ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/285548628

An Automatic Process for Weaving Functional Quality Attributes Using a Software Product Line Approach

Article *in* Journal of Systems and Software · December 2015

DOI: 10.1016/j.jss.2015.11.005

CITATIONS
14

READS
220

3 authors:

Jose-Miguel Horcas
University of Malaga
31 PUBLICATIONS **135** CITATIONS

SEE PROFILE

Monica Pinto
University of Malaga
101 PUBLICATIONS **1,113** CITATIONS

SEE PROFILE

Lidia Fuentes
University of Malaga
232 PUBLICATIONS **2,680** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

ClinicAppChain View project

MAGIC: Software Product Lines and Multi-Agent Systems for the self-management of the IoT View project

All content following this page was uploaded by Lidia Fuentes on 16 October 2017.

The user has requested enhancement of the downloaded file.

An Automatic Process for Weaving Functional Quality Attributes Using a Software Product Line Approach

Jose-Miguel Horcas[1,*], Mónica Pinto[1], Lidia Fuentes[1]

[a] *CAOSD Group, Universidad de Málaga, Andalucía Tech, Spain*

Abstract

Some quality attributes can be modelled using software components, and are normally known as Functional Quality Attributes (FQAs). Applications may require different FQAs, and each FQA (e.g., security) can be composed of many concerns (e.g., access control or authentication). They normally have dependencies between them and crosscut the system architecture. The goal of the work presented here is to provide the means for software architects to focus only on application functionality, without having to worry about FQAs. The idea is to model FQAs separately from application functionality following a Software Product Line (SPL) approach. By combining SPL and aspect-oriented mechanisms, we will define a generic process to model and automatically inject FQAs into the application without breaking the base architecture. We will provide and compare two implementations of our generic approach using different variability and architecture description languages: i) feature models and an aspect-oriented architecture description language; and ii) the Common Variability Language (CVL) and a MOF-compliant language (e.g., UML). We also discuss the benefits and limitations of our approach. Modelling FQAs separately from the base application has many advantages (e.g., reusability, less coupled components, high cohesive architectures).

Keywords: Quality Attributes, Software Product Lines, Aspect-Orientation, Weaving, Model Transformations

1. Introduction

The quality of a software system is measured by the extent to which it possesses a desired combination of quality attributes (QAs) [1] such as usability, confidentiality, reliability, security or scalability. Some quality attributes (QAs) can be modelled using software components and are normally known as functional quality attributes (FQAs) [2]. Examples of FQAs are security (e.g., to allow access control), usability (e.g., to provide contextual help) or error handling (e.g., to respond to the occurrence of errors and exceptions). Note that other QAs (i.e., those related to non-functional requirements) such as cost, efficiency or portability cannot be directly mapped to functional software components, but they can be mapped to architectural or implementation decisions, so they are beyond the scope of this paper.

In order to satisfy application requirements, apart from core functional and non-functional requirements, the software architect should pay special attention to those that can be modelled as FQAs. FQAs are characterised by the following aspects: (1) they are recurrent — i.e., FQAs are normally required by several applications (e.g., security); (2) most FQAs crosscut the system architecture; and (3) they require the incorporation of specialised components inside the architecture (e.g., an authorisation mechanism to satisfy the security FQA). Normally, FQAs are modelled and tailored to a given application, with functional components that are part of the core architecture. But, modelling FQAs separately from the base application has many advantages (e.g., reusability improvement, less coupled architectures, etc.). For instance, an encryption algorithm used to encrypt the information to ensure confidentiality does not depend on the application that needs it.

Modelling FQAs is a complex task, firstly because they are usually composed of many concerns. The security FQA, for example, is composed of confidentiality, integrity, access control and authentication, among others. Secondly, different applications may require different levels of an FQA (e.g., different security levels). For example, a specific application may require access

*Corresponding author

Email addresses: horcas@lcc.uma.es (Jose-Miguel Horcas), pinto@lcc.uma.es (Mónica Pinto), lff@lcc.uma.es (Lidia Fuentes)

control, encryption, and anonymity while another may require only encryption, or may require a different kind of encryption algorithm. Thirdly, some of the concerns of an FQA may have dependencies between them. For example, the confidentiality concern depends on the encryption concern to ensure that all the information is encrypted and cannot be obtained by third persons. Furthermore, some 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 to be able to offer customised help based on the user's previous experience with a given application.

To summarise, there is much variability in FQAs and different dependency levels. Therefore, specifying the set of FQA components and connections that fulfil the application requirements is not a trivial task for the software architect. Our goal is to alleviate the software architect's tasks with respect to FQAs by: (i) defining a family of FQAs with commonalities, variabilities and dependencies; and (ii) implementing a process able to automatically generate the final application architecture that includes the customised FQAs.

The variability of FQAs can be modelled by using different techniques provided by traditional Software Product Line (SPL) [3] approaches. Reviewing the literature, the conclusion can be drawn that little care is taken to model variability of the functional part of QAs [4], and normally the focus is on modelling the functional variability of the application. Some approaches propose techniques for analysing and/or reasoning about the impact of functional variants on the quality of applications derived from an SPL, principally concentrating on non-functional QAs such as performance, availability, cost, or latency [5, 6, 7]. Others address FQAs variability (e.g., QADA [8], RiPLE-DE [9]), but they model these FQAs intermingled with the functional components, as part of the domain analysis of an SPL, and not separately as we propose. The main drawback of these latter approaches is that they do not provide means to easily reuse FQAs and their dependencies across several applications, nor do they facilitate the customisation of FQAs to each individual application.

In order to define a family of FQAs following a generic SPL approach, we need a language to specify and model the variability of FQAs. According to [10] Feature Models (FMs) are the most used variability language, which model variability by means of high-level features that are close to requirements specification. More recently, the Common Variability Language (CVL [11]) was proposed as a standard. Both alternatives are currently well accepted by the SPL community, and can be used in our approach.

