and design, the concept of responsibility was popularized by Wirfs-Brock et al. [7]. A precise definition of responsibility is given in [3]: “a responsibility embodies one or more of the purposes or obligations of an element”. Responsibilities are identified in the analysis model.

3 http://www.andromda.org
In our work, the analysis model has been declared to be a UML model. Therefore, the “elements” referred in the definition above are classes, attributes, operations, etc., and the responsibilities are obligations implied by these elements, e.g. being singleton (class), not exceeding some given value (attribute), satisfying some given precondition (operation), etc. More precisely, we have constructed a built-in taxonomy with almost 150 types of responsibilities, structured in a hierarchy of 6 levels. For instance, in the first level we have the categories: Class, Class Population, Attribute, Operation, Association, Association Population and Inheritance. Then, Operation is decomposed into Parameter, Precondition and Postcondition, etc.

Depending on the way they are declared, we may classify responsibilities into two main categories which hardly determine the way they will be identified in the model:

- **Graphical responsibilities.** They can be inferred from UML graphical elements, e.g. cardinality (of associations, attributes, etc.), properties (readonly, complete/disjoint, subset, etc.), dependencies (create, etc.) and by the like.

- **Non-graphical responsibilities.** They do not appear in the UML model, instead they are declared textually as integrity constraints expressed in natural language or more formal notations as the OCL. They may have a name. We distinguish two main subcategories:
  - **Permanent responsibilities.** They must be fulfilled in any valid state of the system, e.g. declaration of class identifiers or restrictions on values of class attributes. They are represented by means of class invariants.
  - **Event-driven responsibilities.** They must be satisfied when an operation is invoked. There are two types, preconditions and postconditions.

In the proof-of-concept we are describing in this paper, we have focused on one type of responsibility of each category and subcategory presented above, which we think are representative enough of the taxonomy. In particular, as graphical responsibility, we will use association cardinality; for permanent non-graphical responsibilities, declaration of class identifiers; for event-driven non-graphical responsibilities, a selected group of pre and postconditions (enumerated in section 2.3). We present these types in the rest of the section and we illustrate them using an example about a Conference Management System (see fig. 2). Table 1 presents the summary of all the responsibilities of those types found in the example.

![Fig. 2. Conceptual data model for the Conference Management System.](image-url)
Table 1. List of responsibilities of the Conference Management System.

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Model Element Name</th>
<th>Responsibility description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality</td>
<td>Paper</td>
<td>An object 'Paper' may have at most 1 object 'Congress' bound.</td>
</tr>
<tr>
<td></td>
<td>SentPaper</td>
<td>An object 'SentPaper' must have exactly 1 object 'Paper' bound.</td>
</tr>
<tr>
<td></td>
<td>Session</td>
<td>An object 'Session' must have exactly 1 object 'Congress' bound.</td>
</tr>
<tr>
<td></td>
<td>Accepted</td>
<td>An object 'Accepted' must have exactly 1 object 'Session' bound.</td>
</tr>
<tr>
<td>Identifier</td>
<td>Paper</td>
<td>Title is identifier.</td>
</tr>
<tr>
<td></td>
<td>Congress</td>
<td>Name and acronym are both identifiers.</td>
</tr>
<tr>
<td></td>
<td>Date</td>
<td>Date is identifier.</td>
</tr>
<tr>
<td></td>
<td>Session</td>
<td>There cannot be two sessions with the same name in the same congress.</td>
</tr>
<tr>
<td>Pre / post</td>
<td>Congress.showSessions</td>
<td>insertElement: post: Create a SentPaper.</td>
</tr>
<tr>
<td></td>
<td>Accepted.listAll</td>
<td>deleteElement: post: Delete all the Accepted Papers.</td>
</tr>
</tbody>
</table>

2.1 Association Cardinality Responsibility

Each association in the analysis model declares cardinality at each role. Given an association between A and B, the cardinality at one of its roles, let's say role at B, may define 0, 1 or 2 responsibilities depending on the form of that cardinality:

- Cardinality "**" means that there are not cardinality responsibilities induced.
- Cardinality of the form "0..n" induces a cardinality responsibility of upper bound (an instance of A may not be bound to more than n instances of B), whilst a cardinality of the form “n..*" induces a cardinality responsibility of lower bound (an instance of A may not be bound to more than n instances of B).
- Cardinality of the form "m..n" induces cardinality responsibilities of upper bound and lower bound.
- Cardinalities of the form "n" induce a cardinality responsibility of exact number (an instance of A must be bound to exactly n instances of B).

2.2 Identifier Responsibility

Most of the classes in the analysis model have a unique identifier (in addition to the OID) which allows distinguishing two different instances of a class. Identifier declaration is one of the most, if not the most, frequent types of integrity constraints that appears in specification models.

More precisely, a class C may have three types of identifiers:

- **Key.** The identifier is composed of one or more attributes declared in C.
- **Weak key.** The identifier is composed of one or more attributes declared in C together with one or more attributes that are identifiers of a class D such that a composition (in the sense of the UML) exists from C to D.
- **Alternative key.** In addition to a key, the class C may have some other attribute or set of attributes that uniquely identify C’s instances.
It is worth to remark that the three types of identifiers must satisfy several restrictions that are listed below:

- **R1** A class may have key or weak key, but not both at the same time.
- **R2** A class may not have more than one key or more than one weak key.
- **R3** For a class, the existence of alternative key demands the existence of a key.
- **R4** A class with weak key cannot be part of a composition of more than one key.
- **R5** For a class C with weak key such that D is the class linked by composition with C, it is required that D has either key or weak key.
- **R6** For a sequence of classes C1...Ck, such that a composition exists from Cj to Cj+1, 1 ≤ j ≤ k-1, and each Cj has weak key, then Ck must have key.

In the example of fig. 2, the class Congress has both: a key, name, and an alternative key, acronym (note that deciding which identifier is key and which one alternative key is up to the analyst). On the other hand, the class Session has a weak key, composed by its attribute name and the key of the class Congress because Congress is a composition of Sessions, which means that two different sessions belonging to the same Congress may not have the same name.

### 2.3 Pre/Post Responsibilities

The behavior of the operations that appear in sequence diagrams is defined by means of a contract. Following Meyer’s design by contract [11], we distinguish among preconditions and postconditions:

- **Precondition.** Condition that must be satisfied when executing an operation in order to get the agreed results. Typically this involves checking the value of one or more parameters and/or the state of the system.

- **Postcondition.** Describe an action that is performed during the execution of an operation. They may indicate a change in the state of the system or how a value is computed.

Due to the diversity of types of preconditions and postconditions, for the purposes of the proof-of-concept prototype we focus on a few types that are quite representative and allow us analyzing the actions to take in our approach. They are: insertion and deletion of elements (postconditions), get all the instances of a class (postcondition), check existence of an element (precondition), and check that a class has at least one instance (precondition). In Table 1, some examples appear covering all these cases for some operations introduced in the model: send a paper, show all the sessions of a conference, delete a paper sent and list all accepted papers.

### 3 AR3L: Treatment Assignment

Responsibility-driven design [7] may be seen as the process of assigning treatments (i.e., available technological features) to the responsibilities identified in the previous step. This view has been adopted more or less explicitly by the most widely used OO methodologies nowadays, both comprehensive (e.g., UP [12]) or more focused (e.g., Larman’s [3], Fowler’s [2], etc.).
3.1 Responsibilities, treatments and properties

In OO IS 3-layered design, a treatment may imply one or more layers. Component technologies such as Hibernate or EJBs\(^4\), programming paradigms such as aspect-orientation, programming languages such as Java, graphical user interfaces or standards such as Swing\(^5\) or XML, etc., determine the set of treatments available. Some examples on each layer are:

- **InputFilter** (presentation layer): ensures that an element exists using some GUI feature (e.g., a combo box).
- **Dictionary** (domain layer): in-memory structure that keeps track of the instances of a class.
- **OnCascade** (data layer): deletes elements using the OnCascade feature.

Fig. 3 presents a reference model of the AR3L framework. A treatment covers some of the responsibilities defined in our repository, e.g. InputFilter for the ExistsElement responsibility, or OnCascade for DeleteElement. Furthermore, each treatment has some effects on system properties and constraints, collectively known as requirements, e.g. Dictionary supports portability, whilst OnCascade damages it. The current responsibility assignment policy in AR3L binds treatments to responsibilities according to the values required on these requirements. To help the architect taking decisions, a treatment may have additional information bound (e.g., reports on technology, benchmarks, etc.) which can be consulted or updated during the responsibility assignment process. Technologies like Hibernate, Oracle\(^6\), Swing, etc., implement treatments, therefore Oracle may implement OnCascade whilst Swing implements InputFilter. This information is important because of course the architect will generally not select two technologies of similar characteristics (e.g., two GUI libraries) to cover related responsibilities; also, it may be used as a guideline for the subsequent code generation phase. Some existing integrity constraints (e.g., coherence among responsibility and treatment hierarchies with respect to the is-covered-by association) complement this model.

![Reference model of the AR3L framework](image)

---


\(^5\) [http://java.sun.com/javase/6/docs/technotes/guides/swing](http://java.sun.com/javase/6/docs/technotes/guides/swing)

\(^6\) [http://www.oracle.com](http://www.oracle.com)
3.2 An Example: Application to the Conference Manager System

Next we provide scenarios that show the use of AR3L in different contexts. We present the scenarios and extract the information relevant to the responsibility assignment problem in the form of properties and constraints representing those scenarios. To do so, we need first to determine some treatments, properties and constraints to run the simulation.

For obtaining a representative sample in the 3-layer IS architecture context, we enumerate below some treatments bound to the different layers that will be considered in our Conference Management System example:

- **InputSizeControl** (presentation layer): ensures that a set of values (obtained via e.g. a combo box) does not exceed a given number of elements.
- **InputFilter** (presentation layer): ensures that an element exists using some GUI feature (e.g., a combo box).
- **CardinalityControl** (domain layer): piece of code that ensures that the role of an association does not exceed a given number of elements.
- **Dictionary** (domain layer): in-memory structure that keeps track of the instances of a class.
- **DataMapper** (domain and data management layers): provides isolation between the information in the database and memory (applying the Data Mapper pattern [2]).
- **SQL** (data management layer): allows direct manipulation of the database.
- **Trigger** (data management layer): reacts when some condition is violated.
- **DBSchema** (data management layer): declares properties of tables.
- **StoredProcedure** (data management layer): code at the database.

Concerning properties and constraints we have chosen the following:

- **Portability**. To what extent an application is portable from one platform to others (Low, Medium, High).
- **Operability**. To what extent an application is easy to use. We will focus on one factor that heavily affects this criterion, complexity of user interface (Low, Medium, High).
- **Environmental constraints**. Which constraints exist about development that cannot be negotiated. As an example, we mention the database (none, Oracle, PostgreSQL\(^7\)).
- **Development constraints**. Which constraints exist about development that cannot be negotiated. As an example, we mention the development language (C++, Java, .NET).

Table 2 analyses the effect on these properties and constraints of the possible treatments for the association cardinality responsibility. We state for each treatment which are the values of properties and constraints that allow their application, for instance, to apply the InputSizeControl, there should not exist a requirement stating that Interface Complexity shall be Low. Empty cells mean that the treatment does not affect the property or constraint (e.g., using SQL does not affect Interface Complexity). The contents of the table reflect some heuristics (e.g., a treatment that

\(^7\) http://www.postgresql.org
requires a data base has at least Medium technological dependency). Tables for the other responsibilities (not shown here for lack of space) are coherent.

**Table 2.** Influence of some treatments for association cardinality on system properties and constraints

<table>
<thead>
<tr>
<th>Treatments for cardinality responsibility</th>
<th>Portability</th>
<th>Development language</th>
<th>Database</th>
<th>Interface complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High</td>
<td>Low</td>
<td>C++</td>
<td>Java</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>InputSizeControl</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>DataMapper</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>CardinalityControl</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>SQL</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>x</td>
<td>x</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>DBSchema</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

We consider next three possible scenarios around the Conference Management System that will be under control of the software architect:

**Scenario 1:** we are interested in designing a subsystem that, given some keywords extracted from the title, obtains information about the papers and automatically generates the distribution of the papers into sessions. For modifiability and reusability purposes, this subsystem is designed as a service, with interface `assignPapers(allTitles: Set(String)): Set(nameSess: String + papers: Set(String))`.

