Automatic Selection of RIA Software Architectures using Quality Models

Santiago Meliá
IWAD Research Group
Dep. of Software and Comput. Syst.,
University of Alicante, Spain
Email: santi@dlsi.ua.es

Jesús Pardillo
Lucentia Research Group
Dep. of Software and Comput. Syst.,
University of Alicante, Spain
Email: jesuspv@dlsi.ua.es

Cristina Cachero
Dep. of Software and Comput. Syst.,
University of Alicante, Spain
Email: ccachero@dlsi.ua.es

***Abstract*—Rich Internet Applications (RIA) involve complex architectural design decisions, which greatly affect the quality of the final application. Unfortunately, quality requirements tend to be ignored during the architectural design process. This article reports a quality-driven approach for the systematic construction of RIA architectures. Our approach integrates model-driven development and software product lines principles to select and derive in a semiautomatic way the most suitable RIA architecture based on a set of predefined quality requirements. To show the feasibility of this approach, the method has been applied to a real scenario.**

I. Introduction

The complexity of Internet architectures and technologies has increased drastically in last years with the arrival of a new type of desktop-like Web applications, called Rich Internet Applications (RIAs). RIAs have complex distributed n-tiered architectures whose design greatly influences the quality of the final application. Given the fact that low levels of quality are at the root of a low customer satisfaction [1], RIA architectures should be always designed so that they contribute to optimise the user quality requirements.

However, most software engineering methodologies centre on functional aspects, to the detriment of quality characteristics. This non-functional side of architectures is regarded as a technical issue whose treatment, usually relegated until late phases of the project, relies on the software architect experience, instead of systematised knowledge and procedures. This often leads to situations when suboptimal solutions are provided, since, even when architectural-related quality flaws are identified during the testing workflow, making architectural changes at late stages of development is usually unfeasible [2].

The solution to this situation is to design for quality, and not only for functional requirements. Experience shows that designing for quality often requires the use of certain design patterns or styles. For instance, to improve portability and modifiability it may be beneficial to use a layered architecture style [3]. For this reason, in the last years, a myriad of architectural patterns and best practices have been proposed [4], [5], and their impact on the application quality has been widely discussed.

Unfortunately, the application of architectural patterns in the RIA domain is not always straightforward. RIA components usually affect several quality characteristics, and making decisions to improve one of them may inadvertedly hamper another. Also, the knowledge about how RIA components affect quality is usually scattered around, and its acquisition often requires years of experience.

In this paper we propose a systematic approach to the definition and semiautomatic generation of RIA interface architectures that takes into account quality aspects from the very first stages of development. We believe that this proposal is specially valuable for novel architects to understand the impact of their architectural decisions on quality aspects of RIA applications.

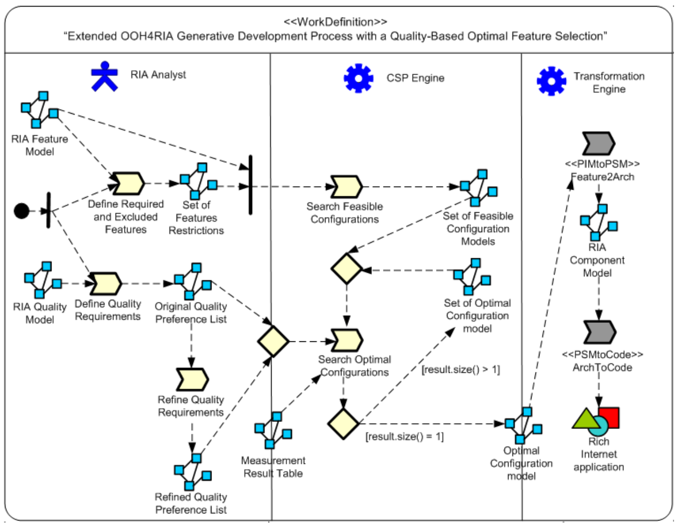
The sequel of the paper presents our solution as follows. We first provide an overview of the architectural design process that vertebrates our approach (§II). Next, Section III details the RIA domain design artefacts involved and Section IV formalises the algorithmic selection of the best-fitted RIA architectural features. Finally, the application of our development process to a real scenario is reported (§V), the related work is reviewed (§VI), and the discussion presented (§VII).

II. The OOH4RIA Architectural Design Process

The purpose of the OOH4RIA architectural design process is to systematise the selection of RIA architectures that are consistent with a set of predefined quality requirements.

