Towards the Dynamic Reconfiguration of Quality Attributes

Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes
Universidad de Málaga, Andalucía Tech, Spain
{horcas,pinto,lff}@lcc.uma.es

Abstract

There are some Quality Attributes (QAs) whose variability is addressed through functional variability in the software architecture. Separately modelling the variability of these QAs from the variability of the base functionality of the application has many advantages (e.g., a better reusability), and facilitates the reconfiguration of the QA variants at runtime. Many factors may vary the QA functionality: variations in the user preferences and usage needs; variations in the non-functional QAs; variations in resources, hardware, or even in the functionality of the base application, that directly affect the product's QAs. In this paper, we aim to elicit the relationships and dependencies between the functionalities required to satisfy the QAs and all those factors that can provoke a reconfiguration of the software architecture at runtime. We follow an approach in which the variability of the QAs is modelled separately from the base application functionality, and propose a dynamic approach to reconfigure the software architecture based on those reconfiguration criteria.

Keywords Quality attributes, reconfiguration, software architecture, SPL, variability

1. Introduction

There are some quality attributes (QAs) that have functional implications in the software architecture of the applications affected by them (Juristo et al. 2007). For instance, adding security to an application implies adding the components that provide the access control methods or encryption algorithms. The variability of these QAs is addressed through the incorporation of some QA-specific functional variability inside the application's software architecture. For instance, the variability of the availability and the security QAs are achieved by having different graphical representations for desktop and mobile environments and by having different access control methods and encryption algorithms, respectively. In this paper we distinguish between (1) the base functionality of the applications;(2) the additional functionality required to satisfy some QAs such as security, usability, or persistence; and (3) those QAs, tradition-ally understood as non-functional properties, such as such as performance, efficiency, or cost, that can be mapped to architectural or implementation decisions, but not directly to functional components.

The additional functionality required to satisfy some QAs (e.g., security) can be composed by many components (e.g., authentication, encryption, or integrity), and different applications may require a customized subset of each functionality (e.g., only encryption). Also, different QAs may have dependencies between them which should be taken into account during architecture elicitation. For instance, the contextual help concern of the usability QA depends on the authentication concern of the security QA to be able to offer customized help based on the user's previous experience with a given application. The high degree of variability exhibited by the QAs exposes the real complexity of managing the variants of the QAs. So, the use of a Software Product Line (SPL) (Metzger and Pohl 2014) to model and manage the variability of the QAs makes sense. However, although SPLs are traditionally used to exploit and handle the commonalities and variations of the base functionality of the applications, QAs variability has not received as much attention (Etxeberria et al. 2007), and there is a lack of existing approaches that integrate QA variability as a part of the systematic variability management of SPLs (Myllärniemi et al. 2012). In this paper, we follow our previous approach (Horcas et al. 2014a) to model the variability of the QAs separately from the base application variability. Separately modelling the variability of the QAs from the variability of the base application has many advantages such as a better reusability of both the QAs and the applications.

Moreover, the functionality of the QAs can vary for several reasons, such as variations in the user and usage needs (e.g., different users may prefer a different level of usability based on their experience with the application), variations in hardware (e.g., fingerprint authentication only works with touch screen devices) and variations in the available resources (e.g., available space affects persistence mechanisms). Reconfiguring the functionality of the QAs may imply changes in many components at the architectural level. For QAs with functional implications, which can vary at runtime, we need to be able to modify the software architecture of the applications affected by them, by changing specific elements at runtime (components, interactions, classes, etc.). Moreover, we need to take into account the impact of all these variations over the product's non-functional QAs (e.g., performance, efficiency) (Myllärniemi et al. 2012).

Systematic literature reviews (Galster et al. 2014; Mahdavi-Hezavehi et al. 2013; Myllärniemi et al. 2012) on QAs and variability management evidence that existing research methods focus on how variations in the application functionality, or in the hardware resources, affect the quality of the overall software architecture in terms of non-functional QAs (e.g., efficiency), but little work is devoted to deal with the dynamic variants in the functionality required to realize some QAs (e.g., security, usability), which, as previously said, need to be introduced into the software architecture as functional components (Horcas et al. 2014a). So, in this paper, we aim to elicit and model the relationships and dependencies between the functionality of the QAs, and the factors that affect them at runtime and thus may provoke a reconfiguration of the QA functionality. These factors include user preferences, context changes, changes in the requirements of the base application, etc. Making explicit these relationships is indispensable to automate the process of reconfiguring the architecture at runtime, and facilitates the evaluation of the architecture configurations. We propose a self-adapting approach that automatically reacts to the changes on those criteria in order to reconfigure the functionality of the QAs at runtime. To do so, we extend our previous static approach (Horcas et al. 2014a) that separately models the variability of the QAs from the base application in order to customize and introduce the QA functionality into the software architecture of the applications at design time, and propose a dynamic approach to reconfigure the QA functionality. Our approach maintains, at runtime, the benefits of separating the QA functionalities from the base application.

