Injecting Quality Attributes into Software Architectures with the Common Variability Language

Jose-Miguel Horcas
Universidad de Málaga,
Andalucía Tech, Spain
horcas@lcc.uma.es

Mónica Pinto
Universidad de Málaga,
Andalucía Tech, Spain
pinto@lcc.uma.es

Lidia Fuentes
Universidad de Málaga,
Andalucía Tech, Spain
lff@lcc.uma.es

ABSTRACT

Quality attributes that add new behavior to the functional software architecture are known as functional quality attributes (FQAs). These FQAs are applied to pieces of software from small components to entire systems, usually crosscutting some of them. Due to this crosscutting nature, modeling them separately from the base application has many advantages (e.g. reusability, less coupled architectures). However, different applications may require differ-ent configurations of an FQA (e.g. different levels of secu-rity), so we need a language that: (i) easily expresses the variability of the FQAs at the architectural level; and that (ii) also facilitates the automatic generation of architectural configurations with custom-made FQAs. In this sense, the Common Variability Language (CVL) is extremely suited for use at the architectural level, not requiring the use of a particular architectural language to model base functional requirements. In this paper we propose a method based on CVL to: (i) model separately and generate FQAs cus-tomized to the application requirements; (ii) automatically inject customized FQA components into the architecture of the applications. We quantitatively evaluate our approach and discuss its benefits with a case study.

Keywords

CVL, quality attributes, SPL, variability, weaving

1. INTRODUCTION

The quality of a software system is measured by the extent to which it possesses a desired combination of quality attributes (QAs) [2] such as usability, reliability, security and scalability. Whether or not a system will be able to exhibit its desired (or required) QAs is substantially determined by its software architecture, through the use of different architectural tactics [3]. For some QAs, the architectural tactic to be followed consists in the injection (i.e. introduction) of specialized elements into the architecture (e.g. an authorization mechanism to satisfy the security QA) [3]. These QAs are normally known as functional quality attributes (FQAs) [12], and are different from other QAs such as cost or efficiency that can be mapped to architectural or implementation decisions, but not directly to functional components. Examples of FQAs are error handling, security, context awareness, usability, persistence, recovery, etc.

Our main idea is to give priority to the FQAs that are required by an application from the early stages of the software development, based on three main contributions: (1) the specialized elements required for satisfying the FQAs are modeled separately from the base software architecture; (2) these elements are then semi-automatically injected into the application architecture; and (3) the approach is implemented using the Common Variability Language (CVL) [10].

The motivation for modeling FQAs separately from the base architecture is that FQAs are normally required by several applications — i.e. they are recurrent, and most of them crosscut the system architecture. So, modeling them separately from the base application has many advantages (e.g. reusability, less coupled architectures, etc.). For instance, an encryption algorithm used to encrypt the information does not depend on the application that needs it. Also, different applications may require different levels of an FQA (e.g. security). For example, a specific application may require access control and anonymity while another may require only encryption, or may require a different kind of encryption algorithm. In other words, there is much variability in FQAs, and the Software Architect (SA) should be able to select the set of specialized architectural elements that need to be injected to fulfill the application requirements regarding a particular FQA, which is not a trivial task.

Regarding the second part of our approach, once we have generated custom models of the FQAs for a given application, these models need to be injected into the base model of the application. Being inspired by Aspect-Oriented Modeling (AOM)[1], in our approach, first we identify and select the points in the base model where the custom FQA models have to be injected, and then automatically generate the application architecture woven with the customized FQAs.

Finally, our third contribution is the technical details of our approach. In order to define a family of FQAs, a language to model the variability of FQAs is needed. Although most of the variability approaches use Feature Models (FMs), their main shortcoming is that an additional process is re-

[1] http://www.aspect-modeling.org/

quired to generate an architectural configuration that meets an FM configuration. CVL is more suitable for use at the architectural level, since it defines links between the variability specification and the product line architecture (PLA). The advantages of using CVL are: (i) it is an MOF-based variability language and this means that any MOF-based architectural language can be used with the variability information of CVL; (ii) the links between the variability and base architectural models make it possible to automatically generate software architecture configurations, ensuring that they fulfill the variability specification, (iii) the semantic of the CVL variation points can be extended, and (iv) CVL includes the most important characteristics of similar variability models (e.g. FM) such as cardinality of variation points, cross-tree constraints, etc. Due to all these advantages there is a great interest in the SPL community in adopting CVL in their proposals [5, 6]. But, since both the CVL language and its tool support are novel, the effort of using CVL is currently considerable.

Summarizing, in this paper we present an SPL approach based on the use of CVL to automatically generate, from a family of reusable FQAs, software architecture configurations that include custom made FQAs. As discussed further on, in Section 3, this approach is an extension of our previous work [11] and defines a more generic, integrated and extensible approach. One important contribution is that the custom made FQAs are automatically woven with the base application by extending the semantic of the CVL variation points. We quantitatively evaluate our approach by using appropriate metrics to assess the benefits of our approach, and illustrate it with an Intelligent Transportation System case study. Also, we discuss the benefits of using CVL in achieving the goals posed in this paper.

Besides this introduction, Section 2 presents the CVL, and describes the case study used throughout the paper. Section 3 overviews our approach and highlights its main novelties in comparison with our previous work [11]. In Section 4 we explain in detail how we model FQAs by using CVL. The customization and injection of the FQAs into the base application of our case study is explained in Sections 5 and 6, respectively. In Section 7 we evaluate our proposal. Section 8 discusses the related work. Finally, Section 9 concludes the paper and presents further work.

2. BACKGROUND INFORMATION

In this section we briefly summarize CVL.[2] Then, the case study followed throughout the paper is described.

2.1 CVL

The CVL is a domain-independent language for specifying and resolving variability over MOF-compliant models.

CVL provides an executable engine to automatically produce a *resolved model* from three main models: (1) the *base model* over which the variability is specified and resolved. (2) The *variability model* that specificies the variability in an abstract level with *variability specifications* (*VSpecs*) and in a concrete level through *variation points*. VSpecs are tree structures representing choices ("features" in most SPL terminologies) and can include logical constraints defined in a subset of the Object Constraint Language (OCL). Variation points define specific modifications to be applied to the base model during *materialization* — i.e. the process of transforming a base model into a configured product model. (3) A *resolution model* that provides resolutions for the VSpecs in order to materialize a base model with a variability model. The main characteristics of CVL that we will use are:

Configurable unit (CU). Set of variation points that hides the internal of a base model, exposing a *VInterface*.

Variability interface (VInterface). Group of VSpecs that have to be resolved to materialize a CU.