For this purpose, we have extended a RIA Generative Software Development (GSD) method called OOH4RIA [6]. OOH4RIA provides a *RIA feature model* that represents the architecture and technological variability of the RIAs family. In this approach, the feature selection process is performed manually, which causes that the quality of the final RIA depends on the expertise of the software architect. In order to provide an automated assistance to this process, in this work we have enriched the approach by defining:

- a *RIA quality model* that represents the set of ISO quality requirements that may be affected by the feature selection,
- a *measurement result table* which establishes a correspondence between the selected quality attributes and the RIA architectural and technological features,
- an enriched *RIA architectural design process* that, based on a Constraint Satisfaction Problem (CSP) formulation,



<<WorkDefinition>>
"Extended OOH4RIA Generative Development Process with a Quality-Based Optimal Feature Selection"
RIA Analyst
CSP Engine
Transformation Engine
RIA Feature Model
Define Required and Excluded Features
Set of Features Restrictions
Search Feasible Configurations
Set of Feasible Configuration Models
RIA Quality Model
Define Quality Requirements
Original Quality Preference List
Set of Optimal Configuration model
Search Optimal Configurations
[result.size() > 1]
[result.size() = 1]
Refine Quality Requirements
Refined Quality Preference List
Measurement Result Table
Optimal Configuration model
<<PIMtoPSM>> Feature2Arch
RIA Component Model
<<PSMtoCode>> ArchToCode
Rich Internet application

Figure 1. Proposed RIA architectural design process and artefacts

automatises the selection of best suited feature configurations according to the proposed quality model.

These main contributions will be further discussed in Section III. In particular, the role that these artefacts play in the RIA architectural design process can be seen in Fig. 1. In this figure, we can observe how the *RIA analyst* departs from the above-mentioned *RIA feature model* to *define the required and excluded features* for the particular application. The result of this activity is a *set of feature restrictions* application artefact. The problem of finding feasible solutions that satisfy a set of restrictions can be mathematically formalised as a CSP. This way of representing the problem allows a *CSP engine* to *search for feasible configurations* through the solution space.

The output of this activity is a *set of feasible configuration models* that is passed on to a *search optimal configurations* activity, also performed by the *CSP engine*. For this activity to be performed, the *CSP engine* needs a *quality preference list*. This list is constructed by the *RIA analyst* by *defining the quality requirements* that apply to the application under construction. For this activity, the *RIA analyst* needs to select and prioritise a subset of quality subcharacteristics from the RIA quality model. Please note how the objective of this process is not to get specific values for specific quality characteristics, but to optimise such values while respecting the architectural and technological restrictions of the application.

When both the *quality preference list* and the *set of feasible configuration models* are available, it is possible to *search for optimal configurations*. The result of this activity is a *list of optimal configurations*, that may have any number of optimal solutions. However, in order to proceed with the semiautomatic generation process defined by OOH4RIA, we need a single solution (depicted in Fig. 1 as the *optimal configuration model* artefact). For this reason, the process includes a *refine quality requirements* activity, that is also performed by the *RIA analyst* in order to iteratively refine the set of quality requirements until the CSP finds a single configuration that optimises them. Alternatively, the *RIA analyst* may, at any time, manually select the *optimal configuration model* out of the *set of optimal configuration models*. This manual selection has been left out of the process for the sake of simplicity.

Once the *optimal configuration model* has been selected, the OOH4RIA approach includes a *transformation engine*. This engine executes a *Feature2Arch* model-to-model transformation to obtain a *RIA component model*, and a *Arch2Code* model-to-text transformation to finally obtain the *rich Internet application* implementation. Interested readers in this part of the process are referred to [6].

III. OOH4RIA Design Artefacts

As we have aforementioned, the OOH4RIA architectural design process heavily relies on three artefacts: a RIA feature model, a RIA quality model and a RIA measurement result table. These artefacts are discussed next.

A. Feature Model

The RIA feature model is an artefact that represents the variability in the RIA solution space, *i.e.*, architecture and technology, so that the development risks are reduced. It is worth highlighting that this model does not try to be exhaustive on any RIA option available. Instead, it focuses on the alternatives that are available inside the RIA family. The feature model succinctly describes the range of applications that can be derived. Indeed, the very same functional description (*i.e.*, domain, navigation, presentation models) can be mapped into different RIAs based on the features being selected. The set of features being selected for a given applications is known as the Configuration Model. This feature model is formalised by a MOF metamodel that permits to associate a valid Configuration model with a RIA Component model, establishing a complete refinement process whose final outcome is a RIA implementation [6].