The analyst concludes that: (1) the service does not need data base, it just performs some on-memory calculations; (2) interface complexity is Low, since both input (non-existing) and output (just a text file) are simple; (3) portability is not a big issue; (4) C++ seems to be the preferred language once checked the environment where the service will be deployed. Table 3, row 1, shows the result of this analysis. The architect receives the first feedback as shown in Table 4, Scenario 1 column. As a summary, all kind of responsibilities are assigned to the domain layer. The architect considers adequate the result and then confirms the proposal. Code production phase may start.

**Scenario 2:** after some time on production, the analyst is reported that the session assignment service behaves quite satisfactorily but not perfectly. Program chairs using the system require the ability to change manually the first assignment proposed by the system. Therefore, the analyst decides to modify the functionality of the system to allow changing paper assignment manually. As a consequence, the Complexity of User Interface is changed from Low to Medium (see table 3, row 2), to allow assignment of not much complex treatments that imply the presentation layer. Once the architect runs AR3L, he/she discovers some additional information bound to the treatment stating that presentation layer treatments are more difficult to implement using C++ than .NET. Since this decision was not a primary issue, he/she changes from C++ to .NET. Now, the assignment of treatments proposes presentation layer technologies as an alternative to domain layer for some responsibilities (see table 4,
Scenario 2 column). The architect decides to exploit this possibility and informs AR3L about his/her choice to produce the final architecture model.

Scenario 3: again the intensive use of the service reveals that program chairs are still not happy, because not all the problems with session assignment are known in advance: authors may give preference about their travel dates later on, papers may be removed for several reasons, etc. Once informed, the architect decides that the solution is making the information about sessions persistent so it may be changed at any time. As an obvious option, he/she states as constraint that the data base shall be the same than the rest of the system, let’s say Oracle. Once AR3L runs, the architect is informed that data layer treatments may not be exploited adequately unless portability is sacrificed. After some trade-off analysis, the architect decides to sacrifice portability (see table 3, row 3). As a result, under the agreement with the analyst, the architect gets a great deal of treatments applicable (see table 4, Scenario 3 column). Since DataMapper is a valid option for all responsibilities, it is finally chosen, and a technology implementing this treatment (e.g., Hibernate) may be selected.

In these scenarios we have followed a responsibility-driven treatment assignment process. However, other strategies are possible. For instance, architects usually feel more comfortable with some technologies that they know well, and may follow then a technology-driven treatment assignment process. For instance, Hibernate is a technology which currently has lot of success among IS developers. A possible strategy would be then to select Hibernate as starting point and then analyse which are the effects of this selection. If we assume this situation in Scenario 3, we may see that most of the responsibilities have then a default assignment to the DataMapper treatment (i.e., the treatment that is implemented by Hibernate). The architect would then confirm or not this treatment, perhaps changing to presentation layer treatments (i.e., InputFilter and InputSizeControl) when possible to avoid errors in lower layers.

Table 3. Mapping scenarios onto system properties and constraints

<table>
<thead>
<tr>
<th></th>
<th>Portability</th>
<th>Development language</th>
<th>Data Base</th>
<th>Interface complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 1</td>
<td>Medium</td>
<td>C++</td>
<td>None</td>
<td>Low</td>
</tr>
<tr>
<td>Scenario 2</td>
<td>Medium</td>
<td>.NET</td>
<td>None</td>
<td>Medium</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>Low</td>
<td>.NET</td>
<td>Oracle</td>
<td>Medium</td>
</tr>
</tbody>
</table>

4 AR3L: An AndroMDA-based Tool

In this section we briefly describe the architecture of AR3L, which is built upon AndroMDA version 3. For space reasons, it is not possible to give the details of the solution, so we basically provide an overview being aware that full understanding would require more thorough explanations. We will focus to the most relevant aspect of using AndroMDA for our work, namely determining the information that must be included in the analysis model to facilitate responsibility identification.
**Table 4. Assignment of treatments to responsibilities in the three scenarios**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Scenario 1</th>
<th>Scenario 2</th>
<th>Scenario 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>Dictionary</td>
<td>Dictionary</td>
<td>Dictionary</td>
</tr>
<tr>
<td>Cardinality</td>
<td>CardinalityControl</td>
<td>CardinalityControl</td>
<td>InputSizeControl</td>
</tr>
<tr>
<td></td>
<td>SQL</td>
<td>SQL</td>
<td>SQL</td>
</tr>
<tr>
<td></td>
<td>DBSchema</td>
<td>DBSchema</td>
<td>DBSchema</td>
</tr>
<tr>
<td></td>
<td>DataMapper</td>
<td>DataMapper</td>
<td>DataMapper</td>
</tr>
<tr>
<td>existsElement</td>
<td>Dictionary</td>
<td>Dictionary</td>
<td>Dictionary</td>
</tr>
<tr>
<td></td>
<td>InputFilter</td>
<td>SQL</td>
<td>DataMapper</td>
</tr>
<tr>
<td>insertElement</td>
<td>Dictionary</td>
<td>Dictionary</td>
<td>Dictionary</td>
</tr>
<tr>
<td></td>
<td>SQL</td>
<td>StoredProcedure</td>
<td>DataMapper</td>
</tr>
<tr>
<td>deleteElement</td>
<td>Dictionary</td>
<td>Dictionary</td>
<td>Dictionary</td>
</tr>
<tr>
<td></td>
<td>SQL</td>
<td>StoredProcedure</td>
<td>OnCascade</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DataMapper</td>
</tr>
<tr>
<td></td>
<td>SQL</td>
<td>DataMapper</td>
<td>DataMapper</td>
</tr>
<tr>
<td>listAll</td>
<td>Dictionary</td>
<td>Dictionary</td>
<td>Dictionary</td>
</tr>
<tr>
<td></td>
<td>SQL</td>
<td>DataMapper</td>
<td>DataMapper</td>
</tr>
</tbody>
</table>

### 4.1 AndroMDA

AndroMDA is an open source MDA framework that takes models of different types (usually UML models stored in XMI produced from case-tools) and processes them for obtaining some result of a given type, using a kind of plug-ins named cartridges. These cartridges implement M2T (model to text) transformations.

The AndroMDA components are managed by the AndroMDA core engine, which takes care of making all components of AndroMDA work together:

- The cartridges provide the ability to process UML standard models for a specific technology using template files.
- Template engines are components that generate the output of the process. In AndroMDA the most commonly used template engine is Velocity.
• AndroMDA Metafacades are facades that wrap the underlying metamodel implementation in such a way that AndroMDA designers are unaware of the details of that metamodel. Metamodels are MOF models such as UML 2.0, etc.

• Repositories allow switching out the repository that performs reading/loading of a MOF model. By default, AndroMDA reads models from XMI files.

• Translation libraries are used to translate OCL expressions into other languages. For example, a translation library could be used to translate an OCL body expression into a Hibernate-QL, EJB-QL, OCL, Java, SQL, etc.

4.2 AR3L Tool: General View

As presented in the previous sections, general architecture of AR3L is divided into two parts, responsibility identification and treatment assignment. The first component uses the AndroMDA platform to interpret models and it has been implemented with a cartridge (the ARC cartridge). The technology assignment component has been implemented using standard Java and thus it could be eventually plugged into another different system.

To run this architecture, the analyst has to provide a UML model, the system properties and constraints, which are managed by the AndroMDA project system. The cartridge has access to this analysis model. In the process of detection, the cartridge builds as result a Data Transfer Object [2] packing all the detected responsibilities that is used in the next step. The assignment of treatments is done by a Velocity template that processes the responsibilities obtained from AndroMDA Core. The template calls to the treatment assignment module to assign a set of applicable technologies to each responsibility. In each call the template engine sends the user project information needed by the module. The result is a mapping from responsibilities to treatments.

Concerning the internal design of AR3L, there are two types of metafacades used in this project, one for responsibility management and another for stereotypes facilities. The metafacades used for managing and storing responsibilities also include the responsibility identification node, they extend the generic metafacades offered by AndroMDA. Each metafacade of this kind offers a collection of responsibilities and a method to access them easily (see fig. 4, operations facade). The second type of metafacades are just for convenience, they let us set the stereotypes related to keys and also to establish metamodel constraints (see fig. 4, attribute facade).

Fig. 2. Use of metafacades in AR3L.
4.3 AR3L Tool: Detailed View

In this section we describe in more detail how the three types of responsibilities detected in this paper are implemented in AndroMDA. According the internal architecture of AR3L, there are two issues to tackle in each type: how the responsibility is detected and how this detection is implemented.

4.3.1 Association cardinality responsibility

Association cardinalities are a typical example of graphical responsibility in which its identification is straightforward from the standard UML model, without needing additional elements, naming conventions or like.

Therefore, to implement the identification of this type of responsibility, we extend an AndroMDA metafacade; this extension will take care of the detection. At this point we face a situation that is recurrent in AndroMDA-based solutions, namely which metafacade is the most appropriate. In fact, each type of responsibility may be detected from several of the UML elements metafacade. Therefore, we need to establish a clear decision criterion in order to improve the understandability of the solution. Applying the same responsibility-driven design principle that we are advocating in the paper, we determine as general criterion to assign the type of responsibility to the metafacade of the UML element that is responsible of that type. This option is preferred in front of detecting the responsibility directly in the metafacade that corresponds to the UML element more related to that type of responsibility because this could provoke some ambiguity. Note that as result of this decision, the resulting code is more structured and more extensible: having such a clear organizational criterion facilitates extension because it is very clear which part of the system (i.e., metafacade) will have to take care of the new responsibility.

In the cardinality case, we basically have two options, to extend the metafacade of the UML element corresponding to associations or the one for classes. Applying the criterion above, we choose the metafacade for classes, which is ClassifierFacade, since each class is responsible for the roles that stem from it. Note that in this case, the first option would have been easier to implement, because for each association that exists in the model there would have been an instance of the metafacade taking care of the responsibilities inferred from that association's roles, that are exactly two (whilst in the case of a class, the number of roles is variable). Anyway, the chosen strategy is not difficult to implement, it just requires the following steps:

1. Building a metafacade that extends the functionality of the generic Classifier Facade. We remark that if other types of responsibilities also are assigned to UML classifiers, this same extension will be used for them (as we show later).
2. Implementing the detection code inside this new metafacade. This code just analyses all the associations of the class and collects all the responsibilities generated by the cardinalities as specified in the beginning of this section.
3. Developing a Data Transfer Object (DTO [2]) to represent this responsibility. Using DTOs makes our solution less dependent on the technology.

4.3.2 Identifier Responsibility

The consideration of non-graphical responsibilities raises the fundamental problem of identifying these responsibilities in the specification model. Several strategies are possible: using UML elements such as notes, stereotypes and tagged values; using
naming conventions; etc.; in fact, several strategies could be eventually used for different types of responsibilities. To select in each case the detection strategy we have take into account the following values: analyst effort, understandability, expressiveness, standardization and solution implementation effort.

For the identifier case, we have selected attribute level stereotypes strategy. Each attribute that is part of an identifier is preceded by the appropriate stereotype. Note that this information must be provided by the specifier. Concerning implementation of the strategy, the first decision is to assign the responsibility to the UML element class again, since identifiers are a concept bound to classes. Therefore, we use the extension of ClassifierFacade presented in section 3.2. Then, we make the following steps:

1. Extending the AttributeFacade for creating one new facade for each of the three types of stereotypes.
2. Including in the ClassifierFacade extension the restrictions that keys must fulfill (R1-R6 at subsection 2.2) in the form of OCL expressions that will be executed by the AndroMDA Core. There are two exceptions. First, R2 is ensured by construction, since all the attributes stereotyped as Key or WeakKey are considered to be part of just one identifier. Second, R6 is a recursive expression difficult to express in OCL and then it will be treated ad hoc.
3. Implementing the detection code inside the extended metafacade. In this code we look in each class attribute and compare it with the previous mentioned stereotypes to compose the keys. For weak keys, this is done in a recursive way.
4. Implementing the code for the restriction R6. This code is included in the same metafacade in the validation operation invoked by AndroMDA when the specification model to process is validated.