Composite VSpec (CVSpec). VSpec that is resolved by resolving other VSpecs that are *variability interfaces*.

Opaque Variation Point (OVP). It allows customizing the semantic of the existing CVL variation points through a model transformation language.

2.2 Case Study

We motivate our proposal presenting a case study based on Intelligent Transportation Systems (ITSs) [1]. In this context, there is a set of services (e.g. road safety, weather conditions,...) that all require communication between vehicles (V2V) and via roadside access points (V2I).

An ITS application requires specific **security** services: *privacy*, to protect the personal information of drivers such as the route followed; *integrity*, to ensure the data authenticity exchanged over the network; and *confidentiality* and *authentication*, to allow the drivers to use certain services of V2I (e.g. payment at electronic toll). **Context awareness** and **usability** FQAs are also required in order to obtain *context information* of the user (e.g. license detector, weariness) or of the car (e.g. GPS, proximity sensors) and to provide *contextual help* according to the users needs, respectively.

3. OUR PROPOSAL

This section presents a general overview of our approach (Figure 1). We distinguish three main stages with two different actors: (1) FQAs modeling, performed by a domain expert in quality attributes; (2) FQAs customization to the requirements of a particular application, and (3) FQAs weaving. Steps (2) and (3) are performed by the SA.

Stage 1: FQAs modeling.

In this stage, an expert in the domain of the QAs models the variability of FQAs following the CVL approach. To do this, he/she first builds the software architecture of the FQAs by using any MOF-compliant language (`FQAs Base Model` in Figure 1), and then defines a variability model of the FQAs in CVL (`FQAs Variability Model`). This supposes a novelty in comparison with our previous work [11] in which the FQAs had to be obligatorily modeled with aspect-oriented software architectures, making use of a propietary ADL and of propietary tools.

Most of the FQAs are composed by many concerns. The security FQA, for example, is composed by access control, authentication, privacy, integrity, and encryption, among other concerns. However, not all of the concerns of an FQA are required by all the systems. For example, an application may require only authentication and access control. Also, the domain expert needs to consider that some of the concerns of an FQA have dependencies between them, such as the confidentiality concern that depends on the encryption concern to ensure that all the information is encrypted and cannot be obtained by third persons. We call these dependency relationships between the concerns of the same FQA,

[2]Complete description in `http://www.omgwiki.org/variability/`

intraFQA-dependencies. Furthermore, FQAs affect each other, so dependency relationships between different FQAs must also be considered. For instance, the contextual help concern of the usability FQA depends on the authentication concern of the security FQA in order to provide customized help based on the previous experience of the user. We call these dependency relationships between concerns of different FQAs, **interFQA-dependencies**.

This supposes a difference between our approach and other proposals that address FQAs' variability (e.g. QADA [13], RiPLE-DE [4]), basically because they model these FQAs as part of the domain analysis of an SPL, and not separately as we propose.

An important thing that is worth highlighting is that this stage is performed only once. This means that the FQAs base model and the FQAs variability model will be completely reused by any application that wants to incorporate these FQAs into its software architecture, just by following stages 2 and 3 of our approach.

Stage 2: FQAs customization.

In the second stage of our approach, the SA creates a configuration of the FQAs (`FQAs Resolution Model`) according to the requirements of a particular application. This means that those variable concerns that are not required by the base application will not be incorporated into the final application. This stage is automated by using the execution engine of CVL (`CVL Execution`) that resolves the variability of the FQAs variation points. This is another novelty of this approach. Previously, in [11], an additional language had to be used to link the FQAs variability model with the FQAs base model. Moreover, the customization of the FQAs depended on the definition and instantiation of aspect-oriented architectural templates, which were defined using our proprietary ADL. The CVL engine takes as inputs the `FQAs Resolution Model`, the `FQAs Variability Model` and the `FQAs Base Model`, and automatically derives (materializes) the resolved model of the FQAs (`FQAs Resolved Model`). This model only contains the software elements of the FQAs that are needed according to the requirements of the application.

Stage 3: FQAs weaving.

Once the FQA resolved model has been generated in the previous stage, the next step consists of "weaving", or composing, it with the software architecture of the base application (`Application Base Model` in Figure 1). The output of this weaving process is an application architecture that also incorporates the FQAs (`Application Resolved Model (application architecture + FQAs)`). This weaving is not a straightforward task since each FQA will have to be woven at different points of the base applications (join points) and, furthermore, each FQA's concerns will be woven according to a different weaving pattern, depending on the semantic of the concern. Moreover, this should be done automatically, without manually modifying the application architecture.

Thus, the challenge here is to define a process that systematically integrates high-level quality solutions into the base architecture of a given application, but without having to understand the inner workings of the quality solutions. In order to do that, we need to adapt the CVL approach (see Section 6). Firstly, our based model is formed by two models, the `FQAs Resolved Model` and the `Application Architectural Model`. Secondly, during the `CVL Execution` the control must be delegated to a Model-2-Model engine (`M2M Engine`) such as QVT, in charge of performing the weaving of these two models. The weaving is performed according to the weaving information provided by the `Application Weaving Model`. This model uses the OVPs of CVL, which are the mechanism provided by CVL to extend the semantic of the CVL variation points. Using the OVPs the weaving rules that indicate where the FQAs must be incorporated into the core software architecture are specified as user-defined M2M transformations (`Weaving Rules (M2M transformations)`). The weaving process introduced here, and detailed in Section 6, is a new contribution of this paper and was not part of the proposal presented in [11].

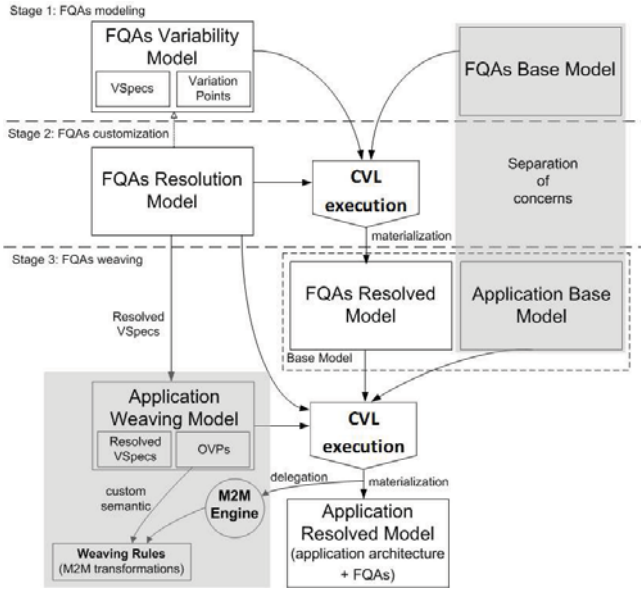

Figure 1: Our approach for modeling FQAs.