Fig. 2 represents the client side of the OOH4RIA feature model using the Czarnecki extended notation [7]. The included architectural features may point either to the components that make up these layers (*e.g.*, the client layer may contain three component types: *UI component*, *UI process component* and *UI entity data*) or to features that can affect the whole layer (*e.g.*, the particular RIA *platform*). Components are related to their specific design features. For instance, a *UI component* can have *validation*, *templating* and/or *I18N* features. Similarly, a *UI process component* can be linked to the *event handling* feature *via* either the *observer* or the *event bus* pattern.

B. Quality Model

The *RIA quality model* is a hierarchical structure that defines a set of quality characteristics and subcharacteristics that can be influenced by RIA architectural and technological decisions. For its definition, we have tailored the well-known ISO/IEC 9126-1 standard [8]. The reason behind this decision is twofold. On one hand, this framework is an international standard, widely supported by both researchers and practitioners. On the other hand, its specification explicitly allows the adaptation (if necessary) of the framework to different needs.

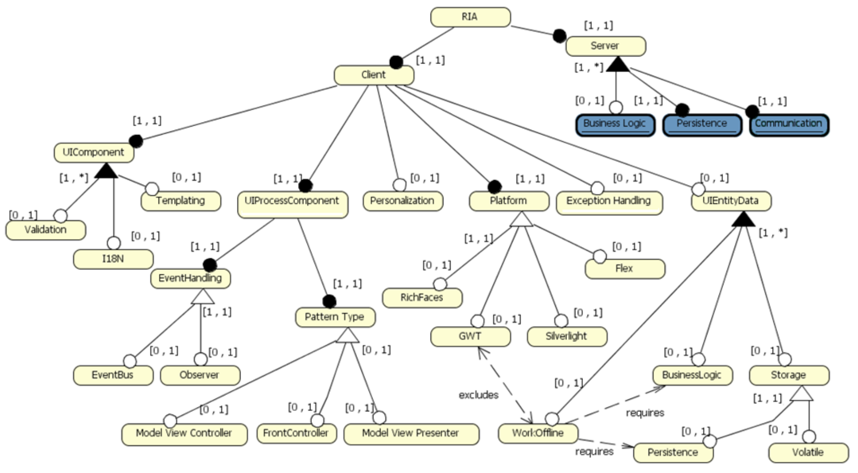


Figure 2. RIA feature model (adapted from [6]).

For the adaptation of the ISO/IEC 9126 quality model to the idiosyncrasy of RIA architectures, we have studied each one of its six quality characteristics (*i.e.*, functionality, reliability, usability, efficiency, maintainability and portability) independently. For each characteristic, we have studied RIA architecture configurations that could affect such characteristic. The result of this study can be seen in Table I. In this table, it can be observed how, according to our model, all the ISO/IEC quality characteristics except for fault tolerance (treated in RIA configurations on the server side) can be affected by RIA interface architectural decisions.

The RIA quality model is used by RIA analysts to define the *quality preference list* artefact of Fig. 1.

C. Measurement Result Table

The *measurement result table* serves to connect the RIA quality model and the RIA feature model. This table is presented in Table I. For the sake of simplicity only the client-side features of the RIA feature model are presented. For these, the table defines how each leaf feature impacts each RIA quality subcharacteristic. The measure of the impact that each feature has on each quality characteristic is an ordinal measure whose value can be -2 (affects very negatively, '$\underline{\times}$'), -1 (affects negatively, '$\times$'), 0 (neutral), +1 (affects positively, '$\checkmark$'), and +2 (affects very positively, '$\underline{\checkmark}$').

Depending on the feature type, the measure value has been obtained differently. In some cases, these values are derived from the architectural and design patterns forces. For example, the main force of the Fowler's supervised controller pattern [4] (which we have called model-view-presenter, MVP) is the improvement of the UI testability. We have thus assigned a value of +1 ($\checkmark$) on the testability of the MVP configuration.