Independently of the variability language used, once an architectural configuration of the FQAs has been generated, a process to incorporate it into the architecture of the base application non-intrusively, is required. For weaving FQAs with the base application we will use some aspect-oriented mechanisms. By combining SPL and aspect-oriented software development (AOSD) technologies, we have defined a generic process to: (i) specify and model the variability and dependencies among FQAs, defining a reusable family of FQAs; (ii) customise the FQAs to fulfil the application requirements and automatically generate an architectural configuration of FQAs; and (iii) weave the customised FQAs into the software architecture of the base application without manually modifying it. We present and compare two instantiations of our generic process using different variability languages and architecture description languages: 1) with feature models and AO-ADL, an aspect-oriented architecture description language [12, 13]; and 2) using the Common Variability Language (CVL) and a MOF-compliant language such as UML [14]. We illustrate our approach with an e-voting case study and quantitatively evaluate both approaches by using suitable metrics (degree of dependency, variability, automation, separation of concerns) to assess the benefits of each approach. Also, we discuss the benefits and disadvantages of both implementations of our approach.

The remainder of the paper is organised as follows: Section 2 presents the challenges addresed in this work and motivates it with a case study. Section 3 overviews our approach. In Section 4 we explain in detail how we model FQAs with two different instantiations of our approach. The customisation and incorporation 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 approach, while in Section 8 we identify and discusses the benefits and limitations of our approach. Section 9 discusses the related work. Finally, Section 10 concludes the paper.

2. Motivation and Challenges

In this section we present the specific challenges addressed in this work and the motivating case study we use to illustrate our approach.

2.1. Challenges

In this section we describe the specific challenges addressed in our work related to FQA modelling and the weaving of a tailored FQA configuration into different applications.

Challenge 1. Manage the variability of FQAs and their customisation according to the application requirements. The issue of the high degree of variability in FQAs has been neglected or even ignored by most software architects as attention has mainly focused on functional variability [9]. The challenge is to model all the possible FQA variation points independently of the final application, which is not a trivial task. *In this paper we define a family of FQAs, following an SPL approach, that supports the customisation of FQAs to satisfy the specific requirements of each application. We provide a process that configures FQA variation points in such a way that variable concerns that are not required by the base application are not incorporated into the final application. In this paper we explore the use of both FM and CVL languages to specify a family of reusable FQAs.*

Challenge 2. Manage dependencies between FQAs. In FQA modelling, dependencies between concerns that are part of the same FQA need to be taken into account, — i.e., *intraFQA-dependencies*. Furthermore, dependency relationships between different FQAs must also be considered, — i.e., *interFQA-dependencies*. These kinds of dependencies often go unnoticed by software architects, who are not domain experts in modelling all types of FQAs. *Using the support provided by the SPL approach these dependencies are automatically traced and incorporated into the solution even if they have not been explicitly selected by the software architect. For example, if a concern X depends on other concerns Y, W and Z, then these other concerns should be automatically incorporated into the solution even if they have not been explicitly selected by the software architect.*

Challenge 3. Define architectural patterns with reusable FQAs. Once the FQAs have been modelled as independently as possible, the customised FQAs need to be woven with the final application. 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 either understand the inner working of the quality solutions, or break the application's core architecture, — i.e., architecture components should be completely unaware of the FQAs they are affected by. This is not a straightforward task since each FQA needs to be woven at different points of the base applications. Moreover, multiple views may be required to appropriately model the FQAs (e.g., behavioural view, structural view), and this makes the weaving process even more complex. *As part of our work we define different architectural weaving patterns, following the non-invasive weaving mechanism of aspect-orientation. For this we follow two different approaches: (i) use connector templates defined by the AO-ADL language; and (ii) use CVL and implement the corresponding model transformations.*

Challenge 4. Support the approach with tools. The approach presented in this paper is not viable without the required tool support. *In our approach we combine several tools for SPL and AOSD in order to completely automate the process of: (1) generating customised software architectures for the FQAs required by an application, and (2) weaving these software architectures with the software architecture of the core functionality of the application.*

2.2. Motivating Case Study

We illustrate our approach with a case study of an online electronic voting (e-voting) application. We have choosen this application because it is easy to understand, is complex enough to show the details of our approach and is, moreover, an industrial case study. In fact, it is one of the demonstrators of the INTER-TRUST project[1], the main motivation of which is separating security related concerns from the application base code. With this project, the industrial partners demand easily instantiable security solutions as part of any distributed application.

E-voting is one of the environments where FQAs requirements are complex. Figure 1 shows a simplified software architecture in UML with the main functionality of an e-voting application. The `Voter Client` component allows clients to cast their votes from different devices (e.g., smart phones, tablets) to the digital ballot box (`Voting Ballot` component). The `Voting Server` component receives the votes and the `Votes Storage` component stores them. Administrators can manage the election data and get the election results through the `Admin Client` component that provides access to the managing functionality of the system (`Voting System` component).

Apart from the base functionality shown in Figure 1, the e-voting application requires strict security restrictions. Concretely, it is of paramount importance to guarantee that: (1) the privacy of the voter must be preserved

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

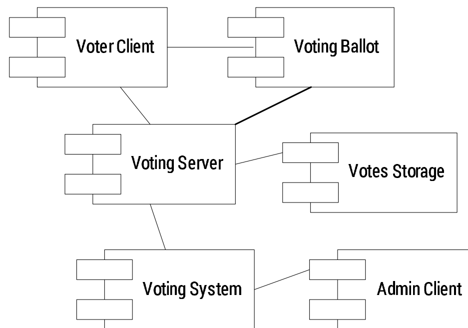


Figure 1: E-voting software architecture.

as well as the confidentiality of the votes; (2) at the same time none of the votes in the digital ballot box can be altered or lost during transmission (i.e., integrity of the votes); (3) the voter must be authenticated using a personal digital certificate, such as an electronic ID card; and (4) administrators must be authorised to perform actions over the election data and to obtain a list of the users authenticated in the system. Further to these security requirements, the e-voting process needs to know the location of the voter for statistical reasons. Moreover, the application must provide contextual help to the user according to their needs and also provide feedback information (e.g., alerts when the battery level of the device is too low). Summarising, from the textual requirements the software architect can deduce that the e-voting application requires, at least, the following FQA concerns:

Security: privacy, confidentiality, integrity, and authentication.

Context awareness: location aware (e.g., GPS, WIFI location) and device aware (battery status).

Usability: contextual help and feedback (alerts and history log).