The rest of the paper describes each stage in more detail, using the case study described in the previous section.

4. MODELING FQAS WITH CVL

This section describes the first stage of our approach, in which the FQAs (security, context awareness, and usability), their commonalities and variabilities, and the dependencies between them are modeled.

4.1 FQAs base model

In our approach the `FQAs Base Model` specifies the software architecture of the FQAs. For instance, the UML software architecture modeling the functionality of the different concerns of the security FQA is shown at the bottom of Figure 2. This architectural model should include the complete functionality of the security FQA. In order to simplify the model, we only include here the `Integrity`, `Confidentiality`, `Encryption`, `Authentication`, and `Hash` components. These are composite components that include other necessary components to implement the functionality of each concern. For instance, `Encryption` includes components to encrypt and decrypt information using different encryption algorithms.

In order to achieve a better modularization we model each FQA independently of each other and then we model the dependencies and interactions between them. Thus, the bottom of Figure 3 shows the high-level architectural model of all the FQAs together. Although not included due to

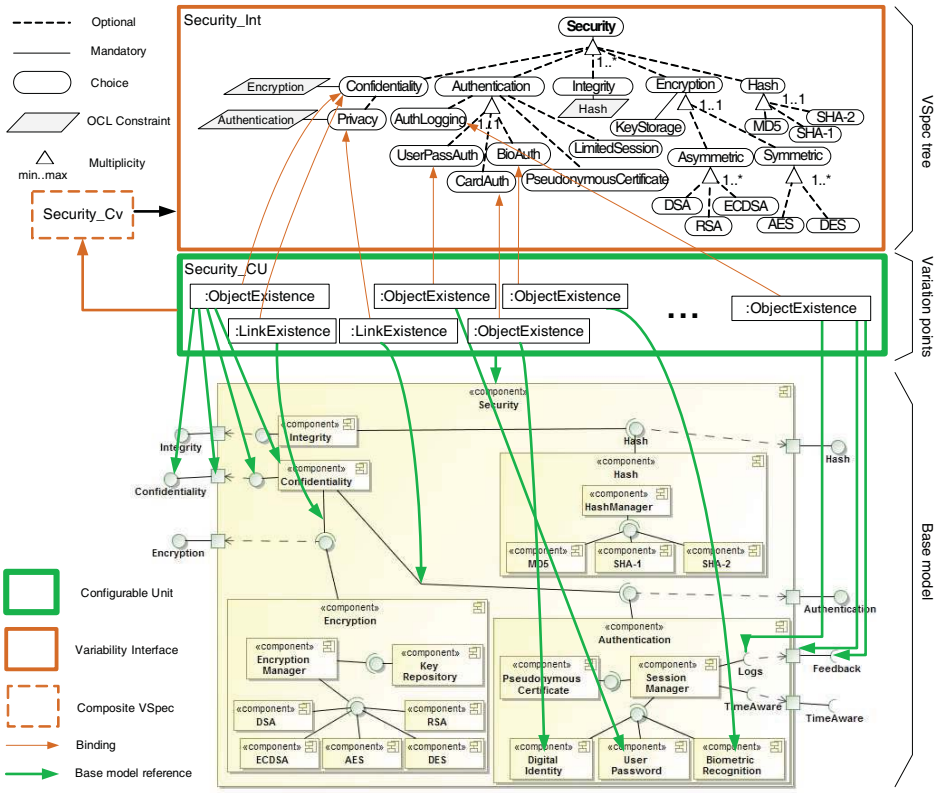

Figure 2: Modeling security FQA in CVL.

the lack of space, the software architectures of the context awareness and the usability FQAs are modeled in a similar way as for the security FQA.

4.2 FQAs variability model

Once the FQAs base model has been specified, the next step is to model the `FQAs Variability Model`. This CVL variability model includes the VSpecs, the variation points, the bindings between the variation points and the VSpecs, and the references from the variation points to the base model.

As an example, the variability model of the security FQA is shown in Figure 2. We distinguish three parts: (1) the VSpec tree of the security FQA (top of Figure 2), (2) the base model of the security FQA (bottom of Figure 2), and (3) the variation points (middle of Figure 2). The variation points are grouped into the security configurable unit (`Security_CU`). Then, the `Security_CU` is bound to a composite VSpec (`Security_Cv`) which refers to the variability interface `Security_Int` — i.e. the VSpec tree.

In the `Security_Int` VSpec we identify all the concerns that are part of the security attribute, model them by choices whose later resolution requires a yes/no decision, indicate which are optional and which are mandatory, and what the intraFQA-dependencies between them are. As stated, in this VSpec tree we only show a subset of the security concerns. These concerns are also composed by other concerns. For instance, there are different kinds of authentication: user + password (`UserPassAuth`), intelligent card (`CardAuth`), and biometric (`BioAuth`).

The kind of variability that we need to express is that "not all of the concerns of an FQA are required by an application and thus, not all of the components of that FQA base model need to be incorporated into the application architecture". In CVL this kind of variability is expressed by using the "existence" variation point that indicates the existence of a particular object, link, or value in the base model. Finally, in CVL, the variation points need to be bound to elements of the VSpec tree and need to refer to elements of the FQA base model. This is how the relationship between the variability model and the base model is specified in CVL. Moreover, these links are used by the CVL execution engine to automate the generation of a product configuration. For instance, the variation point bound to the `Confidentiality` concern in the `Security_Int` VSpec (`:ObjectExistence`) indicates that if confidentiality is decided positively (marked as "True" in the resolution model) in a configuration, the related elements (the `Confidentiality` component and the associated interfaces and ports with their attachments) in the base model will exist in the final application and if confidentiality is decided negatively (marked as "False" in the resolution model) those related elements will be removed from the FQA resolved model.

In order to maintain the consistency and to achieve a good modularization of the design, we specify the variability model at the same abstraction level as we did for the FQA base model. This means that the variability of each FQA is specified independently from the variability of the other FQAs. This can be done in CVL by using composite VSpecs and configurable units. Then, we relate the different variability models defining a "complete" variability model including all the FQAs with their relationships (Figure 3). We apply the conceptual integrity principle[3] to compose the different FQAs configurable units. As Figure 3 shows, the FQAs VSpec includes the three FQAs by including the three

[3]The overall design pattern of a system is reflected in any part of the system.

composite VSpecs previously created (`Security_Cv`, `Usability_Cv`, and `ContextAwareness_Cv`). These VSpecs refer to the interfaces of the configurable unit of each FQA (`Security_CU`, `Usability_CU`, and `ContextAwareness_CU`). Each configurable unit refers to its own composite component in the FQAs base model (bottom of Figure 3). Each of these components have inner variability that we have previously modeled for each FQA.