On other occasions, these values are based on Best Practices of specific RIA platforms. For example, according to [5], data validation is critical to the security of a RIA application. For this reason, we have assigned a value of +1 ($\checkmark$) on the security of the validation feature. Finally, the impact of technological features such as GWT, Flex, etc. on the quality of the application has been measured following the recommendation of RIA experts. For instance, the browser-based RIA platforms (*e.g.*, GWT and RichFaces) have a better installability than the plugin-oriented RIAs platforms (*e.g.*, Flex, Silverlight, etc.) because they do not need to install any plug-in in the client. This is reflected in the values of installability for GWT and RichFaces, both set to +1 ($\checkmark$).

Values of +2/-2 ($\underline{\checkmark}$, $\underline{\times}$) are reserved for occasions when two alternative features both contribute in the same sense (positively or negatively) to a given quality subcharacteristic, but one alternative contributes to a significantly greater extent. For example, the MVP improves security, but the Front Controller improves security to a greater extent. The reason is that this pattern reduces the security holes by proposing a single controller that handles all requests made to the presentation-tier which permits to apply the same security policy in each request, while the MVC and MVP patterns propose multiple controllers where the security policy can be inconsistently applied and can lead to security breaches [9].

At this point, it is important to note that, since our objective is to order architectural alternatives (and not provide objective quality values), this way of assigning measures suffices. The provision of objective measures, which would provide much richer information for the global evaluation of the quality of the resulting RIA architecture, will be discussed in Section VII.

D. Feature Restrictions & Quality Preferences

Apart from the three artefacts presented so far, our approach includes two application-specific artefacts that are of paramount importance for the characterisation of our problem as a CSP.

On the one hand, both the application deployment context and the customer preferences may impose some restrictions on the set of features that the application architecture must contain and/or avoid. As we have aforementioned, this is specified by means of a *set of feature restrictions* artefact. In addition, RIA analysts need to define and prioritise the set

Table I
MEASUREMENT RESULT TABLE (CLIENT FEATURES ONLY)

	Function.:				Reli.:		Usabi.:				Ef.:		Mainta.:				Portabi.:			
	Suitability	Accuracy	Interoperability	Security	Maturity	Recoverability	Understandability	Learnability	Operability	Attractiveness	Time behaviour	Resource utilization	Analysability	Changeability	Stability	Testability	Adaptability	Installability	Co-Existence	Replaceability
Exception Handling						✓							✓		✓	✓				
Personalisation								×		✓							✓			
UIComponent:																				
I18N							✓													
Templating								✓		✓				✓		✓				✓
Validation	✓	✓		✓	✓				✓		×					✓				
UIProcessComponent, Event Handling:																				
Event Bus																				✓
Observer														✓						
UIProcessComponent, Type:																				
Front-Controller				$\underline{\checkmark}$										✓			✓			✓
MVC																				
MVP				✓												✓				
Platform:																				
Flex			✓							✓										
GWT											✓	✓				✓		✓	✓	
RichFaces												✓				✓		✓	✓	
Silverlight			✓					✓		✓										
UIEntityData:																				
Business Logic									✓		$\underline{\checkmark}$			×						
Work-offline						✓			$\underline{\checkmark}$		$\underline{\checkmark}$			×						
Persistence (Storage)						✓														
Volatile (Storage)											✓									

of quality characteristics that should drive the construction of the application. As can be seen in Fig. 1, this is specified by means of a *quality preference list* artefact.

The formal description of these two artefacts is as follows:

Definition 1 (Feature Restrictions): Let F be the set of modelled features. Then, *feature restrictions* over F are the members of a partition of $F = R \cup E \cup F'$ so that:

- R contains the *required* (must-have) features,
- E contains the *excluded* features, and
- F' contains the *free features* that are not in $R \cup E$.

Definition 2 (Quality Preferences): Let Q be the set of modelled quality characteristics. Then, a *quality preference* over Q is a total order $\langle \leq, Q' \rangle : Q' \subseteq Q$. The tuple $\mathbf{q}$ so that $q_{i+1} \leq q_i$ is named *quality preference list*.

IV. RIA ARCHITECTURAL CONFIGURATIONS

As we have aforementioned (see Fig. 1), RIA feasible architectural configurations are collections of features that comply with (i) the feature composition cardinalities (defined in the RIA feature model) and (ii) the set of feature restrictions (defined in the feature restriction artefact). These configurations are obtained by solving the associated CSP problem.

In our approach, the CSP has been defined as the tuple $\langle X, D, C \rangle$ so that: X (variables) is the set of the *optional* features that are described in Table II, D (domains) is the Boolean set, and C (constraints) is the set of constraints derived from the feature composition cardinalities of Fig. 2 [10]. Table II lists the different CSP parts. For space reasons, this table contains only three out of the ten restrictions that are derived from the feature composition cardinalities.