4.3.3 Pre/post Responsibility

In the case of pre/post we have decided to use naming conventions in the textual description, because adding new subtypes of responsibilities means just recognizing a new name. The names used are insertElement, deleteElement, listAll, existsElement and notEmptyPopulation.

The implementation consists of the following steps:

1. Extending the OperationFacade for creating a new facade that may be used later on for dealing with other types of responsibilities assigned to operations.
2. Implementing the detection code inside the extended metafacade. This code gets all the constraints of an operation and compares the element name with the agreed pre/post names.
3. Developing DTOs to represent each of the pre/post responsibilities.

It is worth to remark that in all the cases, when a metafacade is extended, it is necessary to process the new model using Maven to update those Java files that are generated automatically by AndroMDA. Furthermore, it is necessary to modify an XML file (metafacades.xml) to make the new metafacades usable.
5 Conclusions and Future Work

In this paper we have presented AR3L, an AndroMDA-based system for assigning treatments to responsibilities extracted from an analysis model (which takes the form of a UML model), according to some system properties and constraints, in the context of 3-layer-architecture information systems. This assignment is considered as a simplified architecture model, since it reflects which are the technologies (treatments) that compose the architectural solution of the system.

We consider that the most important contribution of our work is to reconcile classical MDD engineering with the important role that software architects play in software engineering. Software architects have lots of knowledge, skills and expertise that are very difficult to include fully in the MDD transformation process. Furthermore, this knowledge, skills and expertise depend on a lot of factors that are difficult to quantify (e.g., information about performance requires lots of benchmarking to be reliable) or to represent (e.g., how to represent information about company’s political or strategic issues?) in a software engine. The need for this kind of mixture between pure MDD and human participation is recognized by other authors, e.g. [13]. There are few proposals that tackle non-functional requirements in the MDD process, e.g. [14], that one is centered on usability evaluation for building user interfaces.

Another contribution has been to use an existing technology like AndroMDA as underlying platform for our prototype. As a result, we have got a system architecture that is highly extensible (with respect to management of responsibilities, system properties and constraints), maintainable (easy to replace some components by others for evolvability or customization), and reasonably portable (the dependency on AndroMDA has been kept to the minimum extent, basically to the part of converting a model responsibility into some invocation to a metafacade operation). It is worth mentioning also that AndroMDA community is very lively (in fact, they recently released AndroMDA 4.0 for public review) which is clearly a point supporting evolvability of our AR3L tool.

Concerning general applicability of our proof-of-concept, we think that the concepts can be abstracted to a wider context and we are already working on this line; concretely we are building a more generic framework called RDT [15], but of course scalability is an open issue that should be experimented soon. The taxonomies that exist in the AR3L reference model may help to tackle this issue, both to incorporate new technologies to the framework as they appear, to establish their effects on properties and constraints, and to use it by working mostly at higher-levels of the hierarchy. On the one hand, both assignments and responsibilities are hierarchically structured, leveraging the complexity of understanding the repositories of these types of elements. On the other hand, hierarchies may be useful for propagation: if some treatment affects a requirement, its ancestors will also affect it; and if some treatment is implemented by a technology, its heirs will too. This default behaviour may be broken e.g., a technology may not implement a particular feature that one could expect. Another point worth mentioning is the use of UML for the functional part of the analysis model, instead of ad hoc domain specific languages, we think that it may make more attractive our approach to the IS development community which may still use their own modeling language.
Future work embraces basically two lines of action. On the one hand, to evolve our current proof-of-concept into a more complete prototype, by developing some case studies more complex that the example shown in this paper. On the other hand, to deeply explore the relationship among the software architect and the MDD transformation process. For instance: in this paper we have proposed a fully automatic responsibility identification process. This has two possible drawbacks. Firstly, as shown in the paper, analysis model must carry more information (stereotypes, naming conventions, etc.) to allow fully automatic identification, and it must be assessed whether this extra modeling effort (which furthermore does not allow importing arbitrary UML models) is worthwhile. Secondly, some decisions may not be known in modeling time, e.g. navigation of associations: a non-navigable role must not generate any association cardinality responsibility.

You may download the AR3L tool implementing the proof-of-concept at www.lsi.upc.edu/~gessi/AR3L.

References

Definición de una Ontología para el Proceso de DSDM considerando Requisitos No-Funcionales

David Ameller y Xavier Franch
Grupo de Investigación GESSI, Universitat Politècnica de Catalunya (UPC)
UPC-Campus Nord, edificio Omega, 08034 Barcelona
{dameller, franch}@lsi.upc.edu

Resumen. La consideración de los requisitos no-funcionales (RNF) en el proceso de desarrollo de software dirigido por modelos (DSDM) impacta en la reducción del estilo arquitectónico del sistema software resultante, así como en la selección de las tecnologías más apropiadas para su implementación. A partir de un análisis de dicho impacto, en este trabajo se enumeran, definen y relacionan los conceptos fundamentales que emergen en el tratamiento de los RNF en el proceso de DSDM, y se ilustra su rol en el proceso mediante un escenarios de uso.

Palabras clave: desarrollo de software dirigido por modelos, requisitos no-funcionales, arquitecturas del software.

1 Introducción

Tipicamente, los requisitos establecidos sobre un sistema software se clasifican como funcionales o no-funcionales. Mientras que los requisitos funcionales nos indican el comportamiento que se espera del sistema, los requisitos no-funcionales (RNF) [1] nos indican las restricciones que éste debe cumplir y las características que se deben potenciar (p. ej., eficiencia, mantenibilidad, usabilidad, etc.).

Una lección aprendida en la ingeniería del software es que el desarrollo de software sin considerar los RNF desemboca en mayores costes tanto para el cliente como para el desarrollador [2, 3]. Aun así, sigue siendo frecuente soslayar este tipo de requisitos. Por ejemplo, en muchas ocasiones la arquitectura y las tecnologías utilizadas vienen prefijadas sin considerarlos siquiera. En esta situación, si no consideramos los RNF de usabilidad, en un proyecto que requiera facilidades para personas discapacitadas, podríamos vernos obligados a cambiar la tecnología usada en el desarrollo de la interfaz del usuario a mitad del proyecto. O bien, podríamos detectar de forma tardía que se requiere un tipo de interoperabilidad entre los usuarios del sistema que nos obligara a replantearnos la arquitectura prefijada en el momento de integrar los componentes de la solución. Desde esta perspectiva, puede decirse que la calidad del software es el “grado en el que un conjunto de características inherentes cumple con los requisitos” [4]. Todo proceso de desarrollo de software debe tener en cuenta esta máxima y el desarrollo de software dirigido por modelos (DSDM) no es una excepción.
En el seno del grupo GESSI (http://wwwlsi.upc.edu/~gessi) de la UPC estamos empezando el desarrollo de ArchiTech, un marco de trabajo para DSDM en el que se consideran los RNF para: 1) educir (del inglés “elicit”) los estilos arquitectónicos y tecnológicos más adecuados para un sistema software en desarrollo, y 2) posteriormente construir la arquitectura software resultante y seleccionar el conjunto de tecnologías correspondientes a dichos estilos que más se adecuen a las necesidades especificadas e implícitas de dicho sistema. ArchiTech es la evolución de RDT [5]. RDT se basaba en la detección de responsabilidades, un concepto demasiado atómico que previsiblemente iba a causarnos problemas de usabilidad y escalabilidad, por ello ese concepto se ha eliminado en ArchiTech, donde nos centramos principalmente en aspectos arquitectónicos y tecnológicos.

El primer problema que hemos encontrado es la necesidad de relacionar todos los conceptos exigidos por ArchiTech. Por ello nuestra prioridad en esta primera fase de investigación es identificar, definir y relacionar con precisión estos conceptos con una ontología. Hemos escogido hacer una ontología ya que exige un cierto grado de rigor, y este rigor nos asegura una base sólida sobre la que construir ArchiTech.

En este artículo proponemos una primera solución al problema identificado mediante la construcción de una ontología. Para ello, en la Sección 2 introducimos las ideas básicas de ArchiTech, en la Sección 3 presentamos la ontología, en la Sección 4 ilustramos ambos mediante un escenario de uso, y en la Sección 5 resumimos las conclusiones e identificamos el trabajo futuro.

2 Impacto de los RNF en el DSDM

El DSDM, y más concretamente el MDA [6], es una forma de desarrollo de software que potencia los beneficios de partir de un modelo muy abstracto, el CIM, e ir bajando este nivel de abstracción paulatinamente pasando por el PIM, posteriormente el PSM y finalmente el código. Actualmente, este proceso de refinamiento semi-automático está centrado en los aspectos funcionales del sistema software, salvo unas pocas excepciones como [7, 8]. Así, los RNF se consideran y se toman decisiones a partir de ellos incluso cuando no forman parte del proceso DSDM, pero de la misma forma en que se han considerado normalmente durante las últimas décadas, es decir una pieza de documentación o un conocimiento implícito del desarrollador.

Nuestra meta en ArchiTech es colocar los RNF al mismo nivel que los requisitos funcionales dentro del DSDM. Ello implicara que podamos modelarlos y que podamos trabajar con ellos de forma semiautomática, cuando el Framework este terminado. La diferencia principal en comparación con los requisitos funcionales, es que los RNF juegan un papel mayoritariamente decisional dentro del proceso DSDM.

La primera observación realizada sobre los RNF está centrada en su nivel de abstracción. Los RNF afectan de una forma u otra al proceso de DSDM dependiendo de su nivel de abstracción. Así, podemos diferenciar entre RNF con consecuencias en la arquitectura (RNF arquitectónicos) o en la tecnología (RNF tecnológicos), dicho de otra forma, requisitos que condicionan la construcción del PIM y requisitos que condicionan la construcción del PSM. Existen otras clasificaciones de RNF para el proceso DSDM como la propuesta por V. Cortellessa en [7] (extendida en [8]), en su
trabajo hacen un uso de los RNF paralelo al proceso DSDM, existiendo de esta forma modelos CIM, PIM y PSM específicos para cada grupo de RNF (p. ej., requisitos de eficiencia). La solución propuesta, en [7], obliga a mantener un gran número de modelos y de transformaciones adicionales. Por el contrario, en ArchiTech, los RNF juegan un papel decisional y ayudan a dirigir el proceso DSDM.

La forma en que los RNF arquitectónicos y tecnológicos afectan al proceso de DSDM es: en primer lugar nos ayudan a educir cuáles son los estilos que más favorecen a los RNF especificados y en segundo lugar nos dirigen el proceso de construcción de la arquitectura y la selección de las tecnologías (v. Fig. 1).

Fig. 1: Impacto de los RNF en el DSDM

3 Ontología para la Integración de los RNF en el DSDM

Esta ontología sigue los principios básicos descritos en [9]. La ontología consta de cuatro grandes grupos: requisitos, estilos, artefactos y modelos. Para todos ellos, definimos los términos básicos y mostramos sus relaciones mediante diagramas de clase UML. Para facilitar la lectura, el número de cada término se usará como referencia (como δ-n). Los términos que son comunes en la ingeniería del software se entrecomillarán con la definición que se considera más adecuada para la ontología (por ello aparecen en inglés). La ontología en una pieza está en el anexo.

Requisitos. Diferenciamos requisitos funcionales de RNF y clasificamos los RNF entre RNF arquitectónicos y RNF tecnológicos (v. Fig. 2).

1. **Requisito funcional:** “A system/software requirement that specifies a function that a system/software system or system/software component must be capable of performing.” [10].

2. **Requisito no-funcional:** “A requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement.” [11].

3. **RNF arquitectónico:** Especifica una necesidad que impacta en la solución arquitectónica (δ-8) del sistema software; p. ej., un NFR que determine la distribución de los componentes tal como: “disponibilidad 24 horas por día, 7 días por semana”, este implicaría la replicación de componentes arquitectónicos (δ-11).
Fig. 2: Diagrama de clases correspondiente a los requisitos

4. **RNF tecnológico**: Especifica una necesidad que impacta en la selección de tecnologías (δ-13) del sistema software; p. ej., un NFR que imponga el uso de una tecnología concreta (“el nuevo sistema debe integrarse con la base de datos del sistema actual”). Puede suceder que un RNF sea a la vez arquitectónico y tecnológico.