4.2.1 *Dependency modeling*

Our approach models the dependencies by using the CVL constraints. CVL constraints express relationships between elements of the VSpec that cannot be directly defined by hierarchical relations. We define the intraFQA- and the interFQA-dependencies at a different level of abstraction: intraFQA-dependencies are defined in the context of each FQA configurable unit while interFQA-dependencies are defined in the context of the complete FQAs variability model.

IntraFQA-dependencies. In Figure 2, each intraFQA-dependency is represented by a prepositional constraint (expressed in OCL) in a parallelogram that captures a condition in a choice. For instance, the dependency 'confidentiality requires encryption' is modeled by attaching the constraint `Encryption` to the choice `Confidentiality`. Thus, the choice `Encryption` has to be positively decided whenever `Confidentiality` is positively decided.

Dependencies are also presented in the security base model. For instance, although confidentiality affects data in general and only requires encryption, privacy also affects people's information and requires the authentication of the user. So, there is a dependency between the *privacy* concern and the *authentication* concern. In Figure 2, the variation point (`:LinkExistence`) bound to the `Privacy` choice refers to a required interface of the `Confidentiality` component in the UML security base model. This link represents the dependency relationship between the *privacy* and the *authentication* concerns. The variation point indicates the existence of that particular link. If `Privacy` is decided positively in a resolution model the link will exist in the resolved model, and if `Privacy` is decided negatively, the link will be removed.

InterFQA-dependencies. The `FQAs_Int` VSpec of Figure 3 includes the CVL constraints that specify the interFQA-dependencies. For instance, the constraint `AuthLogging implies HistoryLog` represents the dependency between the *authentication logging* concern of the security FQA and the *history log* concern of the usability FQA, and means that if the `AuthLogging` choice is decided positively, the `HistoryLog` choice has to be positively decided too. InterFQA-dependencies are also presented in the FQAs base model. It shows how the `Security` component relates with the `Context-Awareness` and `Usability` components due to the existing dependencies between them. For instance, `Authentication` is required by `Usability` in order to provide contextual help. Also, `Security` requires `Feedback` from `Usability` and `TimeAware` from `Context-Awareness` in order to allow the authentication logs and the control of the session time respectively. Finally, `Context-Awareness` requires `Feedback` to provide alerts when the information context demands it.

5. CUSTOMIZATION OF THE FQAS

In the second stage of our approach a valid configuration (customization) of the FQAs variability model is generated, taken as input the requirements of the application under development, which in our case is the ITS case study.

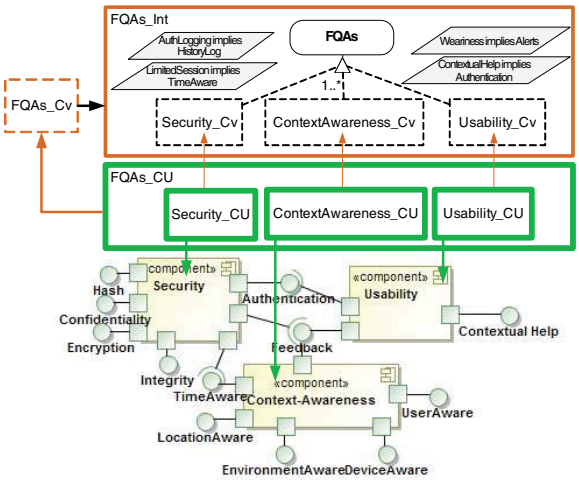

Figure 3: FQAs variability model.

As shown in Figure 1, the customization of the FQAs implies the definition of a `FQAs Resolution Model`. A resolution model is created by deciding which choices of the VSpec tree are positively decided and which ones are negatively decided. The VSpec at the top of Figure 4[4] shows a valid configuration of the FQAs (a resolution model) that satisfies the requirements of our ITS application. Some of the choices shown in Figure 4 have been selected because the application requirements explicitly identified them as needed, such as privacy, integrity, confidentiality, authentication, location aware, user aware and contextual help. Other concerns, such as encryption, hash, feedback, and time aware, had to be selected in order to obtain a valid configuration due to the existing dependencies between those concerns and those that were originally required.

For instance, there are concerns that had to be selected due to the parent-child relationship in the VSpec (e.g. the `Confidentiality` and the `Privacy` choices). Other concerns had to be selected because of the intraFQA-dependencies between concerns of the same FQA (e.g. `Confidentiality` and `Encryption`). Finally, other concerns had to be selected due to the interFQA-dependencies between concerns of different FQAs (e.g. `AuthLogging` of the `Authentication` concern in the security FQA and `Log` of the usability FQA).

Thus, in our case study, there are concerns, as is the case of the encryption or the hash concerns, that are required by other concerns but had not explicitly specified as part of the application requirements. This occurs basically because these dependencies are not always obvious to the application requirements engineer or the SA. By having a domain expert specifying the intraFQA- and interFQA-dependences as part of the definition of the FQAs variability in the first stage of our approach, in this second stage our approach helps the SAs to specify software architectures that are more accurate regarding the specification and customization of the FQAs to the necessities of the applications.

Once the resolution model has been created, the `CVL Execution` engine is executed to automatically generate the `FQAs Resolved Model`, which is shown in Figure 5. Only the necessary functional components are included in the resulting FQAs software architecture.

[4]Middle and bottom of Figure 4 are described in the next section. We show only one figure for reasons of space.