Table II
CSP VARIABLES AND CONSTRAINTS (LISTED BY DEPTH-FIRST TRAVERSAL OF FIG. 2)

CSP Constants: Mandatory Features
`EventHandling`, `PatternType`, `Platform`, `UIComponent`, `UIProcessComponent`

CSP Variables: Optional Features
`BusinessLogic`, `EventBus`, `ExceptionHandling`, `Flex`, `FrontController`, `GWT`, `I18N`, `MVC`, `MVP`, `Observer`, `Persistence`, `Personalization`, `RichFaces`, `Silverlight`, `Storage`, `Templating`, `UIEntityData`, `Validation`, `Volatile`, `WorkOffline`

CSP Constraints: Feature Composition Cardinalities
$\mathtt{Storage}(\mathtt{Persistence}, \mathtt{Storage}, \mathtt{Volatile}) \equiv$
$\quad \mathtt{Storage} \wedge (\mathtt{Persistence} \veebar \mathtt{Volatile})$
$\mathtt{excludes}(\mathtt{GWT}, \mathtt{WorkOffline}) \equiv$
$\quad (\mathtt{GWT} \veebar \mathtt{WorkOffline}) \vee \neg(\mathtt{GWT} \vee \mathtt{WorkOffline})$
$\mathtt{requires}_1(\mathtt{BusinessLogic}, \mathtt{WorkOffline}) \equiv$
$\quad \mathtt{WorkOffline} \Rightarrow \mathtt{BusinessLogic}$
...

CSP Constraints: Feature Restrictions	
$R_f(f) \equiv f \in R$, given $f \in F$	$E_f(f) \equiv f \in E$, given $f \in F$

In order to obtain the *optimal* configuration, that is, the configuration that maximises the values of the quality preferences, the set of feasible configurations (denoted as C) obtained from this CSP need to be ranked and sorted accordingly. The corresponding objective function is based on these definitions:

Definition 3 (Configuration q-quality): Let q be a quality characteristic in the quality model and $\mathbf{c} \in C$ a feasible configuration with n features. Then, the configuration q-quality is the natural number $Q(\mathbf{c}; q) \equiv \sum_{i=1 \ldots n} M(c_i, q)$, where $M : F \times Q \rightarrow \{-2, -1, 0, +1, +2\}$ is a function that models the measurement result table (Table I), given the feature and quality models F, Q, and results $+2$ for '$\underline{\checkmark}$', $+1$ for '$\checkmark$', etc.

Definition 4 (Global configuration quality): Let $\mathbf{q}$ be a quality preference list. Then, the (global) configuration quality is the total order relation $\langle \geq_{\mathbf{q}}, C \rangle$ so that:

$$\begin{aligned} \mathbf{c} \geq_{\mathbf{q}} \mathbf{d} &\equiv Q(\mathbf{c}; q_1) \geq Q(\mathbf{d}; q_1) \ \vee \ \mathbf{c} \geq_{\langle q_2, \ldots, q_n \rangle} \mathbf{d} \\ \mathbf{c} \geq_{\langle q \rangle} \mathbf{d} &\equiv Q(\mathbf{v}; q) \geq Q(\mathbf{w}; q) \end{aligned}$$

With these two definitions, the optimal feasible configuration is the greatest element of C according to $\leq_{\mathbf{q}}$. Since it may happen that it is not unique (in particular for short quality preference lists), our development process (see Fig. 1) includes the refinement of the quality preferences (*e.g.*, by appending new characteristics).

V. Validation

In order to validate our approach, we have implemented the CSP in Prolog[1], and we have tested it against a real project, namely, a Content Management System (CMS) development for the Alicante tax agency (SUMA)[2]. The SUMA CMS is an intranet RIA which permits to introduce different multimedia contents (*i.e.*, documents, files, images, videos, etc.) and to create publication points for SUMA Web applications.

SUMA demands a highly testable user interface that allows him to test all changes quickly (automatically if it is possible). Moreover, they want to be able to easily replace the UI layout and widgets while, at the same time, keeping a consistent look & feel and UI behaviour (events sent or received from/to widgets). On the other hand, the personal application data (*i.e.*, taxes, fines, etc.) requires protecting this information so that unauthorised persons or systems cannot read or modify them.