Destacamos que en ArchiTech nos focalizamos en los NFRs que nos permiten tomar decisiones arquitecturales o tecnológicas, que se obtienen mediante un proceso de refinamiento de los objetivos iniciales de alto nivel (p. ej., “alta eficiencia”), usando típicamente metodologías orientadas a objetivos [12]. Estos procesos de refinamiento quedan fuera de los objetivos de este artículo. Los NFRs no tienen porqué llegar a ser operacionales, es más, en algunos casos un exceso de detalle puede dificultar el proceso de decisión.

**Estilos**. Los estilos definen un esquema global y reusable que explica cómo deben construirse los artefactos. Diferenciamos entre estilo arquitectónico y estilo tecnológico (v. Fig. 3).


6. Para mejorar la reusabilidad de los estilos arquitectónicos, en esta ontología se propone el concepto de variación del estilo. A diferencia de una especialización, una variación del estilo afecta a una parte parcial del estilo, pudiéndose combinar más de una variación para formar nuevos estilos (v. Sección 4 para ejemplo).

7. **Estilo tecnológico**: No existe una definición en la literatura académica, pero siguiendo el patrón de los estilos arquitectónicos (δ-5), los estilos tecnológicos se representan por una descripción de los roles tecnológicos (δ-12) que los forman. Ejemplos habituales son: “Stack solution”, Java y .NET. Los estilos tecnológicos pueden establecer limitaciones (en la Fig. 3 y 4 están representadas por la clase “Restriction”) sobre las tecnologías (δ-13) que implementan el conjunto de roles tecnológicos. Las tecnologías válidas para un estilo tecnológico han de ser capaces de funcionar conjuntamente. Un estilo tecnológico puede especializarse; p. ej., el estilo LAMP es una especialización del estilo “Stack solution”. Igual que antes, se contemplan variaciones de estilos (δ-6) para los estilos tecnológicos. Este concepto es similar al concepto de **modelo de plataforma** descrito en la guía MDA [6], en nuestro caso los componentes son de más alto nivel.
Artefactos. Un artefacto es una pieza de información concreta del sistema software en desarrollo. Consideramos como artefactos la solución arquitectónica y la solución tecnológica (v. Fig. 3). Los artefactos deben seguir las pautas marcadas por el estilo al que pertenecen.

8. **Solución arquitectónica**: A este término muchas veces se le llama simplemente arquitectura o arquitectura software, pero ello puede provocar confusiones con el término estilo arquitectónico (δ-5). “A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity.” [14]. Una solución arquitectónica está construida siguiendo las pautas impuestas por el estilo arquitectónico al cual pertenece.

9. **Solución tecnológica**: Siguiendo la definición de solución arquitectónica (δ-8), definimos solución tecnológica como la descripción de las tecnologías (δ-13) y sus relaciones, así como el papel que juegan dentro del sistema software. Esta solución debe contemplar las tecnologías necesarias para cubrir todos los roles tecnológicos (δ-12) exigidos por el estilo tecnológico (δ-7) al que pertenece. P. ej. Oracle 11g para el rol de SGBD, Ubuntu 9.0 para el de SO, PHP 4.3 para el de lenguaje de programación, etc.

Modelos. En este caso consideramos la clasificación de modelos de MDA [6]: CIM, PIM y PSM. El papel que tendrían estos modelos en ArchiTech sería el siguiente: El CIM representa el dominio (diagramas de casos de uso, diagramas de clases, etc.), el PIM representa la arquitectura (diagramas estructurados en componentes mostrando sus relaciones), y el PSM representa la tecnología (diagramas adaptados para generar código para un conjunto de tecnologías específicas). El concepto mediador para los modelos son las transformaciones, estas vienen determinadas por las soluciones arquitectónicas y tecnologías. Se ha publicado una versión extendida de este artículo en forma de informe que detalla esta parte de la ontología [15].
Fig. 4: Componentes de los estilos y los artefactos

Para poder enlazar los cuatro grupos de conceptos principales han surgido otros conceptos mediadores que detallamos a continuación:

**Componentes de los estilos y de los artefactos.** Tanto los estilos como los artefactos son elementos que se componen de otros elementos, los componentes. Los componentes pueden componerse a su vez de otros componentes de su mismo tipo. Diferenciamos entre cuatro tipos de componentes (v. Fig. 4).

10. **Tipo de componente arquitectónico:** Un tipo de componente arquitectónico es una pieza de un estilo arquitectónico (δ-5), a su vez define como son los componentes arquitectónicos (δ-11). Los tipos de componente arquitectónicos se pueden especializar; p. ej., la capa es un tipo de componente arquitectónico que se puede especializar en un segundo tipo, la capa de persistencia.

11. **Componente arquitectónico:** Un componente arquitectónico es una realización de un tipo de componente arquitectónico (δ-10) que forma parte de la solución arquitectónica (δ-8) de un sistema software determinado. P. ej., el componente de gestión de usuarios de Easychair podría ser del tipo de componente arquitectónico “módulo” (que suponemos forma parte del estilo arquitectónico de Easychair). Cada componente arquitectónico del sistema software juega un rol concreto para cada uno de los roles tecnológicos (δ-12) requeridos por el tipo de componente arquitectónico al cual pertenece. Estos roles concretos nos permiten seleccionar la tecnología (δ-13) más adecuada a nivel de componente arquitectónico.

12. **Rol tecnológico:** Un rol tecnológico es un tipo de tecnología (δ-13). P. ej., el SGBD, o el servidor Web juegan un rol dentro de un estilo tecnológico (δ-7). Estos roles tecnológicos aparecen por necesidades de los tipos de componentes arquitectónicos (δ-10), por formar parte del estilo tecnológico, o por ser dependencia de otro rol tecnológico. Aunque no todos los roles tecnológicos, ni las tecnologías que los implementan, jueguen un papel en la construcción del PSM, es necesario tenerlos todos para poder asegurar que pueden funcionar conjuntamente.

13. **Tecnología:** Una tecnología es una pieza de software que ofrece unas facilidades para la implementación parcial o completa de alguna funcionalidad del sistema software. Una tecnología puede referirse a un producto tecnológico o bien a una tecnología concreta de un producto tecnológico; p. ej., Oracle es un producto y Oracle 11g es una tecnología concreta. Las tecnologías concretas se usan para
construir soluciones tecnológicas (δ-9) mientras que la finalidad de los productos tecnológicos es poder establecer limitaciones para todas las versiones de un producto; p. ej., la especificación de incompatibilidades entre productos tecnológicos.

**Restricciones derivadas de los requisitos.** Dado que los NFR del sistema suelen expresarse de diversas formas (p. ej., lenguaje natural, Volere, etc.) es necesario traducirlos a una nomenclatura que nos permita trabajar con ellas. La nomenclatura escogida para ArchiTech son las restricciones. El lenguaje para la descripción de las restricciones deberá ser capaz de expresar expresiones aritmético-lógicas.


**Propiedades.** Algunos componentes (los tipos de componentes arquitectónicos (δ-10) y las tecnologías (δ-13)) tienen propiedades que permiten determinar cuando es mejor usar uno u otro por medio de la evaluación de las restricciones (δ-14).

15. Las propiedades son características comunes a diversos elementos alternativos. Los componentes con propiedades establecen valores para estos. (p. ej., la propiedad de poder trabajar de forma distribuida).

## 4 Escenario de Uso

El escenario de uso propuesto está ideado para que podamos constatar cómo los RNF pueden afectar de una forma muy significativa al proceso de desarrollo de un sistema software, incluso partiendo de un mismo conjunto de requisitos funcionales. El escenario escogido para ejemplificarlo es un sistema software para la gestión de conferencias (SGC). Se plantean dos posibles alternativas para este mismo escenario:

- **Alternativa A:** Una prestigiosa editorial quiere desarrollar el SGC e integrarlo en su sistema de información para asegurar la calidad de los artículos que publica. El SGC será usado por personas sin conocimientos informáticos. Se ha impuesto un tope de 100.000€ para financiar el proyecto y un plazo de 3 meses para terminarlo.
- **Alternativa B:** Una gran cantidad de conferencias están siendo gestionadas por universidades españolas. Por esta razón se ha propuesto un proyecto universitario coordinado para desarrollar un SGC de ámbito español. Cada universidad desea guardar los datos de las conferencias que organiza. Se espera que los estudiantes que implementan aprendan las tecnologías necesarias durante el transcurso del proyecto. Los costes serán sufragados por las universidades participantes.

Para cada alternativa, la editorial (A) y los estudiantes (B) han extraído un conjunto de requisitos distintos aplicando metodologías orientadas a objetivos [12]. Sólo nos centramos en los NFR que pueden tener consecuencias arquitectónicas o tecnológicas. Podemos ver el resultado de ambas alternativas en la Tabla 1.

Podríamos llegar a las siguientes conclusiones de forma intuitiva:

- Ambas alternativas parecen adecuadas para un estilo arquitectónico en tres capas debido a que se trata de un sistema de información transaccional. En consecuencia la solución arquitectónica resultante sería similar en los dos casos.
- La alternativa A sugiere que sea una aplicación de escritorio. Este tipo de aplicaciones son más habituales y fáciles de usar. Por otro lado, en la alternativa B,
parece más adecuado pensar en una aplicación Web, ya que para éstas existen muchas más tecnologías de software libre y la mayoría tienen mucho soporte.

Tabla 1. Requisitos extraídos en cada alternativa

<table>
<thead>
<tr>
<th>Alternativa A</th>
<th>Alternativa B</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td><strong>El sistema debe ser desarrollado con .Net versión 2.0</strong></td>
<td><strong>El sistema debe ser desarrollado con software libre</strong></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td><strong>La interfaz del usuario debe ser simple e intuitiva</strong></td>
<td><strong>Las tecnologías usadas deben tener un buen soporte de la comunidad</strong></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td><strong>Los datos deben guardarse en el SGBD de la editorial (Oracle 11g)</strong></td>
<td><strong>Los datos de cada conferencia deben guardarse en la universidad organizadora</strong></td>
</tr>
</tbody>
</table>

- El estilo tecnológico en el primer caso viene determinado por el escenario: .Net, en el segundo caso, probablemente optaríamos por LAMP reforzando así la idea de usar software libre en el entorno Web.
- Dependiendo de la experiencia como desarrollador algunas decisiones serían más claras que otras. Por ejemplo, respecto a la gestión de los datos hay inmediatez de opciones, desde ficheros de texto plano hasta el uso de “data mappers”. En la alternativa A no hay mucho que pensar, el escenario nos impone el uso de un SGBD concreto y posiblemente la mayoría de desarrolladores aprovecharían las facilidades para la gestión de datos que ofrece ADO.Net. En contrapartida, la segunda alternativa es menos clara. La necesidad de gestionar diversas bases de datos desde un mismo sistema nos obliga a conocer mucho mejor los detalles y posibilidades de cada opción tecnológica, no sería una elección fácil ni intuitiva (exceptuando si se tiene experiencia previa en un sistema similar).

Como se puede observar a partir de estas conclusiones, los RNF son cruciales para determinar de forma precisa tanto el estilo arquitectónico como el tecnológico. Además los RNF pueden dirigirnos hacia sistemas software completamente distintos.

Tomando como partida el escenario y la alternativa B, a continuación exploraremos los detalles de cómo se llegaría a una conclusión similar, usando en esta ocasión el razonamiento de ArchiTech. En el momento de tomar decisiones, ArchiTech tiene dos principios a seguir: el primero es que siempre que haya especializaciones priorizará la opción más específica, y el segundo es que si más de una opción cumple todos los NFR, será el usuario quien tendrá la última palabra. La necesidad de los conceptos descritos en la ontología se verá reflejada en este ejemplo.

Partiendo de los requisitos de la Tabla 1 el motor de Architech detecería las siguientes restricciones en función de varias propiedades:

- **C1** - La propiedad “Persistencia” debe incluir el valor “distribuida”.
- **C2** - La propiedad “Tipo de licencia” debe ser igual a “OSS”.