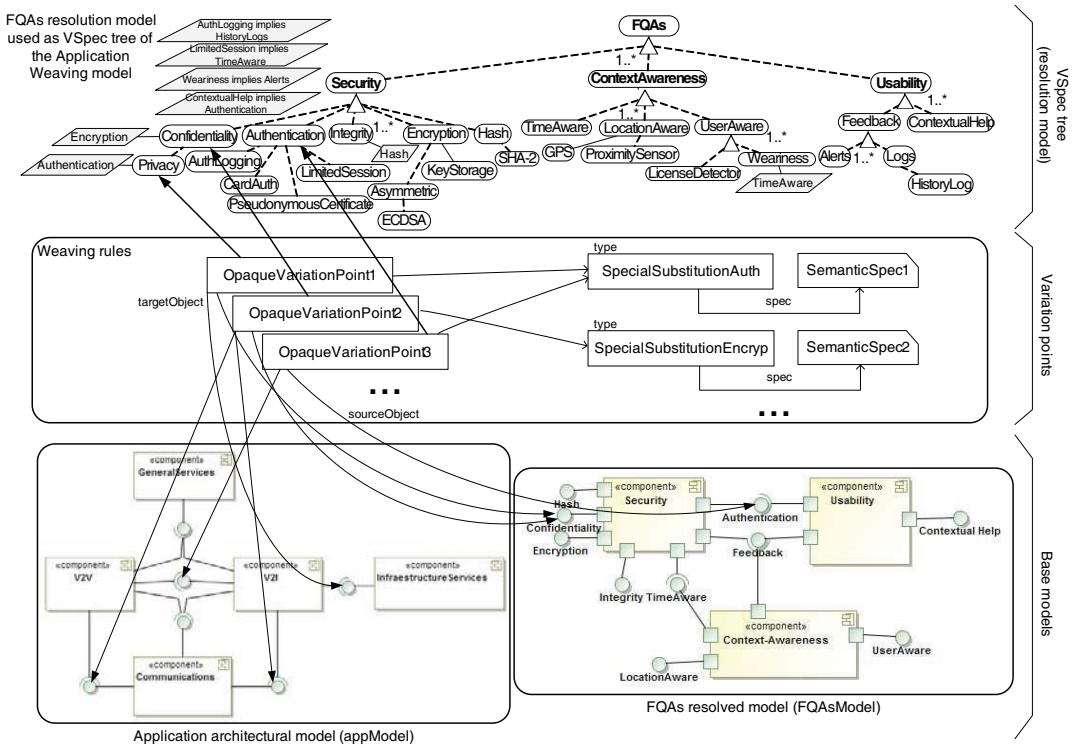

Figure 4: CVL model for the injection of the FQAs inside the application architecture.

6. WEAVING THE CUSTOMIZED FQAS

At this point, an architectural model customized with the required FQAs and concerns has been generated (high level view in the `FQAsModel` in Figure 4 and low level view in Figure 5). Now, in the third stage of our approach this FQAs resolved model has to be incorporated into the software architecture of the core application (`appModel` in Figure 4).

As previously stated, each concern of the FQAs needs a particular transformation because it is woven with the application in a different way. The `Application Weaving Model` (Figure 4) uses the OVPs of CVL to define a new semantic of a variation point using model transformation rules. The steps are: (1) binding an OVP to each of the concerns of the FQAs in the resolution model (e.g. encryption, contextual help); (2) specifying a reference to the specific architectural elements that model that concern in the FQAs resolved model, and (3) indicating how the concern will be woven with the application base model. OVPs are also bound to an OVPType, where this type explicitly defines the semantic of a special substitution. A special substitution implies the combination of a standard substitution and user defined transformations. In our case the special substitution is done from the "source object" to the "target object" referenced by the OVP as we show in Figure 4. Based on this special substitution the SA needs to refer to one or more join points (target objects) in the application model where the behavior of the selected concern (source object) will be incorporated.

We have identified a set of weaving patterns (Table 1) that fulfil our needs to incorporate the FQAs concerns based on how the functionality of the concerns (advices) needs to be applied into the different points of the application (join points). The special substitution needed by each concern is mapped to one weaving pattern presented in Table 1.[5] For instance, the privacy and the authentication concerns have the same special substitution that indicates that only one advice (e.g. `authenticate()`) of the selected concern (e.g. authentication) is woven into the selected join point (e.g. an interface of the base application).

Continuing with our example, to simplify Figure 4, we only show three OVPs bound to three concerns: confidentiality, which is achieved using encryption, privacy, and authentication. For instance, the `OpaqueVariationPoint1` is bound to the `Privacy` concern in the VSpec tree and it is also bound to the OVPType `SpecialSubstitutionAuth`[6] (OVPType 1 in Table 1). This in turn specifies the semantics of the special substitution with the necessary transformations (weaving rules) to incorporate the related element (`Confidentiality` that contains the `Privacy` functionality) of the FQAs resolved model (FQAsModel in Figure 4) into the application base model (AppModel in Figure 4).

Algorithm 1 shows the weaving process during the variability resolution performed by the CVL engine. It takes the set of OVPs defined (S_{OVP}) and for each OVP the semantic of the special substitution associated (OVPType) is executed by the M2M transformation engine (line 4). $S_{joinpoints}$ is the set of join points referenced by the OVP. When CVL is executed taking as inputs the application weaving model, the application base model (appModel) and the FQAs resolved model, the output is an automatically generated model representing the complete application software architecture (Figure 6), which includes the custom FQAs.

Note that the configuration of the architecture (the components and their relationships) is clearly visible in the static part of the design. Additionally, stereotyped dependencies

[5]Implementation in ATL of the M2M transformations are available in `http://caosd.lcc.uma.es/spl/cvl/CVL-models-transformations.zip`

[6]In our approach privacy is achieved using authentication.

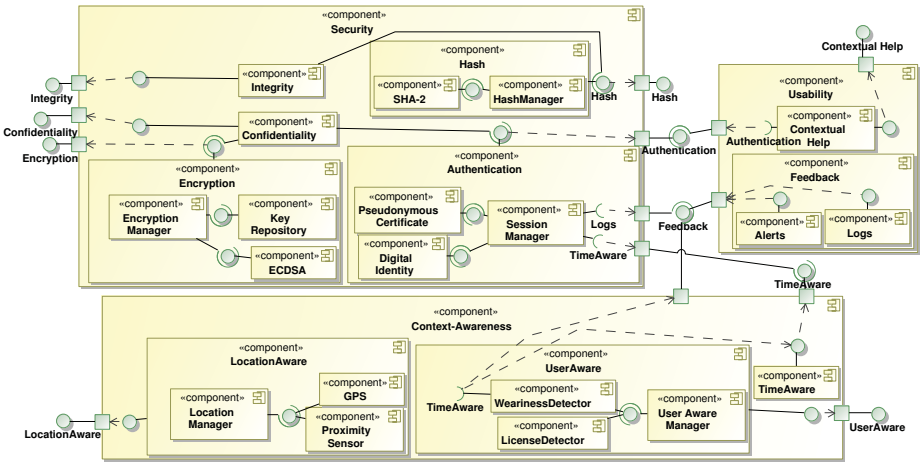

Figure 5: FQAs resolved model.

Table 1: Special substitutions.