The rest of the paper is organized as follows. In Section 2 we briefly overview the state of the art and discuss existing approaches for varying QA functionality, variability modelling of QAs, and reconfiguration of QAs. We also motivate our approach with a real case study. Section 3 presents our approach to dynamically reconfigure the QA functionality at runtime. In Section 4 we show how to separately model the variable functionality of the QAs and how to explicitly model their relationships with the non-functional QAs and the base application. In Section 5 we identify some open issues related to our approach. Finally, Section 6 expounds the conclusions and on-going work.

2. State of the Art and Motivation

There are several approaches to model variability of QAs such as extensions to feature models (Benavides et al. 2006), goal-based models (González-Baixauli et al. 2004), Bayesian belief networks (Zhang et al. 2003), or frameworks like COVAMOF (Sinnema et al. 2006). All these methods are compared in (Etxeberria et al. 2007), but none of them have became a de-facto approach for modelling the variability of QAs, nor do they deal with the reconfiguration of QAs at runtime. Moreover, recent literature reviews (Galster et al. 2014; Mahdavi-Hezavehi et al. 2013; Myllärniemi et al. 2012) expose the lack of mature research into QA variability.

Most of the existing approaches target QA variability from the point of view of the non-functional requirements, such as (Alam et al. 2013; Etxeberria et al. 2007, 2008; Myllärniemi et al. 2012; Zhang et al. 2010). For instance, Etxeberria et al. (Etxeberria et al. 2008) model QA variability with the purpose of evaluating how variations in the base application functionality vary the level of performance and usability. Moreover, they usually model the QA variability together with the variability of the base application, increasing the complexity of capturing and analysing the variability. Similarly, Zhang et al. (Zhang et al. 2010) estimate the impact of functional features on each QA by using the Analytic Hierarchical Process (AHP), while Alam et al. (Alam et al. 2013) model the variability of the interfaces of generic concerns and focus on the impact of the concerns on non-functional QAs through goals-based models.

However, there has been little work devoted to modelling the variability of those QAs that have strong functional implications and need to be modelled as functional components to be incorporated into the software architecture. For instance, Juristo et al. (Juristo et al. 2007) propose an approach in which usability features with major implications for software functionality (e.g., feedback, contextual help) are incorporated as functional requirements into the software architecture. Other static approaches that deal with QA functionalities at the architectural level are the QADA (Quality-driven Architecture Design and quality Analysis) method (Matinlassi et al. 2002) and the RiPLE-DE process (Cavalcanti et al. 2011). QADA (Matinlassi et al. 2002) is a specific method for designing SPL architectures by transforming systematic functionality and QAs into software architectures, but this proposal does not explicitly take into account the quality requirements. The RiPLE-DE (Cavalcanti et al. 2011) process is a domain design process for SPL that can be extended to model the QA variability as part of a family of products. The QA variability is represented in feature model diagrams and in order to achieve desired quality levels, the QAs are enhanced with information of the base application (e.g., the system's response measure). Thus, the variability of the QA functionalities directly depends on the base application, preventing the QA functionalities from being reused.