C1 es una restricción arquitectónica derivada de R3-B (Tabla 1) y C2 es una restricción tecnológica derivada de R1-B. Por supuesto que en un escenario real aparecerían muchos más requisitos y en consecuencia muchas más restricciones. Para que el ejemplo sea fácil de entender nos centraremos sólo en estas dos restricciones.

Una vez tenemos las restricciones del sistema software empezaremos por educir el estilo arquitectónico. Supongamos que el marco de trabajo conoce únicamente los siguientes estilos arquitectónicos:

- **AS1** - “En capas”
- **AS2** - “3-Capas”
Educimos primeramente AS2, usando las pautas descritas en [14]. AS2 es una especialización de AS1. Los tipos de componentes arquitectónicos de AS1 son: módulo y capa. Una capa esta compuesta por módulos. Los tipos adicionales necesarios para AS2 son: capa de presentación, capa de dominio y capa de persistencia. Todos son especialización del tipo “capa”. Además AS2 incluye las limitaciones habituales en una arquitectura en 3-capas, expresadas mediante restricciones: la comunicación entre capas así como el número de capas y su tipo.

El marco de trabajo contempla dos posibles variaciones para AS2: persistencia distribuida y persistencia local. Como es lógico, debido a la restricción C1 se escogerá la variación de persistencia distribuida. Ésta usa dos especializaciones de los componentes: capa de persistencia distribuida y módulo distribuido.

Una vez sabemos cual será el estilo arquitectónico podemos iniciar la construcción de la solución arquitectónica. Para ello se usara el modelo CIM y las reglas de construcción del estilo para separar la funcionalidad en diversos módulos y capas. Cuando se haya determinado cómo se compone la arquitectura del sistema software se aplicarán las transformaciones necesarias para obtener el PIM.

Para la parte tecnológica del proceso, supongamos que ArchiTech conoce los siguientes estilos tecnológicos:

- TS1 - “Stack solution”
- TS2 - “LAMP”
- TS3 - “Java”
- TS4 - “.Net”


En este ejemplo TS2 seria educido como el estilo tecnológico porque las tecnologías que requiere son OSS. Como se ha dicho antes, este ejemplo ha sido simplificado. En un caso real habrían muchas más restricciones que satisfacer.

La solución tecnológica sería el conjunto de tecnologías concretas seleccionadas (aplicando las limitaciones de compatibilidad y otras restricciones) para cada uno de los componentes arquitectónicos. P. ej., el módulo de la capa de presentación encargado del envío de artículos se podría asociar con la tecnología “PHP 5.0”. Un posible razonamiento para esta selección podría ser que a partir de la versión 5, PHP soporta el uso de excepciones las cuales son adecuadas para un sistema transaccional.
5 Conclusiones y Trabajo Futuro

En este artículo se han identificado, descrito y relacionado todos los conceptos que han ido apareciendo en el marco de trabajo que se está desarrollando, ArchiTech, para el tratamiento de NFR en el proceso de DSDM.

Consideramos que el trabajo realizado es un buen punto de partida para desarrollar, en un futuro próximo, un prototipo de ArchiTech. Principalmente el propósito del prototipo será validar los conceptos descritos en este artículo en relación a la integración de los RNF dentro del DSDM. Con referencia a los aspectos ontológicos de la propuesta, una vez la propuesta de este artículo se consolide, será preciso complementarla con extensiones que cubran los conceptos principales de ArchiTech: ontología para los RNF, para los estilos arquitectónicos, para las tecnologías, etc. Algunas serán más dinámicas (p. ej., la citada para tecnologías) mientras que otras evolucionarán más lentamente (p. ej., las referentes a estilos arquitectónicos).

References

ANEXO: Diagrama conceptual en una pieza
Usage of architectural styles and technologies in IT companies and organizations

David Ameller and Xavier Franch
Universitat Politècnica de Catalunya (UPC)
{dameller, franch}@lsi.upc.edu

As in many other software engineering activities, Non-Functional Requirements (NFR) are fundamental in the process of selection of the architectural style and the technologies for developing a software system. To know more about this issue, we are currently driving a survey to find out which architectural styles and technologies are being in use in IT companies and organizations, and their relation to types of NFR. We want to apply this knowledge to Model-Driven Software Development (MDSD), and in fact we are currently developing a framework that considers NFR in the MDSD process.

The survey is available in the following URL: http://www.lsi.upc.edu/~dsdm-survey.

Development of the survey: The survey has been developed following an iterative methodology. At each iteration, it has been revised by IT experts and researchers of the area. There have been three iterations, the first one in plain text and the other two using the current electronic format. It was difficult to find good (and cheap) software for developing surveys; finally, we chose LimeSurvey, which is free open source software (FOSS).

For the dissemination of the survey we are using mainly two strategies. On the one hand, personal contact with software architects that we know personally. On the second hand, advertisement in IT communities hosted in common sites such as LinkedIn and Facebook. In this first stage, we have contacted 7 software architects and advertised in the International Association of Software Architects group and we plan to do the same in other similar groups.

Design of the survey: The survey is divided into three parts: first, common questions about software development; second, questions about the degree of desired interaction with development tools; and third, questions about the usage and knowledge of MDSD. The third part of the survey is driven in three different ways depending on the knowledge of the respondents.

In the first part of the survey we have asked about the used architectural styles, the type of developed applications, the technological styles used in them, questions about the use of database management systems (DBMS), and finally questions about NFR.

The second part consists of two questions about the desirable interaction of a hypothetical development tool. The first question is about code generation and the second is about design decisions.

Finally, in the third part the survey may take three different paths. If the respondent doesn’t know the concept of MDSD, the survey finishes. If he/she knows the concept but doesn’t use MDSD in his/her projects, we ask about his/her knowledge about MDSD, namely about the known initiatives, development frameworks, and several questions to find out his/her opinion about MDSD. If he/she is using MDSD in his/her projects, we ask about the usage of MDSD. It is interesting to see in the third case if they are using different architectural styles or technologies when they develop software using MDSD.

Current status: We have, at the time of writing, 56 responses to the survey. We have planed to wait until we have at least 100 responses before the publication of the results. As a preview, we have seen the following tendencies in the current responses.

From the architectural point of view, more than the 65% of respondents use 3-layered or MVC architectural styles followed by client-server (51%) and SOA (46%). The most common type of application is Web-based (75%) followed by Web Services (58%).
Fig. 1: importance of different types of NFR in software projects

From the technological point of view Java-based technologies are used by a 75% of respondents. Only 21% declare to be using .Net-based technologies, which looks a bit strange because many software development companies currently announce that they develop in .Net (may be this fact will be reduced on the final results), and the least used is stack solution (e.g. LAMP) with 14%. JSP is the most used Java technology (57%), but also a significant amount of respondents use Struts (48%) or Servlets (48%).

When we talk about DBMS, relational databases are, as expected, the most used (88%), being Oracle (69%), SQL-Server (33%), and MySQL (32%) are the most widespread brands. It is a bit surprising that MySQL is used only by 32% of respondents while 77% are developing Web-based applications. The only DBMS-related mechanism that has a relevant amount of importance is stored procedures (39% of respondents think that stored procedures are important or even critical in their applications), whilst triggers and checks have very little importance (80% approximately of respondents report that these mechanisms have no or little importance for them).

The importance of NFR is not very clear, we have found a division of opinions: while 96% of respondents consider NFR (73% at the same level as functional requirements), only 57% use NFR to take architectural and technological decisions. In the Fig. 1 we have summarized the reported importance of each individual NFR. We can see that the requirements such as maintainability, reusability, efficiency, reliability, and usability have a tendency of being more important than portability, cost, standard compliance, and organizational NFR.

When we put NFR into practice, 80% of respondents do not use development tools that analyze the NFR compliance, but 70% would like to use such kind of tools.

In the second part of the survey (questions about interaction) the most accepted answer for all questions is that developers want to be asked, but only for the most important decisions. This answer can be related with the view of MDSD not as a fully automatic process but as an assisted one, in which some decisions are taken by the expert.

A great part of the respondents (79%) do not use MDSD in their software projects, so it is difficult to carry out a good evaluation of this part of the survey at this moment. On the other hand, 50% of them declare to know the concept. Eclipse EMP seems to be the most known platform for MDSD and MDA the most known methodology.

This was an overview of the current answers to the survey, for the final report on it we will do a deeper statistical data analysis. For example, we will analyze for each architectural style or technological style (these terms are defined in the survey) which are the most important NFR.

Conclusions: This survey can be seen as a particular instrument that addresses some of the questions raised in the EASA’09 cfp. Our position is that a way to obtain empirical evidence about the current state of software architectures usage in IT companies and organizations is asking the involved actors.
How do Software Architects consider Non-Functional Requirements: A Survey

David Ameller and Xavier Franch

Universitat Politècnica de Catalunya (UPC)
{dameller, franch}@essi.upc.edu

Abstract. [Context and motivation] Non-functional requirements (NFRs) play a fundamental role when software architects need to make informed decisions. Criteria like efficiency or integrity determine up to a great extent the final form that the logical, development and deployment architectural views take. [Question/problem] Continuous evidence is needed about the current industrial practices of software architects concerning NFRs: how do they consider them, and what are the most influential types in their daily work. [Principal ideas/results] We ran a web survey addressed to software architects about these issues. We got 60 responses that give some light to the questions above. [Contribution] Some empirical data has been gathered from industry. The results of this survey may serve as input for researchers in order to decide in which types of NFRs may be necessary to invest more research effort.

Keywords: Non-Functional Requirements, Software Architectures, Web Survey.

1 The Survey

The survey has been developed following an iterative methodology. Each iteration has been revised by IT experts and researchers of the area. For the implementation we chose LimeSurvey, an open source project for developing surveys.

For the dissemination of the survey we used two strategies. On the one hand, personal contact with software architects and on the second hand, advertisement in IT communities hosted in common sites such as LinkedIn and Facebook. We have contacted more than 10 software architects and advertised in the International Association of Software Architects (IASA) group. The survey was running during the year 2009.

The survey had questions about software development. Concretely, we asked about the used architectural styles, the type of developed applications, the technologic platforms used in them, and questions about Non-Functional Requirements (NFRs).

In this work we show the results about NFRs and their relationship to the used architectural style, the type of developed application, and the used technologic platform.
The results

We had 60 responses to the survey. The main results of this survey about NFR may be summarized as follows:

- Respondents answered about the importance of NFRs in their habitual software development practices: while 96% of respondents consider NFR (73% at the same level as functional requirements), only 57% use NFR to make architectural and technological decisions.

- Respondents rated nine types of NFRs with respect to the importance to their projects as shown in Fig. 1. Requirements such as maintainability, reusability, efficiency, reliability, and usability have a tendency of being more important for architects than portability, cost, standard compliance, and organizational NFR.

- 80% of respondents declared that the development tools that they use are not well-suited for analysing the compliance with the specified NFRs, whilst 70% would like to have them. For us this is a clear indicator that there is an unsatisfied need in software industry.

Other results (e.g., some relations between NFRs and used architectural styles) were also found when analyzing the data gathered.

Conclusions

This survey can be seen as an instrument to show the differences in software development practices between research and industry. In particular, we show the impact of NFRs in the software development practices.