OVPType	Description	Example
1	Only one advice of a concern is woven into a join point.	**Authentication**: the `authenticate()` advice is performed *around* the join point.
2	The same advice is woven multiple times into a join point.	**Time aware**: `currentTime()` is applied twice (*before* and *after*) to measure the time session of the user.
3	The same advice is woven into different join points.	**Location aware**: the `acquirePosition()` advice needs to be applied on the client and on the server side to establish locations.
4	Multiple advices of the same concern are woven into a join point.	**Feedback**: `log()` advices are invoked *before* and *after* the join point.
5	Multiple advices of the same concern are woven into different join points.	**Encryption**: encrypt the information (`encrypt(Object)`) *before* sending it and decrypt it (`decrypt(Object)`) *after* receiving it.
6	Advices of different concerns are woven into a join point.	**Contextual help**: first check whether the user is authenticated (`isAuthenticated()`) and then show information (`showHelp()`) based on the preferences of the user.
7	Advices of different concerns are woven into different join points.	**Integrity**: `hash(Object)` is applied *before* sending information to the server and `checkIntegrity(Object)` is applied *before* use the information in the server.

Algorithm 1 Weaving process using CVL.

Require: S_{OVP}
Ensure: ResolvedModel
1: **for all** vp in S_{OVP} **do**
2: $c \leftarrow vp$.sourceObject
3: $S_{joinpoints} \leftarrow vp$.targetObjects
4: ResolvedModel $\leftarrow$ specialSubstitution(c, $S_{joinpoints}$, vp.type)
5: **end for**
6: **return** ResolvedModel

between components of the FQAs and components of the base application make explicit that the sources of the relationships crosscuts the architectural level, and the targets are the point of the application where they take place (i.e. the join points in the AOM terminology). However, this is insufficient because the interactions between the components are not represented. In order to solve this limitation, in our approach the interactions between the components are represented in a set of sequence diagrams that are also automatically generated [14] by the transformation rules of the special substitutions. For instance, the behavior of the crosscutting relationship of the authentication concern is represented in the sequence diagram in Figure 6 (b). This diagram shows how the `V2V` component invokes the `paymentToll()` method of the `InfraService` interface, and that before the call is effective the `authenticate()` method of the `Authentication` component is invoked. Similarly, the sequence diagram of Figure 6 (c) shows the behavior of the encryption concern when the `V2V` component transmits information; that is, before sending the message to the `Communication` component the `encrypt()` method of the `Encryption` component is invoked.

7. EVALUATION

We evaluate our work both quantitatively, by using metrics to quantify the benefits provided by our approach, and qualitatively, when the use of a metric does not make sense.

Table 2 describes the metrics suite that have been used to evaluate our approach. Using these metrics we have identified that there is: (1) a *degree of dependency* between FQAs (metrics 1–3). This means that there is a significant number of concerns whose inclusion in a particular solution depends on the correct identification of these dependencies, which are not always straightforwardly derived from the requirements of the system. Consequently, these dependencies may go unnoticed by the SA even if they should be taken into account to satisfy the requirements of a system; (2) a high *degree of variability* (metrics 4–6), since modeling the FQAs as a "family" of products, and the automatic materialization of the FQAs base model considerably increases the number of "valid" resolutions of a FQA that can be generated; (3) a high *degree of automation*, due to the high number of architectural elements that are automatically generated in comparison with the manual effort that needs to be made in order to include the FQAs inside the software architectures of the applications. This degree of automation implies a lower development effort and gains in productivity; and (4) a high *degree of separation of concerns* (metrics 7–8), which is improved due to splitting our approach into three stages.

We present the results of modeling the three FQAs described in this paper: security, context awareness and usability.

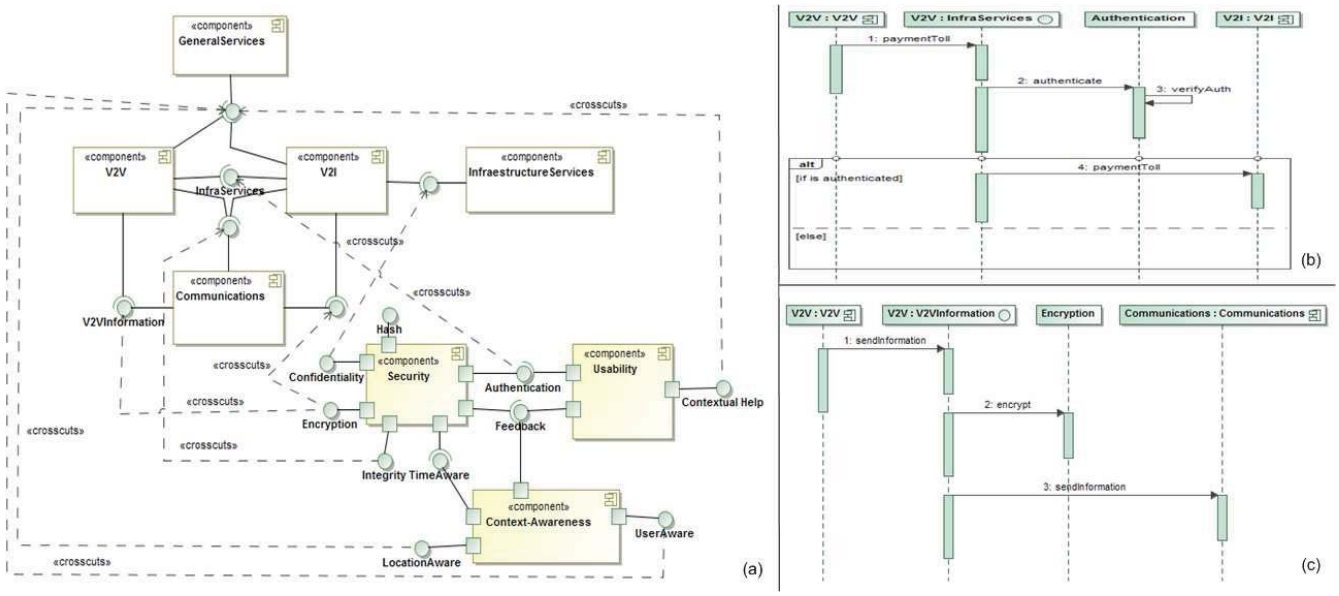

Figure 6: Complete application architecture (a); authentication (b) and encryption (c) sequence diagrams.