In order to apply our approach, this specification has been translated into the next setting: $R_F = \{\text{I18N}\}, B_F = \emptyset$ with the quality preference list set to $\langle$testability, replaceability, security$\rangle$. These settings make up the input of our development process, which has been iterated as shown in Table III in order to obtain a unique optimal configuration.

On each iteration (I), the system indicates the cardinality of the set of candidate configurations (O), together with a suggestion of candidate quality characteristics that may further narrow this set. In this concrete example, Table III shows how the initial number of feasible configurations is 640. The set of initial quality requirements restrict the number of optimal configurations to ten.

Since SUMA is also interested in a highly changeable user interface, we have added changeability (ΔR_F) as a forth

[1]Code available in 'http://www.lucentia.es/lib/2010/quatic/'

[2]http://suma2.dlsi.ua.es/ooh4ria/Projects.html

Table III
TRACE OF THE QUALITY PREFERENCE LIST REFINEMENT

I	ΔR_F	O	Candidate Characteristics
Content Management System (640 feasible configurations):			
0	$\emptyset$	10	recoverability, operability, time behaviour
1	changeability	4	recoverability, learnability, attractiveness, time behaviour, adaptability
2	operability	4	recoverability, learnability, attractiveness, time behaviour, adaptability
3	time behaviour	2	learnability, attractiveness, adaptability
4	learnability	1	N/A

requirement, which further restricts the number of optimal configurations to four. Notice that the iteration process is performed with new quality characteristics, selected by the RIA analyst, who may consult the client for further clarifications on his preferences.

The process stops when (i) a unique optimal configuration is found or (ii) when the RIA analyst manually selects a configuration among those offered by the system. In this particular case, we have iterated the process with the addition of operability, time behaviour and learnability, until the CSP has thrown a single optimal configuration. Such optimal configuration contains the following features: event bus, exception handling, I18N, MVP, RichFaces, storage, templating, UI entity data, validation, volatile.

An in-deep analysis on the solution shows its accuracy: the SUMA CMS configuration model defines a RIA UI architecture mainly focused on testability, which promotes the selection of the MVP pattern, a browser-oriented RIA platform like RichFaces and an exception handling management system. Moreover, its necessity of replaceability fixes the templating mechanism to structure the UI static side and the event bus to define the dynamic side. Finally, the required UI security is achieved by the validation feature, and the UI time behaviour is improved using volatile storage.

VI. Related Work

The approach presented in this paper is based on research and practices from several disciplines, namely Web engineering, requirements engineering, feature modelling and constraint programming. For this reason, this section compares our work with respect to them, and outlines both the similarities and differences of our proposal.

The Web engineering research community has for some time provided Web design methodologies to support RIA development demands (*e.g.*, WebML [11]). The existing approaches have proven successful to deal with RIA functional concerns. However, non functional aspects have been traditionally ignored. Two noteworthy exceptions to this situation are RUX [12] and OOH4RIA [6]. RUX discusses patterns and guidelines for the design of RIA architectures. OOH4RIA has gone one step further and proposes a set of functional concerns (domain, navigation, presentation and orchestration models) which are accompanied by an early representation of the non functional RIA variability (*i.e.*, RIA feature model) and with an RIA-specific architectural style, called RIA component model, that establishes a component configuration in the solution space. Our proposal extends OOH4RIA with a more complete non-functional treatment that is used to guide the OOH4RIA development process.

The reason for this extension is that it is a proved fact that the appropriate management of requirements is one of the most influential factors in the success of software development projects [13]. Although functional requirements seem to be well integrated in existing Web engineering methodologies, the situation regarding non functional requirements is quite different [14]. Among the many artefacts proposed by the Requirements Engineering discipline to deal with non-functional requirements, quality models outstands to the point that there are several ISO standards devoted to the definition of quality models for different domains and perspectives [8], [15]. Our contribution in this sense is the extension of OOH4RIA with a RIA quality model that provides non functional requirements support and tailors the ISO 9126 standard to the RIA context. Also, we have related architectural and technical decisions to this RIA quality model.

For the definition of RIA variability, we have based our approach on works defined in the product-line engineering discipline that justify taking into account software architectural and technological variability using feature modelling (see, *e.g.*, RSEB [16] or [7]). In particular, the work of Czarnecki [7] provides the foundations for introducing the feature model into the OOH4RIA model-driven development process. Our contribution with respect to these approaches is the explicit capture of the non functional requirements and the definition of the impact that different architectural and design decisions have on these requirements.