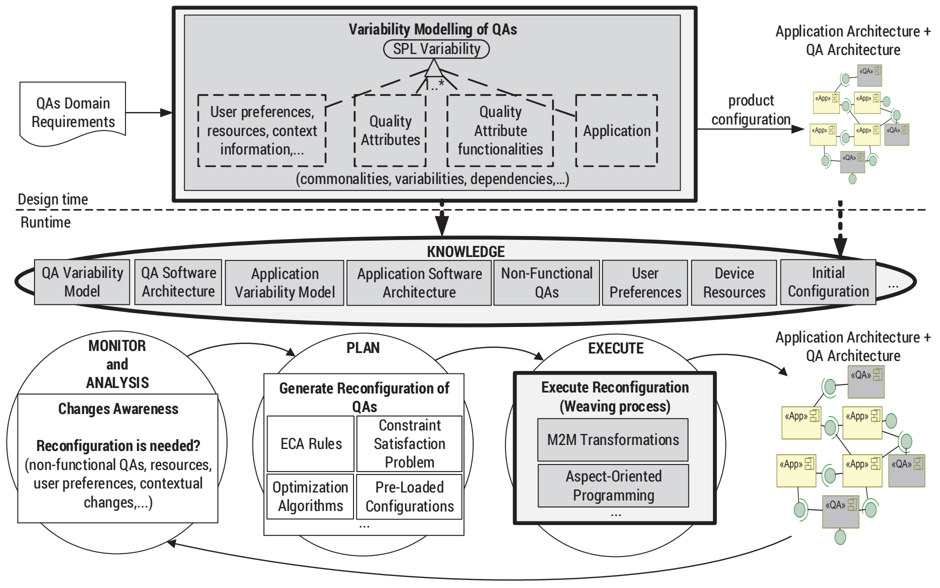
The conclusion is that there are few approaches that address the variability of the QA functionalities using SPLs, and those existing approaches do not consider that the QA functionalities that are independent from the application (e.g., authentication mechanisms, hashing and encryption algorithms, usability features, etc.) should not be modelled along side the functionality of the base application, but rather they should be modelled separately.

Following an approach similar to Juristo et al. (Juristo et al. 2007), our work aims to elicit similar functionalities for other QAs (e.g., security, persistence), not only usability. A difference to existing approaches is that we separately model the context information, the non-functional QAs, the variability of the QA functionalities and the variability of the base application. Moreover, we identify and explicitly define the dependencies with the non-functional properties (e.g., performance, level of security or usability, user preferences) with the purpose of reconfiguring the QA functionality at runtime based on those reconfiguration criteria. The main benefit is that in our approach the same QA functionalities can be reused for different applications and this facilitates the reconfiguration of those functionalities at runtime.

2.1 Motivating Case Study

In order to illustrate our approach, we consider the following case study: an electronic voting (e-voting) application[1] that allows creating and joining different kinds of elections (national, corporative, and social elections) and casting votes from different devices (e.g., smart phones, desktop, etc.). Apart from this basic functionality, some of the QA requirements that could be taken into account in the development of the e-voting application are: (1) all the clients must be authenticated before creating/joining an election, and (2) all the votes must be encrypted. So, an authentication mechanism and an encryption algorithm must be introduced into the software architecture of the e-voting application — i.e., authentication and encryption are the QA functionalities required to satisfy security. However, each kind of election requires a different level of security — i.e., a non-functional QA requirement. For instance, national elections require a higher level of security than social elections, and thus, this also affects the QA functionality because different authentication mechanisms and encryption algorithms must be used for each kind of election. Each kind of authentication mechanism provides a different level of usability for the user. For instance, identifying the user using a social network identifier (e.g., the user's Facebook profile) provides a high level of usability because it only requires a simple click by the user to log in to the application, but the level of security is lower than using a digital certificate authentication. Furthermore, each encryption algorithm has a different memory consumption and efficiency. When the battery of the smart phone is lower, the encryption algorithm needs to be reconfigured at runtime to use a lighter encryption algorithm.

[1] `http://www.inter-trust.eu/`



Variability Modelling of QAs
SPL Variability
QAs Domain Requirements
User preferences, resources, context information,...
Quality Attributes
Quality Attribute functionalities
Application
1..*
(commonalities, variabilities, dependencies,...)
product configuration
Application Architecture + QA Architecture
Design time
Runtime
KNOWLEDGE
QA Variability Model
QA Software Architecture
Application Variability Model
Application Software Architecture
Non-Functional QAs
User Preferences
Device Resources
Initial Configuration
...
MONITOR and ANALYSIS
Changes Awareness
Reconfiguration is needed?
(non-functional QAs, resources, user preferences, contextual changes,...)
PLAN
Generate Reconfiguration of QAs
ECA Rules
Constraint Satisfaction Problem
Optimization Algorithms
Pre-Loaded Configurations
...
EXECUTE
Execute Reconfiguration (Weaving process)
M2M Transformations
Aspect-Oriented Programming
...
Application Architecture + QA Architecture
«App»
«QA»

Figure 1. Reconfiguration approach for QA functionalities.

Eliciting and modelling the relationships and dependencies between the base application, the QA functionalities, the non-functional QAs, and the reconfiguration contexts in order to adapt the software architecture at runtime is not an easy and straightforward task, as demonstrated in this small example. Explicitly specifying and separately modelling the QAs allows us to analyse how they affect the configuration of the application software architecture, and facilitates its adaptation at runtime.