Our position is that a way to obtain empirical evidence about the current state of software architectures usage in IT companies and organizations is asking the involved actors.
In 3-Layer Architecture Maintainability is slightly more important than Reliability

SOA and MVC architects gave higher importance to all requirements (about 0.5 points extra)

Reliability is the only NFR considered critical by a majority of respondents

Organizational and Portability NFRs where considered medium importance by a majority of respondents

Surprisingly Cost is not on top position for architects

60 software architects answered the survey

The most influential NFR is Reliability

The least influential NFR is Portability

NFRs importance in software architectural styles:

In 3-Layer Architecture Maintainability is slightly more important than Reliability

SOA and MVC architects gave higher importance to all requirements (about 0.5 points extra)

Technical profile:
— Population: 60 respondents (>50% from Spain)
— Medium: Electronic Survey (Limesurvey)
— Begin/End: March 2009 - October 2009

David Ameller and Xavier Franch
(dameller, franch)@essi.upc.edu
Universitat Politècnica de Catalunya - GESSI
Dealing with Non-Functional Requirements in Model-Driven Development

David Ameller, Xavier Franch
Universitat Politècnica de Catalunya (UPC)
Barcelona, Spain
{dameller, franch}@essi.upc.edu

Jordi Cabot
INRIA-École des Mines de Nantes
Nantes, France
jordi.cabot@inria.fr

Abstract—The impact of non-functional requirements (NFRs) over software systems has been widely documented. Consequently, cost-effective software production method shall provide means to integrate this type of requirements into the development process. In this vision paper we analyze this assumption over a particular type of software production paradigm: model-driven development (MDD). We report first the current state of MDD approaches with respect to NFRs and remark that, in general, NFRs are not addressed in MDD methods and processes, and we discuss the effects of this situation. Next, we outline a general framework that integrates NFRs into the core of the MDD process and provide a detailed comparison among all the MDD approaches considered. Last, we identify some research issues related to this framework.

Keywords-non-functional requirements; model-driven development.

I. INTRODUCTION

Non-functional requirements (NFRs) are one of the main targets of research in the Requirements Engineering community [1] and their impact on practice has been documented in seminal papers [2], individual case studies [3] and types of industrial projects [4]. Given this reported impact of NFRs, we may say that any reliable and efficient software production process shall adequately handle them.

A software production paradigm that is gaining acceptance in the last years is Model-Driven Development (MDD) [5]. According to [6], “Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing”. In other words, MDD uses models as the primary artifact of the software production process, and development steps consist of the (semi-)automated application of transformation steps over these models. Due to its promised benefits, MDD is being one of the main issues of communities and research groups like OMG, and is also mentioned as a driver in particular types of systems (e.g., [7] for self-adaptive systems).

According to the statement above, we may wonder whether current MDD approaches integrate NFRs in the production process. We will show in the paper that most current MDD approaches only focus on system functional requirements when generating system models, not integrating NFRs into the MDD process. Disregarding NFRs will usually provoke that the generated system does not completely satisfy some (if not all) of the stakeholders’ expectations represented by NFRs. We believe that this is a strong argument against current MDD approaches that limit their success and applicability, and hampers their adoption by the industry.

In this vision paper, we are interested in identifying the challenges to overcome in order to effectively integrate NFRs into the MDD production process. To do so, we first provide more details about the current state of the art of MDD with respect to NFRs, understanding the limitations of the MDD methods that are not able to deal with NFRs, and analysing the approaches that apply some kind of treatment to NFRs. Next, we visualize a MDD general framework that smoothly integrates NFRs into the MDD process and discuss some variations. Last, we formulate some challenges and research lines stemming from this framework. To exemplify and motivate our findings, we use an academic exemplar about the development of a web portal for a travel agency.

II. BACKGROUND: MODEL-DRIVEN DEVELOPMENT

MDD is a development paradigm where models (and their transformation) play a fundamental role [5][6]. In MDD, models are used to specify, simulate, verify, test and generate the system to be built.

The most popular MDD method is the Model Driven Architecture approach, an OMG standard [8], that has been used as the basis for many other later MDD methods. MDA distinguishes several types of models. Platform Independent Models (PIM) specify the software system in an independent way from the technology platform chosen to implement it. Platform Specific Models (PSM) refine the PIM to specifications of the implementation platform. That is, two different implementations of the same system would share the same PIM but have two different PSMs, each one adapted to the technological capabilities of each platform. A third type of model, Computation Independent Models (CIM, a kind of business model), exists, but in this paper, we will focus on the transformation from PIM to PSM.

Model-to-Model (M2M) transformations evolve a PIM into a PSM. Last, Model-to-Text (M2T) transformations are used to generate the executable system from the PSM. This step includes generating several code artifacts glued together: Java business classes, Oracle DB schemas, etc.

Fig. 1 summarizes the models and transformations considered in this paper.

![Figure 1. The MDA approach: models and transformations](image)

1
III. MOTIVATION: THE TRAVEL AGENCY WEB PORTAL CASE

In this section we present an academic exemplar that we will use in the rest of the paper for illustration purposes.

The ACME travel agency offers transportation and accommodation services. The management has decided to deploy a web portal in order to offer some online functionalities to its customers, e.g. user management, payment facilities and searches (hotels, flights, etc.). Together with these functionalities, many NFRs appear during the requirements elicitation process. E.g. since the portal is providing e-commerce transactions, security requirements like R1 = “The system shall detect and report unauthorised data accesses” are a must. The effect of this NFR can be manifold, for instance in a Web-based environment, firewalls are an architectural solution that supports this goal.

Other NFRs depend on the specific characteristics of the travel agency and the planned portal usage. For illustration purposes, let’s consider two scenarios:

- **Scenario 1.** ACME is a specialized travel agency that offers luxury vacation packages to exotic destinations in 5-star hotels. It has a reduced portfolio of clients that plan their vacations using the system.

- **Scenario 2.** ACME is a world-wide leader travel agency. The company offers hundreds of packages that are assembled by combining data imported from other transportation and accommodation sites.

These scenarios impose some particular NFRs that capture their most essential characteristics. Thus, in Scenario 1, the number of expected visits is not too high and therefore scalability is not an issue. On the contrary, scalability and availability are key concerns to ensure the success of the portal in Scenario 2. Clearly, a good production process should be sensitive to these differences and should result in different systems for each scenario. To make this statement more evident, let’s consider one particular system dimension, the deployment architectural view as defined by Krutchen [9].

The deployment architectural view refers to the physical distribution of the software system components. Since the system we are considering as exemplar is a Web application, we may identify the following types of components [10]: the Web Server (WS), the Application Server (AS) and the Data Base Management System (DBMS). All these components can be deployed on the same node (Single Server Configuration, SSC), or using one of the several possible separations of components (e.g., separation of the DBMS). Also in the design of the deployment architecture it is possible to consider any type of component replication. Each deployment strategy affects some software quality attributes [11]. For instance, component replication (e.g. WS and AS) supports scalability, because more simultaneous connections may be established; replication also may improve efficiency especially if a load balancing component coordinates the incoming traffic. Table I sums up the effect of these strategies on some common architectural properties, according to [10].

At this point, the software architect has the duty of choosing the most adequate deployment strategy for the given set of NFRs, by comparing them with the effect of each strategy on the quality attributes. For the two scenarios described above, examples of convenient options are:

- **For Scenario 1**, the DBMS is kept separated from the WS and AS since scalability and availability are not major concerns, whilst security is increased by placing a firewall between the DBMS and the other two components (see Fig. 2, a). Replication is not implemented since its benefits are again concerning criteria that are not important for the given NFRs, whilst others would be damaged.

- **For Scenario 2**, since the agency provides a world-wide service, the WS and AS are replicated to improve availability and performance in those sites for which a greater number of clients may be expected. A load balancing system coordinates the different WS to improve performance even more. DBMS containing data local to the sites are put together with the WS and AS, and firewalls are also deployed for protecting each local DBMS. As a final decision, a centralized DBMS contains some replicated data that may be of interest for performing some data mining operations. Fig. 2, (b), provides the whole picture.

Other deployment options are possible. It is not a goal of this section to discuss them, but just to emphasize the fact that the final form of the software architecture depends on the set of elicited NFRs and to give some initial idea of the type of knowledge to manage and decisions to be made.

**TABLE I. EFFECT OF COMPONENTS’ DEPLOYMENT ON SOME ARCHITECTURAL PROPERTIES**

<table>
<thead>
<tr>
<th>SSC</th>
<th>DBMS separated</th>
<th>DBMS &amp; AS separated</th>
<th>Replication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Poor</td>
<td>Average</td>
<td>Good</td>
</tr>
<tr>
<td>Scalability</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
<tr>
<td>Availability</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Good</td>
<td>Average</td>
<td>Average</td>
</tr>
<tr>
<td>Security</td>
<td>Poor</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Complexity</td>
<td>Good</td>
<td>Average</td>
<td>Poor</td>
</tr>
</tbody>
</table>

**Figure 2.** Two different deployment architectures for the Web portal case.
IV. NON-FUNCTIONAL REQUIREMENTS IN MODEL-DRIVEN DEVELOPMENT: STATE OF THE ART

In the previous section, we have shown that NFRs have an important effect in the final form that the software system takes. If we consider MDD, we may say that an optimal MDD production process should be able to deal with the set of elicited NFRs and use them to select and apply the most adequate transformations, in order to generate a software system that satisfies the desired NFRs. In this section we investigate to what extent this need is currently fulfilled.

We distinguish MDD approaches that do not consider NFRs as part of the transformations, from those that do.

A. MDD Approaches not supporting NFRs

We may find a great variety of MDD-based approaches in the literature, many of them following the two-level (PIM and PSM) classification introduced in the OMG’s MDA approach [8]. Among the most popular ones, we find the Executable UML proposals, with [12] as the most popular representative. Executable UML methods use a reduced subset of UML that it is directly executable, either using UML interpreters or by providing a direct translation from the models to the final code.

Using such Executable UML methods, the travel agency model consists of use case diagrams, class diagrams, sequence diagrams and activity diagrams that express the roles, functionalities, data and behaviour of the system. None of these artifacts is able to express any kind of NFR. Thus, the transformation from PIM to PSM is fixed and it is not possible to choose the most appropriate strategy for a given set of NFRs: the PSM will be close to, or far from, the elicited NFRs depending on the system quality factors implicitly encoded in the predefined transformations. Some action is required in order to make the MDD approach effective.

We believe that this a critical situation, even more considering that this Executable UML method [12] is the basis for the upcoming OMG standard “Semantics of a Foundational Subset for Executable UML Models” that pretends to increase the use of UML in a MDD context.

If we consider the general form of MDD (see Fig. 1), we may envisage two different, non-exclusive approaches to make a generated product compliant with the stated NFRs:

1. The software developer directly modifies by hand the result of the MDD process (see Fig. 3, left). In its simplest form, she directly modifies the code obtained after the final M2T transformation. In the best case, she will able to work at the PSM level, modifying the model to adapt it to the NFRs, and then use the M2T transformation (possibly modified somehow) to generate the code. This manual adaptation of the system collides with the essence of the MDD paradigm and has several drawbacks:
   o Takes longer to produce the software.
   o Provokes lower reliability of the final product due to the human-based post-process.
   o Damages traceability and thus comprehension.
   o In case of changes due to maintenance, either the post-process has to be replicated or the maintenance is directly made on the final product.

2. The MDD engineer modifies the M2M transformation in order to obtain a PSM that satisfies the NFRs (see Fig. 3, right). In our example above, we could have three transformations for producing PSM compliant to the SSC, DBMS separated, and DBMS and AS separated, strategies. The drawbacks above are therefore solved, but others appear in their place:
   o The complexity of the MDD framework is greater, because there are more transformations to maintain.
   o It is difficult to anticipate all the possible scenarios, in fact it may be even impossible (e.g., in Table I, replication may be applied in many different ways, and each would require a different transformation).
   o The selection of the most appropriate transformation (for the given set of NFRs) to apply relies on the software architect, becoming a human-based pre-process, incrementing thus the likelihood of errors in decision-making.
   o When the software architect realizes that the available transformations are not adequate for the current process it is necessary to build a new ad-hoc one, making the initial configuration time longer.

The two approaches presented above represent two extreme cases. Hybrid solutions may also exist, where some NFRs are addressed by the M2M transformation and others remain under the final responsibility of the developer.

To sum up, we may state that MDD approaches that are not able to deal with NFRs in the software production process suffer from severe drawbacks that must be manually fixed by either the developer or the MDD engineer and that, therefore, may compromise their adoption.

The situation is even worse when considering not the theory but the real state of practice of MDD, hampered by the limitations of MDD tools available in the market. For instance, their code-generation capabilities are limited to particular technologies/languages (which implies that usually only some parts of the system can be transformed and generated by the tool) and it is not always possible to change the predefined M2M and M2T transformations offered by the tool. Therefore, a scenario more realistic than those depicted in Fig. 3 is described below (see Fig. 4):

- The MDD engineer specifies a PIM that contains only information about system functional aspects.
- The software architect defines (or chooses from the modeling tool she is using) a set of transformations that are applied to different parts of the PIM, generating each an unrelated part of the target PSM. Each generated PSM part is compliant with a particular technology.
NFRs in the MDD process, we have set up a Systematic B.