3. A generic process for managing FQAs

In this section we present a general overview of our approach (Figure 2) that extends the classic framework for SPL engineering [3] as follows:

- **Domain Engineering** (top of Figure 2). The goals of our domain engineering process are: (i) to define the commonality and the variability of the FQAs, and (ii) to construct a reusable FQA software architecture that accomplishes the desired variability.

  The requirements from the FQA domain are taken as input for this process. From this input, domain experts identify the commonality and variability of the FQAs as well as the existing dependency relationships. These include the intraFQAs-dependencies and the interFQAs-dependencies. The FQAs analysis allows a variability model to be specified and defined for the FQAs and a software architecture to be constructed that supports the variability. Reusable architectural patterns for weaving the FQAs are also specified by the domain experts in order to define how the different FQA concerns should be composed with the core architecture of the base application. The output of the process is a product roadmap that determines the major common and variable features of future FQAs architectures. An important thing that is worth highlighting is that this process is performed only once. This means that the FQAs software architecture and the FQAs variability model will be completely reused by any application that wants to configure and incorporate these FQAs into its software architecture.

  Variability can be expressed through multiple techniques such as FMs [15], annotations [16, 17] or by using a variability language such as CVL [18]. In this paper we use FMs and CVL.

- **Application Engineering** (middle of Figure 2). The goal of the application engineering process is to bind the FQAs variability according to the application requirements. To do this, the application architect identifies the FQAs required by the application and creates a product configuration of the FQAs that fulfils those requirements. Then, an FQAs architecture configuration is automatically generated as the realisation of the product configuration. The output of this process is an architecture configuration of the FQAs that only contains those artefacts of the FQAs software architecture that are needed according to the application requirements.

- **Weaving** (bottom of Figure 2). The goal of the weaving process is to incorporate the FQAs architecture configuration generated in the previous process into the core architecture of the application being built. The core architecture only contains the base functionality of the application without any elements related to the FQAs requirements.

  The weaving process is not a straightforward task since each FQA has to be woven at different points of the base applications (join points). Further-

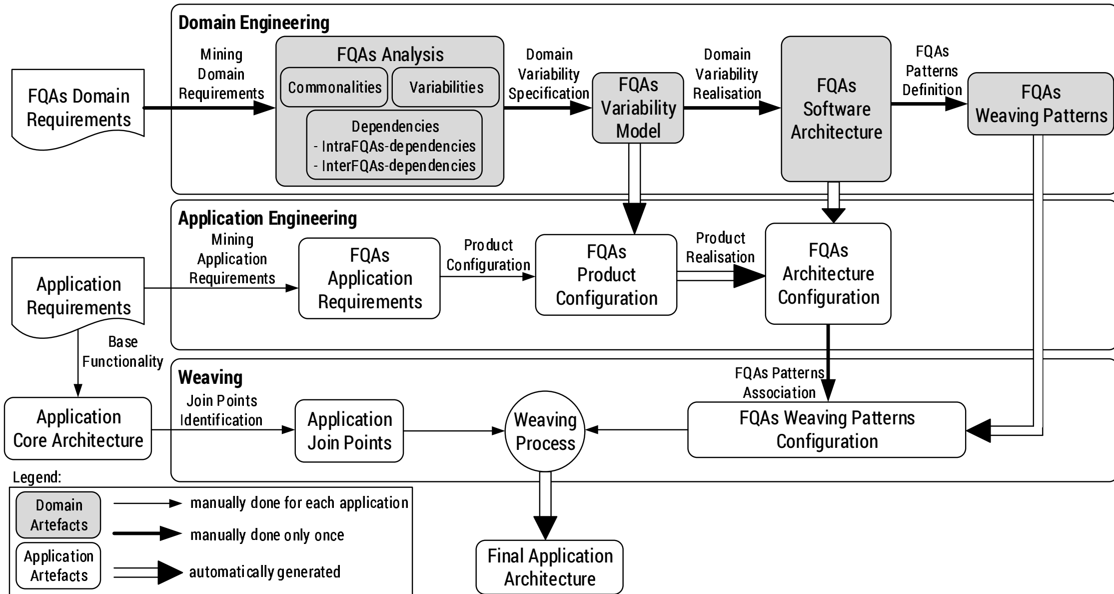


Figure 2: Generic process for weaving FQAs customised for an application.

more, each FQA concern will be woven according to a different weaving pattern, depending on the semantics of the concern. For instance, the authentication concern is usually woven before the application join points in order to authenticate the user (or check whether or not they are authenticated) before executing the target join point. So, the application architect must identify the points in the application where the FQA concerns will be incorporated, and associate the set of FQAs weaving patterns provided as part of the FQAs domain engineering process with the customised components of the FQAs architecture. Also, the weaving process should be done automatically, without manually modifying the core architecture of the application. The output of this weaving process is a software architecture of the application that also incorporates the required FQAs.

4. FQAs Domain Engineering

This section describes the first phase of our generic process, in which the FQAs, their commonalities and variabilities, the dependencies between them, and the weaving patterns of the FQA concerns are identified and modelled. As we have already said, we present two instantiations of our generic process: (1) using FMs, AO-ADL and an external Variability Modelling Language (VML) (see Figure 3); (2) using CVL and any MOF-based language (see Figure 4).

4.1. FQAs Analysis

The FQAs Analysis sub-process encompasses all activities for eliciting and documenting the common and variable requirements of the FQAs [19]. The list of documented FQAs is very long [20, 1, 2]. For instance, Juristo et al. [2] identify a list of functional usability concerns such as feedback, contextual help, undo and cancel operations, shortcuts, etc. By analysing this information, domain experts can classify the FQAs (e.g., usability) and their concerns (e.g., feedback, contextual help) and identify their common and variable parameters. Domain experts must also take into account the different kinds of dependencies between the FQAs, like for example, contextual help implies authentication. All this information must be formally specified using a variability model.