3. Our Approach

In this section we present our approach to automatically reconfigure the software architecture of the QAs at runtime.

Like other existing reconfiguration approaches (Abbas et al. 2010; Pascual et al. 2015; Horcas et al. 2014b), we follow the MAPE-K loop of the Autonomic Computing (AC) paradigm (Kephart and Chess 2003), where "MAPE" stands for Monitoring, Analysis, Plan, Execution and 'K' stands for Knowledge. The Knowledge is one of the key aspects that differentiates a MAKE-K loop approach from others. In our approach, the **Knowledge** (top of the runtime part of Figure 1) includes the application and the QAs variability models and software architectures, as well as the information about the non-functional QAs, the user preferences and the device resources that may affect the functionality of the QAs. More importantly, it includes the dependency relationships between them. Finally, an initial configuration of the application and the QAs is included. This information needs to be modelled at design time (top of Figure 1), and the successful reconfiguration of the QAs in an application will largely depend on the level of details included in the variability modelling of the QAs and in their correctness. Our approach for specifying this information is described in Section 4.

The development of the automatic process to reconfigure the QAs at runtime consists in providing support to: (1) **Monitor** the application to listen for changes in the values of non-functional QAs, resources, user's preferences, context information, and any parameter previously modelled that may provoke a reconfiguration; (2) **Analyse** the criteria values to determine whether or not a reconfiguration of the QA functionalities is needed (`Changes Awareness` process in Figure 1); (3) generate a **Plan** for the new configuration with the set of changes that need to be done in the current configuration of the QA functionalities (`Generate Reconfiguration of QAs`). Different strategies and technologies can be applied to generate the new configuration (e.g., optimization algorithms, Event-Condition-Action rules, Constraint Satisfaction Problem); and (4) **Execute** the plan by adapting the QA configuration deployed inside the application architecture at runtime, according to the new configuration of the QA generated (`Execute Reconfiguration`).

Since we model the variability of the applications and the QAs separately, the reconfiguration of the QAs implies that the software architecture of the QAs is first customized and then composed/integrated into the architecture of the running application. To do so, we use model-to-model (M2M) transformations at runtime (e.g., using models@runtime), and Aspect-Oriented Programming (AOP) to inject (i.e., weave) the new configuration and remove (i.e., unweave) the previous one (`Weaving process` in Figure 1). As shown in grey in Figure 1, our contributions to the dynamic reconfiguration of the functionality of QAs are the variability modelling of the QAs, i.e., the Knowledge (see Section 4), and in the approach followed to execute the new configuration (not included due to space limitations).

4. Variability Modelling of QAs

This section describes how the variability management of the QAs and their functionalities can be achieved so as to successfully handle the reconfiguration of the QA functionalities of our case study at runtime. Concretely, we propose integrating the QAs and their functionality variability as part of the systematic variability management of an SPL. Continuing with our case study, our approach models, through different sub-tress (right to left in the variability model of Figure 2), the following: (1) the variability of the e-voting application functionality; (2) the variability of the QA functionalities; (3) the variability of the QAs as non-functional properties, and (4) the factors that can provoke a reconfiguration of the QA functionality, such as resources, user preferences and context information. The dependency relationships between them are modelled using cross-tree constraints.

The most relevant parts of our variability model are highlighted in both Figure 2 and the example shown in Figure 3:

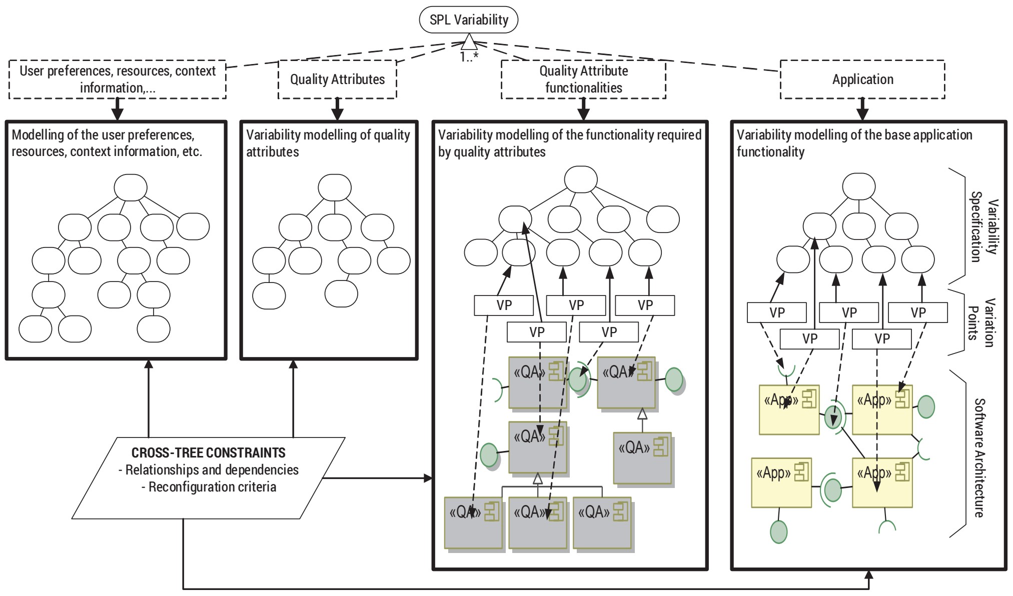

SPL Variability
1..*
User preferences, resources, context information,...
Quality Attributes
Quality Attribute functionalities
Application
Modelling of the user preferences, resources, context information, etc.
Variability modelling of quality attributes
Variability modelling of the functionality required by quality attributes
Variability modelling of the base application functionality
Variability Specification
Variation Points
Software Architecture
VP
«QA»
«App»
CROSS-TREE CONSTRAINTS
- Relationships and dependencies
- Reconfiguration criteria

Figure 2. Separately modelling the variability of the QA functionalities.

(APPLICATION BASE FUNCTIONAL VARIABILITY) Figure 2 shows that the base application and the QAs functionalities are separate and have their own software architecture and variation points. The variation points define how the architecture is modified when a particular feature is selected, which enables us to automatically generate a customized software architecture according to the features selected.

This separation between the software architecture of the base application and the software architecture of the QAs with functional implications makes sense because most of the components that implement the functionality required to fulfil the QAs are independent of the architecture of the base application, and can be reused in different application domains. For instance, the implementation of a particular encryption algorithm (e.g., RSA) does not change from one application to another. In addition, existing dependencies can also be modelled explicitly within the variability model. For instance, authentication mechanisms normally require the introduction of a panel in the application to allow users to introduce their credentials.

In Figure 3 we show that the functionality of the e-voting application (i.e., the different kinds of elections) is modelled separately from the non-functional QAs and from the functionality of the QAs required to satisfy them.

(QAs FUNCTIONAL VARIABILITY) In Figure 3, we can observe the functionality required to address security, persistence, and usability QAs.[2] There are several possibilities for satisfying the authentication requirement (e.g., using a digital certificate or a social identity, among others), multiple encryption algorithms available (e.g., RSA, DSA,...), and the possibility to provide contextual help to the user.

(QAs NON-FUNCTIONAL VARIABILITY) In addition to the functional variability of some QAs, there are variants of the QAs that are related to the architecture's intrinsic qualities, and the materialization of these non-functional QAs variability does not directly affect the components and relationships in the architecture, but it does affect them indirectly.

[2] We have omitted the variation points of the software architectures in Figure 3 for the sake of simplicity.

For instance, in Figure 3, we represent information regarding the security QA in two different subtrees. On the one hand, we model the functional variability of the components that implements the security features, as explained above. On the other hand, we model the variability of the security QA as non-functional properties indicating the assessed level of security (e.g., high, medium, low) that satisfies a specific product requirement. Thus, if a product requires a higher level of security, security functionality needs to be reconfigured to include a more secure authentication mechanism or a stronger encryption algorithm. The relationship is represented in the variability model as a cross-tree constraint between the level of security in the non-functional QAs (see cross-tree constraint `(2)` in Figure 3) and the feature representing the functionality to be included in the software architecture. This functional feature is associated with the component(s) in the QA architecture that provides the required level of security.

(RECONFIGURATION FACTORS) Apart from the non-functional QAs and the application functionality, there are other factors whose variations at runtime may affect the current configuration of the QA functionalities. As we stated in Section 1, variations at runtime in the context information, user preferences and needs, resources, etc. may all provoke a reconfiguration of the QA architecture.

We maintain the variability of those concerns separate in order to facilitate the modelling of them, and to explicitly define the reconfiguration criteria over the QA functionalities. For instance, as shown in Figure 3, we model the possible contexts in which the e-voting application can be executed (e.g., smartphone, desktop). In this way, the software architecture of the QA can be reconfigured based on the context information of the application — e.g., changing the usability level when the application detects that it is running over a desktop or smartphone, as a cross-tree constraint `(1)` as shown in Figure 3.