NFRs using UML extensions [14][15][16], including the NFRs and/or for a specific domain (see Table II).

In what follows we provide some additional details.

- M2T transformations are applied to the PSM for obtaining the final code.
- The developer complements the generated code and combines the generated code-excerpts into a coherent architecture.

This process is adding some new drawbacks:
- There is not a single transformation generating a complete PSM, but a set of partial transformations generating separated pieces that may yield an incomplete PSM. Even, some tools skip the generation of the PSM and jump directly to the code.
- The different pieces generated by the transformations need to be manually linked, writing additional glue code.
- With respect to NFRs, each transformation results on PSM parts that may not satisfy the stated NFRs (in fact, depending on the available transformations each excerpt can enforce different and maybe contradictory NFRs).

B. MDD approaches that deal with NFRs

To know about the approaches that currently deal with NFRs in the MDD process, we have set up a Systematic Literature Review [13] that we briefly describe belos.

A.

Specific formalism (different for each NFR) in which the analysis can take place. Examples are [25][19][26][27]. In these approaches each kind of NFR may be seen as a whole dimension of the software. [28][25] propose analyzing each NFR type separately and also to use different abstraction levels for NFRs (at CIM, PIM and PSM levels).

As a conclusion, we may say that although several valuable approaches have been proposed that deal with NFRs in the MDD process, none of them propose an integrated view, which is the goal of this vision paper.

V. NON-FUNCTIONAL REQUIREMENTS AS PART OF THE MODEL-DRIVEN DEVELOPMENT PROCESS

In the previous section we have shown that MDD approaches that do not consider NFRs as part of the generation process suffer from serious drawbacks, and that, unfortunately, this is the predominant type of approach nowadays. In this section we discuss a general solution to this problem.

A. Basic concepts for dealing with NFRs in MDD

Many authors have reported the intimate relationship among requirements and architectures and also the great impact that NFR have on architectures [29][30][31]. For example, in the analysis of Section IV, we have shown how new components (e.g., firewalls and load balancers) and physical component allocation (e.g., replication) can be justified in terms of the NFRs that must be satisfied. Therefore, we envisage an approach to MDD in which the PIM is transformed into a complete software architecture. Transformations have the mission of allocating the responsibilities coming from the PIM functional part to components that are deployed into an architecture that satisfies the NFRs.
But NFRs are also important when determining the choice of technologies needed to implement the architecture. For instance, it may be necessary not just to know that a relational data base is needed, but also that a particular brand, or even version and release, is the right choice. Interoperability requirements (e.g., “The portal shall be compatible with our current data base in the central management system”) or non-technical requirements [32] (e.g., “The data base vendor shall provide 24×7 call center assistance”) are clear examples of NFRs with this effect.

Table III describes the main elements that constitute our envisioned framework proposal. Remarkably, and following the discussion above, we introduce two kinds of models between the PIM and the code: the model representing the architecture, and the model representing the technological solution. Whilst the latter is clearly a PSM, the former lays in between the two levels of abstraction and therefore we denote it by PIM/PSM. For each kind of model, we include between parentheses the requirements that are satisfied by the elements in that model. Finally, as a consequence of having two different intermediate models among the PIM and the code, we have two corresponding M2M transformations, M2M_{arch} and M2M_{tech}.

B. An NFR-aware MDD process: Integrating NFRs into the PIM

We believe that the most natural way to integrate NFRs into the MDD process is by considering NFRs from the very beginning of the development process, i.e. as part of the PIM. As functional requirements, NFRs become first-order citizens of the MDD process.

The MDD process then works as follows:

- The analyst specifies a PIM that contains both the functional and non-functional requirements, PIM(f + nf).
- The MDD decisional engine decides, given the PIM(f + nf) and the contents of the MDD knowledge base (i.e., information about non-functionality, architectures and technologies), the final form of the transformation M2M_{arch}:

\[
\text{M2M}_{\text{arch}}: \text{PIM}(f + nf) \rightarrow \text{PSM}(f + nf_0)
\]

This transformation takes PIM(f + nf) as input and produces PIM/PSM(f + nf_0), a model describing an architecture that implements all the functionality \( f \) in a way that satisfies the elicited subset of NFRs \( nf_0 \) whose satisfaction depends on the decisions made at the architectural level.
- Once the PIM/PSM(f + nf_0) has been generated, the MDD decisional engine applies a second M2M transformation that generates the PSM for the desired final implementation technology. This PSM follows the architectural guidelines expressed above (and thus, satisfies \( nf_0 \)) but also takes into account all the remaining \( nf \) (directly related to technologies, as those mentioned in V.A), forcing the adoption of a particular technology or product:

\[
\text{M2M}_{\text{tech}}: \text{PIM/PSM}(f + nf_0) \rightarrow \text{PSM}(f + nf)
\]

- Last, a simple M2T transformation can be applied to obtain the code from the technology:

\[
\text{M2T}: \text{PSM}(f + nf) \rightarrow \text{Code}(f + nf)
\]

In the framework, the transformations are presented as single functions. In fact, this is a simplified view since a transformation will be in fact a composition of the application of many transformation rules. Thus, we may say that:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f, nf )</td>
<td>The elicited functionality and non-functionality of the system (not represented as model)</td>
<td>An IEEE 830-compliant Software Requirements Document</td>
</tr>
<tr>
<td>PIM(( f ))</td>
<td>PIM that specifies some functionality ( f ) of the system</td>
<td>A UML class diagram specifying the system data</td>
</tr>
<tr>
<td>PIM(( f + nf ))</td>
<td>PIM that specifies all the requirements of the system</td>
<td>An i*/model of the system complemented with UML data and behavioural diagrams</td>
</tr>
<tr>
<td>PIM/PSM(( f + nf ))</td>
<td>Model mixing PIM and PSM levels that specifies some functionality ( f ) satisfying the NFRs ( nf )</td>
<td>A 3-layer architecture expressed with the ACME Architectural Description Language (ADL)</td>
</tr>
<tr>
<td>PSM(( f + nf ))</td>
<td>PSM that specifies some functionality ( f ) satisfying the NFRs ( nf )</td>
<td>A model with a class diagram annotated with database stereotypes (e.g. &lt;&lt;PK&gt;&gt;, &lt;&lt;Table&gt;&gt;) that only have meaning for the Oracle DBMS</td>
</tr>
<tr>
<td>Code(( f + nf ))</td>
<td>Executable system that implements the functionality ( f ) satisfying the NFRs ( nf )</td>
<td>Implementation of the 3-layer architecture above using Java components, XML interchange data formats, Oracle DB schema, etc.</td>
</tr>
<tr>
<td>M2M</td>
<td>M2M transformation from a PIM to a PSM</td>
<td>Transformation of a UML specification into a technological solution including an Oracle data base and a Pound load balancer, among others</td>
</tr>
<tr>
<td>M2M_{arch}</td>
<td>M2M transformation from a PIM to a PSM/PSM that represents the architecture of the system</td>
<td>A mapping from an Executable UML model of functionality into a 3-layer architecture expressed with the ACME ADL</td>
</tr>
<tr>
<td>M2M_{tech}</td>
<td>M2M transformation from a PIM/PSM into a PSM that represents the technological solution of the system</td>
<td>Transformation of the ACME architectural model into a representation of technology that, for instance, annotates a class diagram with Oracle-compliant database stereotypes</td>
</tr>
<tr>
<td>M2T</td>
<td>M2T transformation from a PSM to the executable system</td>
<td>Transformation of a stereotyped UML diagram to EJB Java classes</td>
</tr>
</tbody>
</table>
being M2M either M2M_{arch} or M2M_{tech}. From a conceptual point of view, the vision of the transformation as a single function is a convenient simplification that does not hamper the generality of the approach.

C. Example: deciding the need of firewall components

In this example we illustrate the kind of information to record, and steps to apply, in order to derive part of the architectural model for the Web portal example presented in Section III. We remark that the notations used to represent the models, and even the concrete steps taken and their order are just an example of how they may look like, we refer to Section VI for further discussion.

We distinguish three parts: the knowledge base used by the MDD decisional engine; the creation of the starting PIM; and the application of our MDD process itself. For the latest, we will restrict to the creation of the PIM/PSM.

1) Representing the MDD knowledge. We focus on the concepts directly related to NFRs. First, it is necessary to represent the types of NFRs managed and the consequences that architectural decisions may have on them. We can represent this using a tabular structure (like Table I) or by means of a notation like the NFR framework [30], used with similar purposes in several works (e.g., [33][34]). The model depicted in Fig. 5, top, shows an excerpt of the information needed, with several softgoals to represent the NFRs and two particular operationalizations for them (each with a different positive/negative effect on them).

Next, it is necessary to represent the implications of each operationalization on the architecture. This is described for the firewall case in the lower part of Fig. 5. The firewall solution requires three participants: the firewall component itself, and two subsystems that are connected, the internal (i.e., protected) and the external ones. These elements are in fact instances of architectural metalelements, e.g. subsystem, defined according to some architectural ontology like those in [9][35].

2) Creating the PIM(f+n). The process starts with the PIM definition. For the functional part we can still follow any existing proposal, e.g. Executable UML. For the NFRs, we may decide to use a natural language representation based on requirement patterns as in [36]. which allows to establish easily the link between such NFRs and the predefined NFRs types in the MDD knowledge base (KB). For instance, Fig. 6 represents the R1 NFR (see Section 3) and the link with the Security NFR type maintained in the KB.

3) Creating the PIM/PSM. The following actions are taken to process R1:

- The MDD decisional engine chooses, using some appropriate analysis technique (e.g., [30][37]), the Firewall operationalization to support R1.
- As a consequence of the system being a Web application (which is a decision coming from the intrinsic nature of a Web portal), a transformational step decomposes the system into three main subsystems: WS, AS and DBMS. The MDD Knowledge Base knows that the communication between these subsystems is: WS \leftrightarrow AS \leftrightarrow DBMS.
- The assignment of elements from the functional part of PIM(f+n) into WS, AS and DBMS, takes place. In particular, the data model elements are assigned into DBMS.
- Since R1 is referring to data protection, and since DBMS is bound to data, the MDD decisional engine decides that the protected subsystem for the firewall is the DBMS. Since the communication for Web application is from AS to DBMS, it is also possible to deduce that the “source” of the Firewall is the AS.
- In Scenario 1, since there is no replication, there are just one AS and one DBMS, and thus just one Firewall is induced (see Fig. 6). In Scenario 2, due to replication, there are as many Firewalls as pairs AS-DBMS. The fact that the WS and the AS are deployed together completes the information needed to determine the final form of the architecture.
• To determine the most adequate formalism for representing the non-functional part of PIM($f^+nf$).
  We have used in the example the NFR framework, that is basically a qualitative-oriented one, but also more quantitative approaches may be considered, e.g. in QoOS-style [38].

• To embody in the MDD decisional engine all the knowledge needed to make informed architectural decisions, i.e. to determine the concrete form that the M2M functions take. In other words, the M2M are required to provide a correct output in all possible situations. This is a very strong condition mainly because of: 1) the amount of knowledge to represent is huge and not always clear; 2) the conflicting nature of NFRs: architectural decisions permanently require trade-off analysis.