4.2. FQAs Variability Model

With the domain information captured during the analysis of the FQAs, domain experts define a variability model of the FQAs. We start with feature models. The top of Figure 3 shows a partial view of the feature model representing FQAs, depicting only some of the FQAs (i.e., security, persistence, context awareness and usability). For reasons of space we do not include either all FQAs or all possible features, but the idea is that this variability model covers all FQAs independently of the applications, so it should be extended for new FQAs or concerns when neccesary. For instance,

note that we have also considered the persistence FQA despite the fact that it is not required by the e-voting case study. We use our FM tool Hydra[2] for editing and instantiating FMs. We group all FQAs in the same tree, and define one child feature per FQA. The concerns related with a FQA (e.g., `Feedback`) are specified as a subtree of the FQA feature (e.g., `Usability`). Features can be mandatory (as `TraceFile` of `Usability`) or optional (as every FQA such as `Persistence`). Hydra also defines groups of features (as temporal, database or file storage alternatives for persistence. Dependencies between FQAs are specified as dependencies between features, which are called *cross-tree constraints*. In Hydra cross-tree constraints are expressed in a textual way using a combination of regular expressions over features (e.g., negative features, all features, etc.). Figure 3 shows some examples such as `BatteryStatus implies TimeAware and Alerts`, which must be read as "the selection of the `BatteryStatus` feature implies the automatic selection of the `TimeAware` and `Alerts` features".

An alternative to FMs is to use the VSpecs tree of CVL[3]. FMs and VSpec trees have similar expressivity. Figure 5 shows an excerpt of the security FQA that states that, if the security feature (called VSpec in the CVL language) is selected, then at least one of the child features or security variation points (e.g., `Confidentiality`, `Authentication`, etc.) must also be selected. Moreover, security concerns or variation points are composed of other sub-concerns. For instance, there are different kinds of authentication: user + password (`UserPassAuth`), intelligent card (`CardAuth`), and biometric (`BioAuth`). The cross-tree constraints in a VSpec tree are specified by propositional constraints in OCL (represented as a parallelogram).

Now let us see how to connect the variability model with the FQAs software architecture (i.e., domain variability realisation). Feature Models benefit from having a formal basis [15] and they are well-supported by tools that make it possible to formally reason about variability and to manage the product generation phase easily. But features represented in an FM are close to requirements specification so an additional process is required in order to generate an architectural configuration that meets an FM configuration. We need something extra to establish this correlation and link the features to the architectural elements. In [12, 13] we proposed using a separate variability language (e.g., VML [18][4]). Following a negative variability approach we start from a software architecture with the complete functionality of the FQAs and using VML we specify the actions to (see middle of Figure 3): (1) modify this complete architecture by removing the elements that do not need to be there when a feature is not selected in a particular configuration of the feature model (e.g., if `Persistence` is not selected, then we have to remove the persistence component and its connections with `Usability` and `Security` related components), and (2) to instantiate or assign values to the parameters of the architecture when a particular alternative is selected from the feature model (e.g., the parameters of the `HistoryLog` component when the `Logs` feature is selected from the feature model). Note that arrows linking the FM, the VML program code and the architecture are there to help explain the relationships between them, and are not part of any language or model.

To the contrary if we use CVL, the use of VML is no longer needed since the variation points of the VSpecs tree have explicit links to elements of the FQA software architecture. For example, in Figure 4 the Security concern modelled as `Security_Cv` Vspec is defined as the `Security_CU` variation point that is linked to the `Security` architectural component. Another advantage of CVL is that the complete variability model of the FQAs can be divided and modularised in different levels of detail and thus, it is possible to model each FQA separately in different variability models (like `Security` in Figure 5) and then relate those models defining a complete variability model, that includes all the FQAs, with their dependency relationships (i.e., the interFQAs-dependencies) (see Figure 4). The CVL variability models of other FQAs are detailed in [14].

4.3. FQAs Software Architecture

The FQAs software architecture models the complete functionality of the different FQA concerns. This means that all the architectural elements (components, interfaces, connectors, ports, etc.) that cover all the FQA concerns must be defined. This architecture can be specified in any Architectural Description Language (ADL). For instance, the bottom part of Figure 3 presents the FQAs software architecture modelled in AO-ADL (see a complete definition in [21]). AO-ADL is an Aspect-Oriented Architecture Description Language that provides support for separating and injecting crosscutting concerns in a non-intrusive way at the