(DEPENDENCY RELATIONSHIPS) Here we highlight the different kinds of dependencies that exist between the QA functionalities and the rest of information in our QA variability model, and how these dependencies are modelled using tree and cross-tree constraints. Even though the relationships between

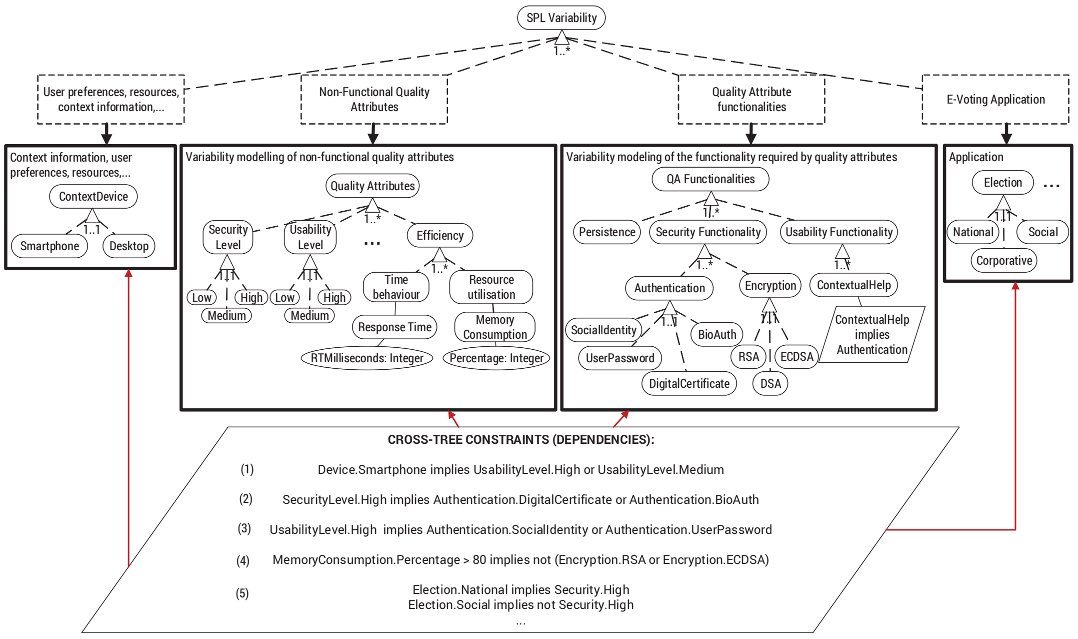

SPL Variability
1..*
User preferences, resources, context information,...
Non-Functional Quality Attributes
Quality Attribute functionalities
E-Voting Application
Context information, user preferences, resources,...
ContextDevice
1..1
Smartphone
Desktop
Variability modelling of non-functional quality attributes
Quality Attributes
1..*
Security Level
Usability Level
...
Efficiency
1..1
Low
High
Medium
Time behaviour
Resource utilisation
Response Time
Memory Consumption
RTMilliseconds: Integer
Percentage: Integer
Variability modeling of the functionality required by quality attributes
QA Functionalities
1..*
Persistence
Security Functionality
Usability Functionality
Authentication
Encryption
ContextualHelp
SocialIdentity
UserPassword
DigitalCertificate
BioAuth
RSA
ECDSA
DSA
ContextualHelp implies Authentication
Application
Election
...
National
Social
Corporative
CROSS-TREE CONSTRAINTS (DEPENDENCIES):
(1) Device.Smartphone implies UsabilityLevel.High or UsabilityLevel.Medium
(2) SecurityLevel.High implies Authentication.DigitalCertificate or Authentication.BioAuth
(3) UsabilityLevel.High implies Authentication.SocialIdentity or Authentication.UserPassword
(4) MemoryConsumption.Percentage > 80 implies not (Encryption.RSA or Encryption.ECDSA)
(5) Election.National implies Security.High
Election.Social implies not Security.High
...

Figure 3. Modelling dependency relationships between QA functionalities and reconfiguration criteria.

some of these elements have been already identified in previous research (Sinnema et al. 2006; Zhang et al. 2010; Pinto and Fuentes 2011; Horcas et al. 2014a), none of the existing work represent them explicitly in a variability model and none of them do it in a way in which the configuration of the QAs can be automated and performed dynamically at runtime.