These problems lead to propose a second alternative specially interesting until clearly accepted technical solutions for the previous points are provided. Instead of considering NFRs as part of the PIM and then be an input of the MDD process, we may consider that the MDD process asks the software architect the NFR-related information as it is needed. The resulting process becomes:

- The analyst specifies a PIM that contains just the functional requirements, PIM(f).
- The transformation function M2M$_{arch}$ takes PIM(f) as input and produces PIM/PSM($f^+nf_0$) ($nf_0$ stands again for those NFRs that concern the architecture). To produce this output, the MDD process presents a series of questions $Q = \{q_1, ..., q_n\}$ to the software architect whose answer is needed in order to decide the transformation steps to apply. The software architect provides answers $A = \{a_1, ..., a_n\}$ according to the NFRs $nf_0$. If we denote by $\sigma_{arch}$ the function that records the mapping from questions to answers, $\sigma_{arch}(q_i) = a_i$, the transformation function is defined as:

\[
M2M_{arch} \colon \text{PIM}(f) \times \sigma_{arch} \rightarrow \text{PIM/PSM}(f^+nf_0)
\]

- The subsequent M2M transformation for the technology acts the same, requiring a similar $\sigma_{tech}$ function to obtain from the MDD engineer the information needed to make informed decisions:

\[
M2M_{tech} \colon \text{PIM/PSM}(f^+nf_0) \times \sigma_{tech} \rightarrow \text{PSM}(f^+nf)
\]

- The M2T transformation is not affected:

\[
M2T \colon \text{PSM}(f^+nf) \rightarrow \text{Code}(f^+nf)
\]

Questions that the MDD decisional engine may raise to the architect may be manifold. For instance, there may be high-level questions like the type of organization with respect to departments (e.g., to decide which nodes are part of the physical architecture) and lower-level ones like the probability of execution of a given operation or use case.

The two NFR-aware approaches presented in this Section V represent two extreme visions but of course we can think of hybrid solutions, in which the MDD decisional engine supports decision-making for some types of NFRs, architectural elements and technologies, whilst the software architect and developer may provide the information missing under demand.

E. Comparison

In this section we compare the two NFR-aware approaches presented in this section with the three approaches presented in Section IV. Fig. 7 aligns the five approaches for an easier comparison. When comparing, please pay attention to: the number and nature of models and transformations; the extent of requirements in the models (enclosed in parenthesis); and the type of interaction with the human assistant (where, and in which direction). Table IV includes a detailed comparison respect to several criteria.

In short, the main benefits of NFR-aware approaches are:

- NFR-aware approaches fully integrate NFRs into the MDD process. Especially in the first NFR-aware framework presented (Fig. 7(d)), NFRs are considered at the same level than the functional specification, being both part of the departing PIM. Knowledge may be incrementally stored in the MDD knowledge base (gradually improving accuracy of results) and may be reused in each new project.

- As a consequence, there is no need for the developer neither to write glue code (since the different components of the PSM model are already interrelated) nor to adapt the code to satisfy the NFRs (since the NFRs have been already taken into account when creating the PSM model).

- Instead of obtaining several incomplete PSM, using a single transformation that targets a specific architecture a single, a comprehensive and unified representation of the system is derived.

- Two levels of abstraction are recognized, one for representing architectures, other for representing technologies. This distinction fits with the levels of abstraction that practitioners use in their daily work.

- The explicit representation of NFRs allows defining model transformation repositories inside the MDD knowledge base that can be used to select the proper transformations to apply. Also, when NFRs are considered at the PIM level, classical analysis techniques from Requirements Engineering may be applied in the early stages of the MDD process.

- Hybrid approaches (between options from Fig. 7(d) and 7(e)) allow customizing the NFR-awareness to the resources, skills and preferences of software architects. For instance, an empirical study that we recently conducted shown that software architects are reluctant to lose all the control over the architectural decisions to be made [39].

But as the Table IV shows, these benefits are not for free. Incorporating NFRs results in higher modeling effort, both for constructing the PIM and for building the MDD knowledge base. Also, it requires discipline to keep this MDD knowledge base up to date. Complexity of the MDD process is the overall challenge to overcome.
TABLE IV. COMPARISON AMONG THE DIFFERENT MDD STRATEGIES ANALYSED IN THE PAPER

<table>
<thead>
<tr>
<th>MDD approaches not dealing with NFR</th>
<th>NFR-aware MDD frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 7(a)</td>
<td>Fig. 7(b)</td>
</tr>
<tr>
<td>Fig. 7(c)</td>
<td>Fig. 7(d)</td>
</tr>
<tr>
<td>Fig. 7(e)</td>
<td>Fig. 7(f)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project set-up</th>
<th>MDD configuration time for a particular project</th>
<th>Production process</th>
<th>Criticality of human intervention during production</th>
<th>Complexity of the process</th>
<th>Knowledge reuse and learning ability</th>
<th>MDD KB maintenance cost</th>
<th>Product Traceability</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fair (just functionality is modeled)</td>
<td>Probably high (if a new transformation is needed)</td>
<td>High (full post-process adaptation)</td>
<td>High (high responsibility of the developer at the end)</td>
<td>Low (the MDD infrastructure is static)</td>
<td>Very low (just the functional-related knowledge is reused)</td>
<td>Low (since it just covers functionality)</td>
<td>Very low (generated product modified)</td>
<td>Very low (changes made are probably lost if product generated again)</td>
</tr>
<tr>
<td>Fair (just functionality is modeled)</td>
<td>None (transformations applied as are)</td>
<td>Fair (slight post-process adaptation will probably be needed)</td>
<td>Fair (slight post-process adaptation will probably be needed)</td>
<td>High (several transformations co-exist)</td>
<td>Low (learning ability comes from the MDD engineer)</td>
<td>Fair (updates up to the MDD engineer)</td>
<td>Fair (depending on the complexity of the post-process adaptation)</td>
<td>Fair (depending on the complexity of the post-process adaptation)</td>
</tr>
<tr>
<td>High (several notations used to build the PIM)</td>
<td>None (transformations applied as are)</td>
<td>Very high (post-process adaptation and gluing)</td>
<td>High (high responsibility of the developer at the end)</td>
<td>Very high (several heterogeneous transformations exist)</td>
<td>Very low (just the functional-related knowledge is reused)</td>
<td>Fair (updates up to the MDD engineer)</td>
<td>Extremely low (generated product modified; information across models)</td>
<td>Very low (changes made are probably lost if product generated again)</td>
</tr>
<tr>
<td>Very high (NFRs need to be modeled)</td>
<td>None (transformations applied as are)</td>
<td>None (if transformations are complete)</td>
<td>None (since there are no human interactions)</td>
<td>High (the transformations used will be more complex)</td>
<td>Very high (NFR-related knowledge may be reused and may grow)</td>
<td>Very high (all new knowledge needs to be modeled)</td>
<td>Potentially complete (all decisions can be traced)</td>
<td>Very high (it is possible to work only at PIM level)</td>
</tr>
<tr>
<td>Fair (just functionality is modeled)</td>
<td>None (transformations applied as are)</td>
<td>Low (guided conversation with human)</td>
<td>Low (she just needs to respond to very concrete questions)</td>
<td>Moderate (human intervention simplifies the process)</td>
<td>High (some NFR-related knowledge may be reused and may grow)</td>
<td>High (some new knowledge needs to be modeled)</td>
<td>High (answers to questions may be recorded)</td>
<td>High (functionality at PIM level; changes on NFRs require new questions)</td>
</tr>
</tbody>
</table>

VI. DISCUSSION AND RESEARCH AGENDA

Putting a NFR-aware MDD production process into practice looks like a great challenge. In this section we outline the most relevant issues to investigate with emphasis on requirements-related issues.

1) Modelling of NFRs at the PIM-level. (a) Which types of NFRs are most relevant to the MDD process? It is important to devote efforts to the NFRs that software architects perceive as the most important. Surveys (e.g., [39]) and interviews are needed. (b) Which notation to use for representing NFRs? As commented, quantitative and
qualitative approaches are the two (non-exclusive) big families. This is an old research topic in Requirements Engineering (already appearing in the 2000’s roadmap [40]) and results obtained in contexts other than MDD may be transferred here. (c) How NFRs may be linked to the functional elements? Some approaches have been cited [14][17][18] at this respect.

2) Elicitation and representation of architectural knowledge. (a) Which types of architectural knowledge exist and how are they used in practice? Again empirical studies are needed to give light to this question [41]. (b) Which are the quality attributes corresponding to these styles? (c) Which are the matching rules that allow determining the architectural solution that best fits the required NFRs?

3) Nature of models. The classification of MDD models into CIM, PIM and PSM as defined in the MDA approach results in some rigidity in our context. We have already defined the architectural model as an intermediate PIM/PSM model. The situation may even be more confusing if we inject the concept of architectural view [9] into the core of the MDD process. For instance, we may envisage that the evolution from the PIM down to the architectural models yields to a sequence of models in decreasing abstraction level, corresponding to the different architectural views, from the logical view down to the physical view. In this case, labelling the models may be artificial. We remark too that current MDD approaches focus on the architectural logical view, thus addressing other views is a progress by itself.

4) M2M transformations. Challenges are: (a) Gradually developing and incorporating in the framework transformations for all popular architectural styles. (b) Selecting the best alternative for each non-deterministic transformation depending on the expected NFRs. (c) Defining a transformation composition language for gluing separate transformations into the MDD models. This last point is highly connected with the vision promoted in [24][27] where different types of NFR are handled separately. Being true that the specificities of each NFR type makes it difficult to treat them uniformly, it is also clear that we need to be able to reconcile them since the generated system needs to fulfill all of them together. (d) The framework presented here conceives the application of transformation (and thus obtention of models) top-down with respect to abstraction level. However, this does not need to be always this way. For instance, a technological NFR fixing the brand and release of the data base product will have an implicit consequence on some other more abstract model, namely to know that a data base of a particular type (relational, OOR, ...) has to be integrated into e.g. the development view of the architecture. (e) How can they be integrated. In some sense, we may say that different architectural styles use different ontologies, e.g. whilst SOA talks about services, choreography and MOM, layered architectures introduce layers, persistence and push communication model. Incorporating this concept into the framework has consequences on its very core. If the M2M translation from PIM to PIM/PSM renders a software architecture, it follows that each architectural style requires a different metamodel, thus both PIM/PSM models and M2M transformations are dependant on the architectural style, becoming families of models and functions:

\[ (M2M_{arch(st)}: PIM(f^i \text{nf}) \rightarrow PIM/PSM_{(st)}(f^i \text{nf0}) \text{ has style} ) \]

Determining the architectural style should be the first decision to be made in the MDD process. Adopting a pure MDD perspective, it should be determined from the PIM(f^i \text{nf}). However, it is true that the decision of whether it must be, for example, an SOA or a Web rich client architecture is often a decision made before the MDD process starts for reasons that are not always tangible and are only in the architect’s mind.

7) Correctness and completeness issues. Last but not least, we mention the need of accurately investigating the notion of correctness of an NFR-aware approach. We may envisage the following conditions that need to be refined to the chosen formalisms. A couple of examples of predicates to investigate are:

- The NFRs should be correct both independently (e.g., there are not contradictory NFRs) and when referred to the functionality \( f \) (each functional element is qualified by meaningful types of NFRs): \( \text{correct(nf)} \land \text{applicable(nf, f)} \)
- The knowledge embedded in the MDD knowledge base should find feasible alternatives for any given NFRs that fulfill the correctness and applicability conditions above: \( \text{correct(nf)} \land \text{applicable(nf, f)} \Rightarrow \text{reducible(KB, nf)} \)

VII. CONCLUSIONS

In this vision paper we have: explored the state of the art; envisaged some generic solution to the identified problems; and enumerated new lines of research and challenges to overcome; or one requirement-related practice, the management of non-functional requirements (NFR) in the model-driven development (MDD) production paradigm.
Being this a vision paper, the main goal has been to agree on a perspective of the current state of the addressed problem and in the need to keep progressing towards several directions. Concerning the state of the art:

- We have analysed how MDD methods not dealing with NFRs behave to ensure their satisfaction.
- We have run a systematic literature review to learn insights of the MDD methods that deal with NFRs.

Concerning the improvement of this state of the art:

- We have formulated an MDD-aware general framework which allows customization to different settings with their own peculiarities.
- We have discussed variations on this framework.
- We have aligned and thoroughly compared the different alternatives discovered, trying to make clear not just the benefits but also the obstacles of this general framework.

From these obstacles, we have formulated a research agenda with the hottest open issues.

All in all, this paper agrees with the observation in [44]: “...MDD has a chance to succeed in the realm of large, distributed, industrial software development, but it is far from a sure bet”. We hope that this paper contributes to boost MDD has a chance to succeed in the realm of large, distributed, industrial software development, but it is far from a sure bet”. We hope that this paper contributes to boost MDD’s chances to succeed in such environments. We have discussed variations on this framework.

References