[2] The Hydra tool, `http://caosd.lcc.uma.es/spl/hydra/`

[3] The CVL tool, `http://modelbased.net/tools/cvl-2-tool/`

[4] Visit the web page of our group `http://caosd.lcc.uma.es/aoadl/`

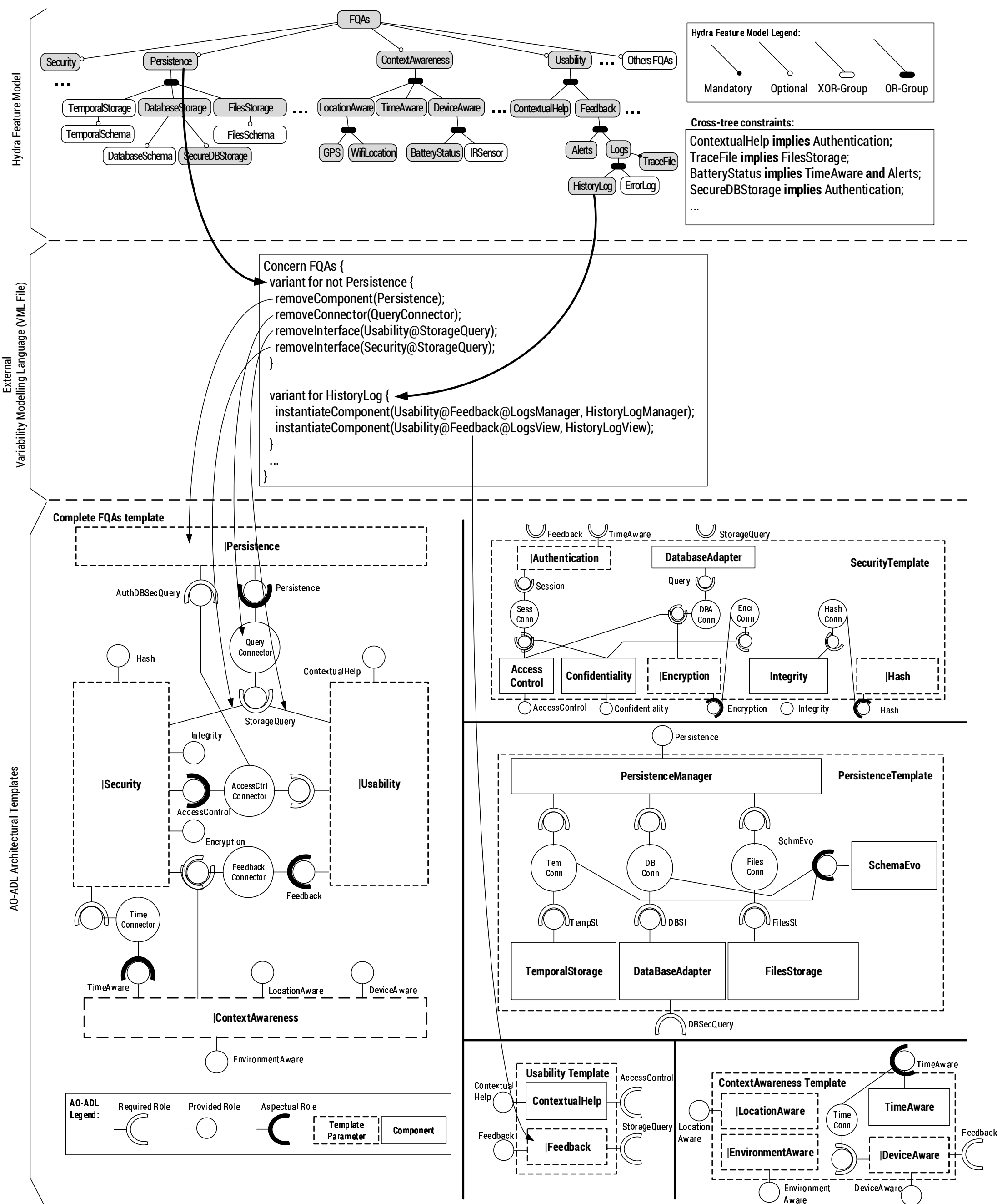


Figure 3: Domain Engineering of the FQA family using FMs, AO-ADL and VML languages.

architectural level. The main entities of AO-ADL are components, connectors, required and provided roles (i.e., roles are special connector interfaces in AO-ADL), and an aspectual role to connect components modelling crosscutting concerns (i.e., advice in AO terminology). Components modelling crosscutting concerns that are injected at different architectural join points, are attached to an aspectual role. AO-ADL also provides support for modelling parameterised architectural patterns by defining an AO-ADL architectural template [22]. AO-ADL is completely supported by the AO-ADL Tool Suite[5]. We define the software architectures with two

[5] http://caosd.lcc.uma.es/aoadl/index.htm

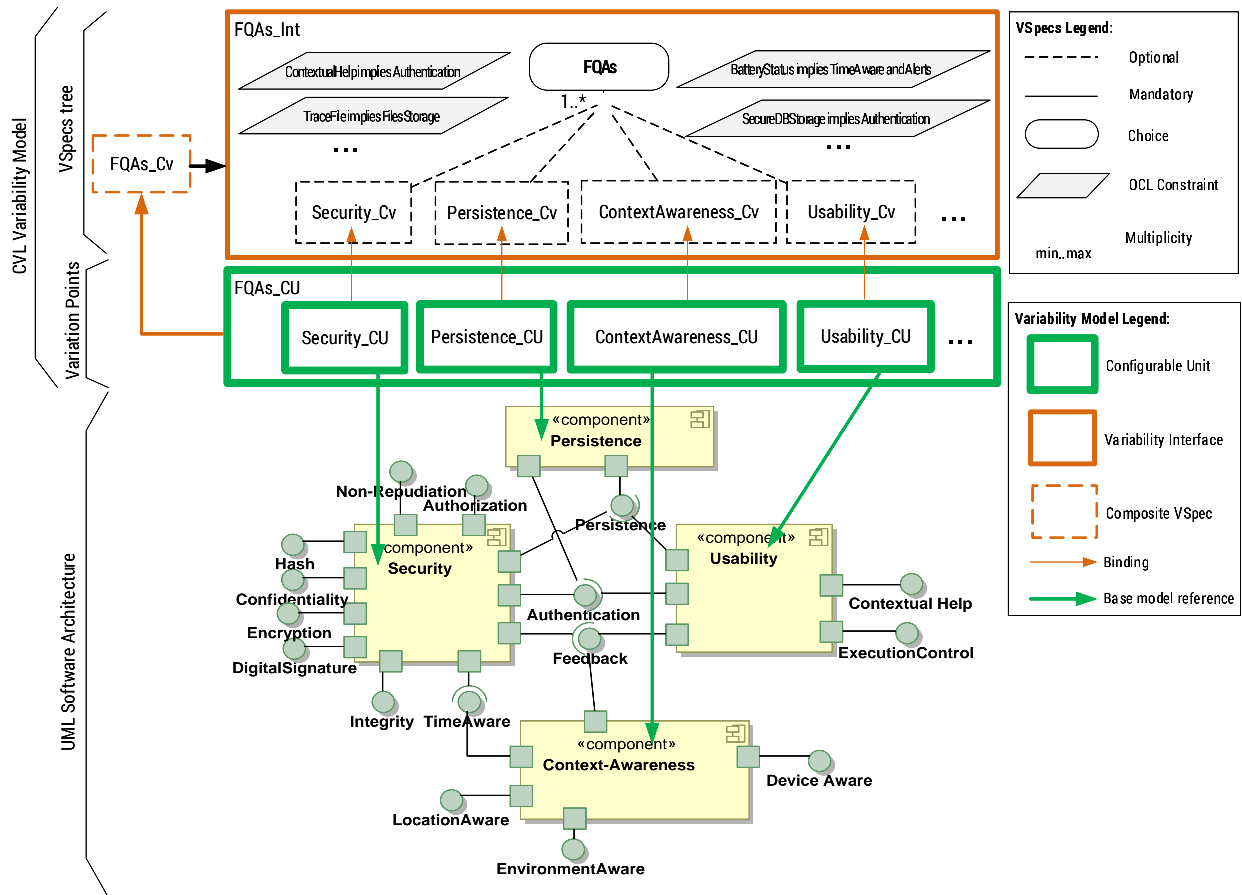


Figure 4: Complete variability model of the FQAs in CVL.