Firstly, there are dependencies between the functionalities required by different QAs. For instance, in Figure 3, we can observe that the contextual help functionality requires authentication in order to provide customized help to the users based on their previous experience with the application. Secondly, non-functional QAs have direct dependencies with some QA functionalities. For example, identifying the user using a social network identifier (e.g., the user's Facebook profile) provides a high level of usability because it only requires a simple click by the user to log in to the application (see cross-tree constraint `(3)` in Figure 3). However, social identity authentication provides a low level of security. So, the security and usability functionality need to be reconfigured when the application requires a different level of security or usability. Thirdly, there are QA components that can have dependencies with the application, and are not obvious. For instance, national elections require a higher level of security than social elections (see cross-tree constraint `(5)`), and this implies changing the authentication mechanism in use (see cross-tree constraint `(2)`).

Another reconfiguration criterion based on the memory consumption of the application is modelled similarly (see cross-tree constraint `(4)`). This dependency forces a reconfiguration to avoid the use of a heavy encryption algorithms if the consumption exceeds 80% memory.

4.1 Variability Modelling of QAs with CVL

In order to demonstrate the feasibility of our approach, we use the Common Variability Language (CVL) (Haugen et al. 2008). CVL facilitates the QA variability modelling, in contrast to using classic feature models, principally for three reasons:

1. CVL allows us to characterize the possible variant space for QAs (Myllärniemi et al. 2012), without needing to extend the variability model to support additional attributes. For instance, the variant space for the *response time* concern can be characterized with a numeric metric, by using an integer variable in CVL (e.g., 200 milliseconds).
2. CVL provides units of modularization (i.e., Composite Variability Specifications and Configurable Units (CVL Submission Team 2012)) to specify the variability as a reusable sub-trees of the model that gather a set of reusable variation points.
3. CVL allows cross-tree constraints to be defined using the Object Constraint Language (OCL)[3], and thus, include dependency relationships with numeric variables.

Once a product configuration of the QA functionality has been generated at the feature level, the variability is resolved by executing the CVL engine and a customized architecture of the QAs is generated by applying the specific modifications defined in the variation points. These modifications depend on the architectural description language used to described the architecture and are beyond scope of this paper.[4] The new version of the QA architecture needs to be integrated with the current application architecture. This integration task can be done using MDD or AOSD techniques, as discussed in Section 3. This maintains the benefits of the separation of concerns exposed in the modelling of the QA functionalities.

5. Open Issues

We have identified some open issues in our approach to be completed in our ongoing work:

- In the dynamic reconfiguration of QAs the trade-off between them requires a special consideration. For instance, adding a component to the architecture may affect performance or introduce security vulnerabilities. Our approach supports the modelling of the QAs trade-offs as part of the variability models by extending the models with utilities values as in (Zhang et al.

[3] `http://www.omg.org/spec/OCL/`

[4] CVL supports any MOF-compliant language and provides a rich taxonomy of variation points, and custom model transformations (CVL Submission Team 2012).

2010). But in this paper we only consider some trade-off that can be modelled as a explicit dependencies between the features of the variability model. As future work, we plan to complete our approach to include complete support for QA trade-offs and safe reconfigurations.

- Specifying and modelling the dependencies between the QAs and the application is not a straightforward task and increases the complexity of applying our approach. However, this complexity is inherent to the problem that we are solving and does not depend on the applied approach. On the other hand, the manual definition of these dependencies does not guarantee that the resulting architecture complies with the desired QA. CVL dependencies are formally defined using tree and also cross-tree constraints, and apart from providing support to automate the management of the dependencies, they can be used to verify and check the correctness of the final software architectures.
- The existence of different types of dependencies such as structural, behavioural, or temporal dependencies needs to be taken into account. Each kind of dependency may need to be modelled in different levels — e.g., structural dependencies are modelled at the architectural level, while behavioural and temporal dependencies may also need sequence diagrams to correctly define them.

6. Conclusions and Future Work

Separately modelling the variability of the QAs from the variability of the base functionality of the application has many advantages (e.g., reusability, less coupled architectures, etc.). Our approach models the QA functionalities separately from the application functionalities and adds them as components into its software architecture. We propose a dynamic process to reconfigure the QA functionalities based on the factors (i.e., non-functional QAs, user preferences, resources, etc.) that may change at runtime and imply a change in the current configuration of the QAs already deployed within the application architecture. In order to do this we identify and model the dependency relationships between the QA functionalities and all those factors that may affect them, such as the non-functional QAs.

In our ongoing work we plan to solve all the open issues detected in our approach and study the impact of the technologies (e.g., optimization algorithms) in our reconfiguration process in order to complete its implementation.