Table 2: Metric Suite

Dependency Metrics	**1. #intraFQA-dependencies:** It measures the number of dependencies (tree-constraints and OCL constraints) between the concerns of a FQA. **2. #interFQA-dependencies:** It measures the number of dependencies (cross tree-constraints and OCL constraints) between different FQAs. **3. #dependent-elements:** It measures the minimum number of architectural elements (components, interfaces, relationships,...) that need to be defined due to the existence of a dependency.
Variability Metrics	**4. #choices:** It measures the total number of choices in a VSpec. **5. #resolutions:** It measures the total number of valid resolutions that can be generated from a VSpec. **6. Variability level:** Expressed as the ratio `#choices:#valid resolutions`.
Separation of Concerns Metrics	**7. Lack of concern-based cohesion (LCC):** It measures the number of concerns tangled in a particular component. **8. Concern diffusion over architectural components (CDAC):** It measures the number of components in which a concern is scattered.

Degree of Dependency.

The degree of dependency of the FQAs is shown in Table 3. By applying the dependency metrics (see Table 2), we count the number of intraFQA- and interFQA- dependencies, as well as the number of architectural elements that are needed in order to satisfy those dependencies. Note that these numbers correspond only with the concerns of the three FQAs presented in this paper.

At this point, we would like to highlight that each FQA dependency can imply the incorporation of a considerable number of architectural elements into the software architecture. For instance, the security FQA has only four intraFQA-dependencies but, in order to satisfy these four dependencies, the SA needs to define at least 33 architectural elements. InterFQA-dependencies are even more difficult to satisfy because they involve concerns of the other FQAs. For instance, to satisfy the interFQA-dependency of the usability FQA with the security FQA, the SA needs to define at least 13 architectural elements.

The relevant issue here is that, using our approach, this complexity is not managed directly by the SA. Basically, because we make those dependencies explicit to make sure he/she is made aware of them and must only select the desired concerns that satisfy the dependencies. Then, the required architectural elements are automatically incorporated into the architecture. Note that this is certain only by assuming that the domain experts correctly do their job of modeling the FQAs and the dependencies between them.

Table 3: Degree of dependency

Dependencies	Security	Context awareness	Usability
#intraFQA	4	3	1
#elements	33	12	3
#interFQA	2	1	1
#elements	21	11	13

Table 4: Degree of variability

	Security	Context awareness	Usability	Total
#choices	23	15	10	48
#components	21	19	7	47
#resolutions	5354	575	79	313784

Degree of Variability.

Table 4 shows the number of choices that were specified in the VSpec of the security, the context awareness, and the usability FQAs, the number of components that were defined in the software architecture of each FQA and the number of different "valid" resolutions (configurations) that can be generated using our approach.

The number of possible resolutions will depend on the number of initial choices and on the number of dependencies between choices in the VSpec. In Table 4, we see that, for security, the domain expert will only once have to make, the effort of defining a VSpec with 23 choices and of specifying an architectural model with 21 components. Then the CVL engine automatically generates one of the 5354 valid security configurations based on the selections in the VSpecs done by the SA. The correctness of the generated software architectures will depend on the correct specification of the variability model. The variability level is lower for context

Table 5: Degree of automation

Case study	Specified elements manually	Specified elements automatically	Degree of automation
ITS	27	25	48.08%
HW	42	38	47.50%
TS	115	16	12.20%
CS	43	34	44.20%

awareness (15 choices: 575 resolutions) and usability (10 choices: 79 resolutions) than for security.

These numbers indicate that some FQAs have a high degree of variability — i.e. there are many different configurations of security that can be created to satisfy the security requirements of different applications, and the manual specification of these configurations by the SA (5354 in the case of security) is a hard and error-prone task. So, this metric indicates that the use of an SPL approach makes sense.

Degree of Automation.

Defining the architectural model of the FQAs and the variability model with the VSpecs and the variation points is a difficult and specialized task. However, in our approach, this effort only needs to be made once and then the FQA models can be reused in the development of many systems. Therefore, the main effort consists of modeling the core architecture of the application. Thus, in order to specify the degree of automation of our approach, we compare the number of architectural elements (i.e. components, provided and required interfaces, and relationships) that are manually created in the specification of the core software architecture with the number of architectural elements that are automatically generated, as defined in Equation 1. This is an adaptation of the degree of automation defined in [8], to be able to use this metric at the architectural level. Basically, at the architectural level, the complexity of defining a software architecture is measured by the complexity of designing their components, interfaces, and the relationships between those components; and it does not depend on the complexity of the particular implementation of the components.

$$\text{Degree of Automation} = \frac{\#elements\_FQAs}{\#elements\_core + \#elements\_FQAs} \quad (1)$$

We have applied the degree of automation in our approach when: (1) security, context awareness and usability are added to the ITS case study, and (2) security and usability are added to a health watcher (HW) industrial case study, a toll (TS) case study and a crisis management (CS) case study (see Table 5). We note that, in the case study of this paper, the core application is composed by 27 architectural elements, and when the FQAs are incorporated, 25 new elements are added to the architecture, obtaining a degree of automation of 48.08%. This value is higher than in other case studies because the ITS requires more QAs concerns, and those concerns have many dependencies between them. Thus, the results obtained by applying this metric indicates that our approach can be useful. However, an empirical study that imply SAs developing projects of different complexities using our approach needs to be performed to confirm the benefits suggested by this metric.

Degree of Separation of Concerns.

In our approach, the separated modeling of the FQAs from the core application architecture and the subsequent combination with the use of CVL and the crosscuts relationships all contribute to improve the separation of concerns. On the one hand, we have modeled each FQA separately from each other in order to (1) identify the concerns and their dependencies, and (2) avoid the duplication of concerns in different FQAs. The result is that the concerns of the FQAs are well-encapsulated only in those components that were part of the security, the context awareness, and the usability composite components (the CDAC metric for all of them is 1). The LCC metric has also been applied to the same components, and the results show that they only contain security, context awareness, or usability concerns, respectively, and the information of the required concerns of the other FQAs to satisfy their dependencies. On the other hand, this good degree of separation is kept after the incorporation of the FQAs into the application architecture due to the use of the crosscuts relationships at the architectural level.

7.1 Discussion

The evaluation results obtained indicate that the effort of separately defining FQAs forming an SPL family presents the following advantages: (i) helps the SA to identify the dependencies, take them into account and resolve them; (ii) helps the SA to create different configurations of the FQAs; (iii) helps the SA to incorporate the customized FQAs into the architecture; (iv) there is a better degree of separation of concerns due to the encapsulation of the crosscutting behavior of the FQAs in separate software components. This facilitates the subsequent modification of the application and/or the FQAs. The initial results obtained by some of the metrics (degree of dependency and degree of variability) support our decision to use techniques and tools of SPLs for modeling the FQAs. Others (separation of concerns) support our decision to model them separately from the base application. Finally, the degree of automation suggests that it is worth using our approach with regard to the effort required to generate and introduce the customized FQAs models into a software architecture. However, we need to complete the evaluation with empirical studies in order to evidence the benefits and usefulness of our approach.