Last but not least, semantics of feature models have been already modelled as CSP by other authors. In particular, [10] aligned feature models with CSP and also took into account the optimisation problem associated to the raking of the valid configurations. Many of the terms used herein can be thus referred against the definitions provided in [10].

VII. Discussion

The most outstanding characteristics of the RIA development process presented in this work are: (i) it is iterative, which enables designers to gain insight incrementally on which features should be selected; (ii) it makes the relationships between non functional requirements and features explicit, which enables designers to understand which features are better than others with regard to the quality preferences, (iii) it permits to store the rationale behind the architectural decisions taken, (iv) it is supported by a well-founded formalisation (CSP) for the feature selection, (v) all its input artefacts are low-coupled, in order to facilitate the refinement of the input artefacts (adding new features, refining the impact that features have on requirements, etc.) as more knowledge becomes available, and (vi) it allows for an automatic code generation of the RIA application.

However, there are still some points to improve. The particular level of abstraction of the feature model makes that some knowledge contained in architectural patterns defined at higher or lower levels of abstraction has been left aside (*e.g.*, patterns related to architectural layers). Also, we have assumed that each feature impacts quality characteristics independently; however, it is also possible that groups of features, instead of features in isolation, could determine quality characteristics. In order to overcome this issue, the measurement results table can be empirically refined by automatic measures over the implemented architectures and statistical processes that serve to empirically determine the impact each feature has on each quality attribute. Last but not least, concerning the usability of our development process, it should be supported by proper design artefacts and interaction models. For instance, an appropriate diagrammatic notation for managing the feasible configurations can help to boost the refinement process (see Fig. 1) by improving its readability. All these aspects outstand as future lines of research.

References

[1] L. Chung and J. C. S. do Prado Leite, "On Non-Functional Requirements in Software Engineering," in *Conceptual Modeling: Foundations and Applications*, 2009, pp. 363–379.

[2] P. Clements and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] E. Folmer and J. Bosch, "Architecting for usability: a survey," *Journal of Systems and Software*, vol. 70, pp. 61–78, March 2004.

[4] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] J. Meier, J. A. Homer, D. Hill, J. Taylor, P. Bansode, L. Wall, R. B. Jr, and A. Bogawat, "Microsoft Patterns Practices: Rich Internet Application Architecture Guide," Microsoft Corporation, Tech. Rep., 2009.

[6] S. Meliá, J. Gómez, S. Pérez, and O. Díaz, "Architectural and Technological Variability in Rich Internet Applications," *IEEE Internet Computing*, vol. In Press, June 2010.

[7] K. Czarnecki, "Generative Software Development," in *SPLC*, 2004, p. 321.

[8] International Organization for Standardization, "ISO/IEC 9126-1:2001 Software engineering –Product quality– Part 1: Quality model," http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749, 2001.

[9] D. Trowbridge and D. Mancini, "Enterprise Solution Patterns Using Microsoft .NET. Patterns and Practices. Version 2.0." Microsoft Corporation, Tech. Rep., 2003.

[10] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés, "Automated Reasoning on Feature Models," in *CAiSE*, 2005, pp. 491–503.

[11] A. Bozzon, S. Comai, P. Fraternali, and G. T. Carughi, "Conceptual modeling and code generation for rich internet applications," in *ICWE '06: Proceedings of the 6th international conference on Web engineering*. New York, NY, USA: ACM, 2006, pp. 353–360.

[12] M. L. Trigueros, J. C. Preciado, and F. Sánchez-Figueroa, "Engineering rich internet application user interfaces over legacy web models," *IEEE Internet Computing*, vol. 11, no. 6, pp. 53–59, 2007.

[13] M. Epner, "Poor Project Management Number-One Problem of Outsourced E-Projects," *Research Briefs, Cutter Consortium*, November 2000.

[14] M. Lang and B. Fitzgerald, "Web-based systems design: a study of contemporary practices and an explanatory framework based on method-in-action," *Requirements Engineering Journal*, vol. 12, no. 4, pp. 203–220, 2007.

[15] International Organization for Standardization, "ISO/IEC 25000:2005 Software Engineering –Software product Quality Requirements and Evaluation (SQuaRE)– Guide to SQuaRE," http://www.iso.org/iso/catalogue_detail.htm?csnumber=35683, 2005.

[16] M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," in *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*. Washington, DC, USA: IEEE Computer Society, 1998, p. 76.