Acknowledgments

Research funded by the Spanish projects TIN2012-34840 (co-funded by EU with FEDER funds) and MAGIC P12-TIC1814.

References

N. Abbas, J. Andersson, and W. Löwe. Autonomic software product lines (ASPL). In *ECSA*, 2010. ISBN 978-1-4503-0179-4. doi: 10.1145/1842752.1842812. URL http://doi.acm.org/10.1145/1842752.1842812.

O. Alam, J. Kienzle, and G. Mussbacher. Concern-oriented software design. In *MODELS*. 2013. ISBN 978-3-642-41532-6. doi: 10.1007/978-3-642-41533-3_37. URL http://dx.doi.org/10.1007/978-3-642-41533-3_37.

D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. *Managin Variability for SPLs*, pages 39–47, 2006.

R. d. O. Cavalcanti, E. S. de Almeida, and S. R. Meira. Extending the RiPLE-DE process with quality attribute variability realization. In *QoSA-ISARCS*, 2011.

CVL Submission Team. Common Variability Language (CVL), OMG revised submission. http://www.omgwiki.org/variability/, 2012.

L. Etxeberria, G. S. Mendieta, and L. Belategi. Modelling variation in quality attributes. In *VaMoS*, 2007. URL http://www.vamos-workshop.net/proceedings/VaMoS_2007_Proceedings.pdf.

L. Etxeberria, G. Sagardui, and L. Belategi. Quality aware software product line engineering. *J. Braz. Comp. Soc.*, 14(1):57–69, 2008.

M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou. Variability in software systems — a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306, 2014. ISSN 0098-5589. doi: 10.1109/TSE.2013.56.

B. González-Baixauli, M. A. Laguna, and Y. Crespo. Product line requirements based on goals, features and use cases. In *Requirements Reuse in System Family Engineering*, 2004.

O. Haugen, B. Moller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, 2008. doi: 10.1109/SPLC.2008.25.

J.-M. Horcas, M. Pinto, and L. Fuentes. Injecting quality attributes into software architectures with the common variability language. In *CBSE*, 2014a. ISBN 978-1-4503-2577-6. doi: 10.1145/2602458.2602460. URL http://doi.acm.org/10.1145/2602458.2602460.

J. M. Horcas, M. Pinto, and L. Fuentes. Runtime enforcement of dynamic security policies. In *European Conference on Software Architecture*, ECSA, pages 340–356, 2014b. doi: 10.1007/978-3-319-09970-5_29. URL http://dx.doi.org/10.1007/978-3-319-09970-5_29.

N. Juristo, A. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *IEEE Transactions on Software Engineering*, 33(11):744–758, 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70741.

J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1160055.

S. Mahdavi-Hezavehi, M. Galster, and P. Avgeriou. Variability in quality attributes of service-based software systems: A systematic literature review. *IST*, 55(2):320–343, 2013. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2012.08.010. URL http://www.sciencedirect.com/science/article/pii/S0950584912001772.

M. Matinlassi, E. Niemelä, and L. Dobrica. *Quality-driven Architecture Design and Quality Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture*. 2002.

A. Metzger and K. Pohl. Software product line engineering and variability management: Achievements and challenges. In *FOSE*, 2014. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593888. URL http://doi.acm.org/10.1145/2593882.2593888.

V. Myllärniemi, M. Raatikainen, and T. Männistö. A systematically conducted literature review: Quality attribute variability in software product lines. In *SPLC*, 2012. ISBN 978-1-4503-1094-9. doi: 10.1145/2362536.2362546. URL http://doi.acm.org/10.1145/2362536.2362546.

G. G. Pascual, M. Pinto, and L. Fuentes. Self-adaptation of mobile systems driven by the Common Variability Language. *Future Generation Computer Systems*, 47(0):127–144, 2015. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2014.08.015. URL http://www.sciencedirect.com/science/article/pii/S0167739X14001630.

M. Pinto and L. Fuentes. Modeling quality attributes with aspect-oriented architectural templates. *J. UCS*, 17(5):639–669, 2011. doi: 10.3217/jucs-017-05-0639. URL http://dx.doi.org/10.3217/jucs-017-05-0639.

M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Modeling dependencies in product families with COVAMOF. *ECBS*, 2006.

G. Zhang, H. Ye, and Y. Lin. Quality attributes assessment for feature-based product configuration in software product line. In *APSEC*, 2010. doi: 10.1109/APSEC.2010.25.

H. Zhang, S. Jarzabek, and B. Yang. Quality prediction and assessment for product lines. In *CAiSE*. 2003.