Despite the benefits of using the techniques and tools of SPLs and CVL in particular, we have also identified some shortcomings to our approach. A possible disadvantage is that the SA has to deal with many models. However, we need to take into account that some of these models are just configurations of the others (e.g. the resolution models), others are automatically generated by CVL (e.g. the resolved models), and others are defined only once and are reusable in other applications (e.g. the FQAs base model and the FQAs variability model).

Another disadvantage of our approach is that the weaving rules implemented as model-to-model transformations in the last stage of our approach depend on the meta-model used to specify the core software architecture. This means that these rules will need to be adapted or redefined if the SA decides to use a software architectural model that, in spite of being MOF-dependent, does not incorporate the same meta-model constructors that we used to define the weaving rules. This is however a minor limitation in the sense that our approach enables the integration of different model-to-model transformations as part of the weaving step, by using the extension mechanism provided by CVL (i.e. the OVPs). Also note that even if the weaving rules need to be adapted,

this is the last step of our approach and all the previous steps remain without changes — i.e. the definition and the variability modeling of the FQAs as well as the configuration of them do not change.

Finally, another limitation of our approach is that we have modeled only the dependencies between the concerns of the FQAs, but there can also be dependencies between the concerns of the FQAs and the base application (e.g. the feedback concern of the usability FQA may require introducing a new panel into the graphical interface of the base application in order to show the feedback information). So, as part of our on-going work, we plan to extend the variability model to include these kinds of dependencies.

8. RELATED WORK

Most of the approaches that model QAs variability principally focus on the analysis of the QAs as non-functional requirements (e.g. cost, maintenance) in the final product of an SPL, and/or how the variations in the functional components of the application affect those QAs. For example, in [15, 16], Tawhid and Petriu propose a technique to model the commonality and variability in structural and behavioral SPL views using MDD. They add generic annotations related to a QA (e.g. performance) in a UML model that represents the set of core reusable SPL assets. Then, through model transformations, the UML model of a specific product with concrete annotations (e.g. UML profiles with stereotypes [7]) of the QA is derived, and a model for the given product is generated. Annotating the base model makes this highly related to variability specifications and prevents the reuse of both the base model of the application and the variability model of the QAs.

QADA [13] is a specific method to design PLAs by transforming systematic functionality and QAs into architectures, but this proposal do not take into account the quality requirements explicitly. RiPLE-DE [4] is a domain design process for SPL that models the FQAs variability by using FMs complemented with numerical values from the base application to evaluate and achieve the desired quality levels. However, the variability of the FQAs directly depends on the base application, avoiding the reuse of the FQAs.

In [5], the authors adopt the CVL approach to specify and resolve the variability of workflows. Then, they compose the detailed structural and behavioral design models of the chosen variants by using a Reusable Aspect Models (RAM) weaver. However, this external weaver is responsible for composing the reusable aspects instead of implementing the weaving process by using CVL as we do. Additionally, they apply the CVL approach at the design level while we focus at the architectural level (e.g. component diagrams).

Architectural patterns are also used to integrate QAs into software architectures [9]. This approach uses patterns for architectural partitioning in order to help the SA satisfies non-functional characteristics of the system. The main limitation of this approach is that a specific implementation of a pattern for a QA cannot be directly re-used in other different application architecture, and the pattern needs to be applied from the scratch.

9. CONCLUSIONS AND FUTURE WORK

We have proposed a generic, integrated and extensible approach for modeling FQAs separately from the base application architecture. By separating the modeling of the FQAs from the application architecture we have improved the modularization and the reusability of models. CVL makes our approach suitable for any MOF-compliant language and allows us to automatically generate architectural configurations of the FQAs and inject them into the application architecture by extending the semantic of the CVL variation points.

In our follow-up work, we plan to evaluate our approach with empirical studies in order to evidence its benefits and usefulness. For example, we need to quantify the problem of using many models in order to save effort for the SAs. We also plan to extend the variability model in order to include those FQAs' concerns that have dependencies with the base application.

10. ACKNOWLEDGMENTS

Work supported by the European INTER-TRUST FP7-317731 and the Spanish TIN2012-34840, FamiWare P09-TIC-5231, and MAGIC P12-TIC1814 projects.

11. REFERENCES

[1] INTER-TRUST: Interoperable Trust Assurance Infrastructure. `http://www.inter-trust.eu`.

[2] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock. Quality Attributes. Technical report, 1995. `http://resources.sei.cmu.edu/asset_files/TechnicalReport/1995_005_001_16427.pdf`.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 3rd edition, 2012.

[4] 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.

[5] B. Combemale, O. Barais, O. Alam, and J. Kienzle. Using CVL to operationalize product line development with reusable aspect models. In *VARY*, 2012.

[6] J. B. F. Filho, O. Barais, J. Le Noir, and J.-M. Jézéquel. Customizing the common variability language semantics for your domain models. In *VARY*, 2012.

[7] M. Fontoura, W. Pree, and B. Rumpe. *The UML profile for framework architectures*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[8] A. Harrington and V. Cahill. Model-driven engineering of planning and optimisation algorithms for pervasive computing environments. In *PerCom*, pages 172–180, 2011.

[9] N. Harrison and P. Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In *Software Architecture*, volume 4758 of *LNCS*, pages 263–270. 2007.

[10] O. Haugen, A. Wąsowski, and K. Czarnecki. CVL: Common Variability Language. In *SPLC*, 2012.

[11] J. M. Horcas, M. Pinto, and L. Fuentes. Variability and dependency modeling of quality attributes. In *SEAA*, 2013.

[12] N. Juristo, A. Moreno, and M.-I. Sanchez-Segura. Guidelines for eliciting usability functionalities. *IEEE TSE*, 33(11):744–758, 2007.

[13] M. Matinlassi, E. Niemelä, and L. Dobrica. *Quality-driven Architecture Design and Quality Analysis Method: A Revolutionary Initiation Approach to a Product Line Architecture*. VTT publications. Technical Research Centre of Finland, 2002.

[14] M. Pinto, L. Fuentes, L. Fernández, and J. Valenzuela. Using AOSD and MDD to enhance the architectural design phase. In *On the Move to Meaningful Internet Systems: OTM Workshops*. 2009.

[15] R. Tawhid and D. Petriu. Integrating performance analysis in the Model Driven Development of Software Product Lines. In *MoDELS*. 2008.

[16] R. Tawhid and D. Petriu. Automatic derivation of a product performance model from a software product line model. In *SPLC*, 2011.