levels of granularity. In the first level, shown on the bottom left of Figure 3, there is a composite component representing each FQA. For example, `Security` and `Usability` are composite parameterised components (i.e., sub-templates in AO-ADL terminology) that will be bound at the application engineering phase to concrete components. Circles represent connectors, that in this case, are used to represent the interFQA-dependencies at the architectural level. For instance, "usability requires persistence" and this dependency is represented by the `QueryConnector` connector and the `StorageQuery` required interface (i.e., role in AO-ADL terminology). The bottom right of Figure 3 shows the AO-ADL templates (i.e., parameterisable composite components) modelling each FQA in detail. Elements defined as parameters are related to OR features in the feature model (e.g., `|Authentication` in the `SecurityTemplate`), which have to be instantiated later, for a specific application (e.g., with a concrete authentication mechanism).

Regarding the CVL case, one of the most important benefits of this language is that the product line architecture can be specified in any MOF-compliant language. In our approach we decided to use the widely known standard UML. In the UML full version of the architecture, each FQA is modelled with a composite component and the inter-FQA dependencies are modelled using provided-required interfaces. This architecture for the security FQA is defined at the bottom of Figure 5, where each concern (e.g., `DigitalIdentity`) is modelled with a UML component and dependencies are modelled using classical provided-required interfaces. For example, we define a cross-tree constraint which says that "confidentiality requires encryption" and this is represented in UML with a provided-required interface between both components. The main difference between UML and AO-ADL is that with AO-ADL this dependency is modelled by the `EncrConn` connector that provides the encryption functionality through a special AO-ADL interface called *aspectual role* (in black).

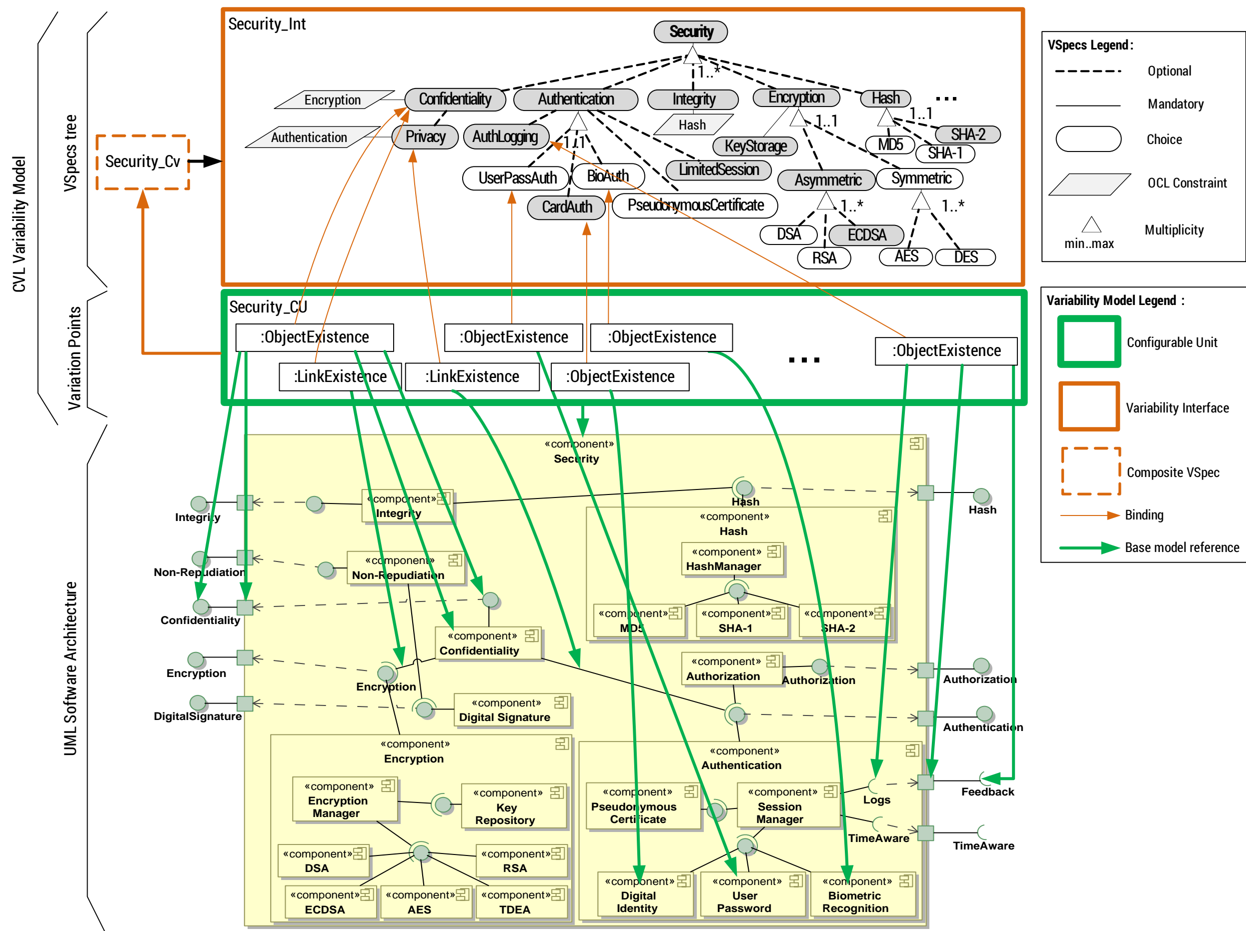


Figure 5: Domain Engineering of the Security FQA using CVL and UML.

In practice, this aspectual role means that we can modify the encryption interface (e.g., adding new encryption algorithms) without modifying the connection (i.e., interfaces) between `Encryption` and `Confidentiality` components. To do this in UML, we would need to manually modify the relationships between the encryption interfaces.

4.4. FQAs Weaving Patterns

The approximation of first separating the crosscutting functionality (i.e., the crosscutting FQAs in our approach) as an independent model and then weaving it with the base components affected by these crosscutting behaviours (i.e., the software architecture of the base application without FQAs in our approach) is typically followed by Aspect-Oriented Modelling approaches (AOM)[6]. In fact, AOM approaches have as their goal to allow a modeller to separate crosscutting behaviours in the detailed design development stage, and then the *model weaving* is carried out between base and crosscutting models (i.e., *aspectual models* in AO terminology). We follow the same model weaving approach in order to weave the FQAs architecture configuration and the application architecture in the application engineering phase (see Section 5). But, this is not a trivial task for two main reasons: (1) each concern in the FQAs architecture has to be woven at different points of the base applications (join points) and, (2) each FQA concern will be woven according to a different weaving pattern, depending on the semantics of the concern. For example, authentication is required to be woven before the user vote, but it is also required in other join points, for instance around the contextual help. In AO terminology this is defined as FQAs (i.e., crosscutting models) are composed before, after or around the base model (i.e., different kinds of *advice* in AO terminology).

[6] `http://www.aspect-modeling.org/`

Table 1: Weaving Patterns.

Weaving Pattern	Description	Example
WP1	Only one advice of a concern is woven into a join point.	**Authentication**: the `authenticate()` advice is performed *around* the join point.
WP2	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.
WP3	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.
WP4	Multiple advice methods of the same concern are woven into a join point.	**Feedback**: `log()` advice is invoked *before* and then *after* the join point.
WP5	Multiple advice methods 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.
WP6	Advice methods 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.
WP7	Advice methods 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* using the information in the server.

In our approach, the crosscutting functionality of each FQA concern is encapsulated in a software component. Let us consider that we want to inject one or several behaviours (i.e., concerns of each FQA) or "aspectual components" (following the AO terminology) between two components that are connected through a certain connector (e.g., a provided-required interface). To simplify, we consider that aspectual components implementing each FQA concern only provide one interface, and the possible join points are the methods of the interface. This simplification is reasonable since aspectual components are supposed to implement only one concern to avoid having entangled functionality (see Figure 5). In addition, each aspectual component interface can implement one or several methods (or advice in AO terminology). For instance, the digital signature concern has only one advice: the `sign(Object)` method; while the encryption concern has two: the `encrypt(Object)` method and the `decrypt(Object)` method. Additionally, each advice incorporated into a join point can be executed at a different time: *before*, *after*, or *around* the join point.

Table 1 summarises the weaving patterns (WPs) defined. The first three WPs refer to the application of one advice at one join point, once or several times or applied in several join points. WP4 and WP5 specify the application of multiple advice of the same concern in one or in several join points. And the last two WPs define the advice weaving of different concerns into one or several join points. Combining two or more of these weaving patterns we cover all the weaving possibilities. In Section 6 we show how we have implemented these generic weaving patterns in AO-ADL and in UML architectures. The specification of the kind of the advice (before, after, and around) makes sense in the structural view because our intention is to model that the crosscutting behaviour modelled by the FQAs is affecting specific methods of the component's interface, and not the component's interface as a whole. So the advice will be injected before, after or around that specific method. In the case of AO-ADL, the target method where the advice is applied and the kind of the advice are specified in XML inside the aspectual binding of the associated connector (see Section 6.1). In the case of CVL, a «crosscutts» relationship is specified between the advice of the FQA and the target method of the application's interface in the structural view, and additionally, a sequence diagram is automatically generated to specify when the advice is applied (see Section 6.2).

5. FQAs Application Engineering

This section describes the second phase of our generic process (Figure 2), in which a valid configuration (customisation) of the FQAs variability model is generated, taking as input the specific requirements of the application under development (in our case, the e-voting case study of Section 2.2).

5.1. Creating a Product Configuration

A product configuration in SPL approaches is defined as the set of features that satisfy the application requirements. In this task of the process, the software architect should map the high-level application requirements and the features (or VSpecs) present in the variability model (i.e., feature model or VSpec tree). For example, in Section 2.2 it is stated that the e-voting process needs to provide real-time information about the users that are voting. This requirement helps in the usability of the application for the administrator, by providing him/her with a rapid graphical visualisation of the users that are voting at all times. To include this specific FQA in the application, the software architect simply has to select the `HistoryLog` concern of the usability FQA in the variability model.

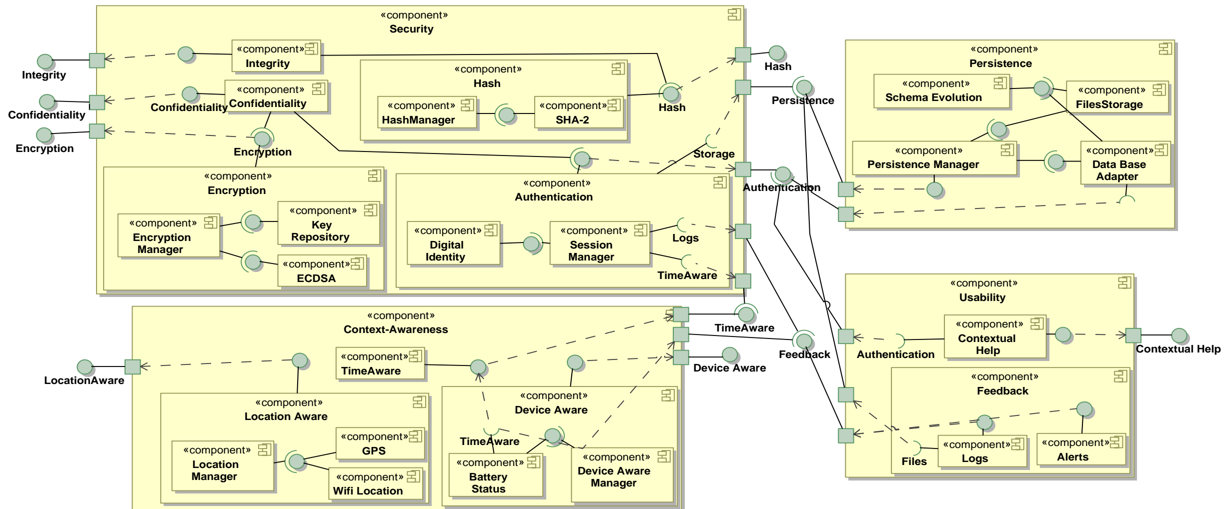


Figure 6: FQAs architecture configuration (resolved model of the FQAs).

Formally, a configuration of a feature model (called a *feature model configuration*) is a new tree resulting from the selection of a set of features that satisfy the application requirements. A configuration is valid if all features contained in the configuration and none of the features excluded by cross-tree constraints are present. In order to check whether or not a configuration is valid, Hydra uses a Java library for Constraint Satisfaction Problems (CSP) called Choco[7]. Hydra generates the minimal configuration, with the least number of features that satisfy the initial constraints. In the CVL approach, a product configuration is known as the *resolution model*, which is created by deciding which choices of the VSpec tree are positively decided and which ones are negatively decided (see selected features in grey at Figure 3 and Figure 5 for the e-voting case study).

In both approaches, the generation of a configuration is automatically done by a tool. In our approach this tool will automatically check parent-child dependencies between concerns and sub-concerns expressed as tree constraints; and interFQA- and intraFQA- dependencies expressed as cross-tree constraints. This will alleviate the software architect's task in selecting each of the concerns needed by the application. Coming back to our example, the requirement "the privacy of the voter must be preserved as well as the integrity of the votes" determines the selection of privacy and integrity features. But, privacy requires authentication (intraFQA-dependency) and it is a child of confidentiality that depends on encryption (other intraFQA-dependency), while integrity requires hashing, so although the requirements would not have explicitly stated that the e-voting application requires encryption and hashing, they will be selected automatically by the FM or CVL tool.

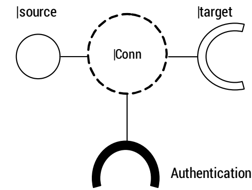
Moreover, there are other dependencies that are not so evident to the software architect, especially the interFQA-dependencies. In our case study, note that, because of the definition of the cross-tree constraint `TraceFile implies FilesStorage` which defines a dependency between a usability concern (i.e., `Logs/TraceFile`) and a persistence concern (i.e., `FileStorage`), the `FilesStorage` concern is automatically selected as part of the configuration. Note also that persistence was not initially a requirement of the e-voting application, but it needs to be selected in order to obtain a valid configuration. The FQA product configuration not only has the features selected by the software architect, but also those that depend on the selected ones. This guarantees that only the minimal but necessary set of concerns considering both the interFQA- and intraFQA- dependencies are present in the resulting configuration. Also, regarding those concerns that have to be selected due to the parent-child relationship, the tool will not generate a valid configuration until the software architect selects a specific value for each branch. For example, if the software architect selects the encryption FQA (or even if it is selected automatically due to a dependency relationship), the tool will not generate a valid configuration until the software architect selects a specific encryption algorithm (e.g., `Encryption/Asymmetric/ECDSA`, see dark features in Figure 3).

[7] `http://choco-solver.org/`

5.2. Product Realisation

Once the FQA product configuration has been obtained, the automatic process must generate an architecture configuration. A FQAs architecture configuration consists of the set of components and connections that realise or implement the features that are part of a feature configuration (or resolution model in CVL). In the case of FMs, the tool reads the feature configuration obtained with Hydra and for each feature applies the corresponding VML rule to automatically generate an AO-ADL architecture of the FQAs required by the application. For example, for the `HistoryLog` feature the second VML rule of Figure 3 is applied so the `HistoryLogManager` and `HistoyLogView` components are added to the final architecture.

In the case of CVL, the same variability model includes the variation points that indicate how the variability expressed in the VSpec tree is materialised inside the FQAs architecture. The variation points need to be bound to elements of the VSpec tree and need to refer to elements of the FQAs architecture. For instance, the variation point (`:ObjectExistence`) bound to the `Confidentiality` concern in the `Security_Int` VSpec indicates that if `Confidentiality` is selected in a resolution model, the related elements (the `Confidentiality` component and the associated interfaces and ports with their attachments) in the architecture will exist in the final architecture configuration. If `Confidentiality` is not selected those related elements will be removed from the FQAs resolved model. Similarly, the variation point (`:LinkExistence`) bound to the `Privacy` feature represents the dependency between the privacy (part of the `Confidentiality` component) and the authentication concerns. The variation point indicates the existence of that particular link. If `Privacy` is selected in the resolution model the link will exist in the resolved model, and if `Privacy` is not selected, the link will be removed. The CVL tool will generate the resolved model of the FQAs, which is shown in Figure 6, and only includes the necessary functional components. This customised FQA architecture is now ready to be woven with the software architecture of the core application, as described in next section.




```
1.   <aspectual_binding name="Authentication_AB">
2.    <pointcut_specification>
3.     <pointcut>
       (//provided_role[@name='++=tp.getXPathElement
       (doc, $$$//instance/provided_role[@name='source']
       /instance_name$$$)--` ]) and (//operation[@name='*'])
4.     </pointcut>
5.    </pointcut_specification>
6.    <binding order="last" operator="around">
7.     <aspectual_component aspectual_role_name="Authentication">
8.      <advice label="authenticate" />
9.      <attachment>
10.      <argument_binding target="user [OBJECT]" />
11.      <argument_binding target="BOOLEAN [returnType]" />
12.     </attachment>
13.    </aspectual_component>
14.   </binding>
      ...
15. </aspectual_binding>
```

Figure 7: Connector template for Authentication.

6. Weaving the FQAs and the application architecture

At this point, an architectural model with the required FQAs and concerns has been generated. Now, in the third process of our approach (lowest part in Figure 2) this FQAs architecture configuration has to be incorporated into the software architecture of the core application (Figure 1).

6.1. AO-ADL: Weaving an FQA configuration using AO-ADL connector templates

As AO-ADL is an aspect-oriented architecture language, the aim of this language is to perform an aspect-oriented weaving, which means, without modifying existing components of the base software architecture. In order to do that, AO-ADL extends traditional connectors with *aspectual roles* (roles filled in black color in Figure 7). An aspectual role is a special role where components that encapsulate crosscutting concerns are attached by defining an aspectual binding section. These components that play the role of "aspects" (in aspect-oriented terminology) are called in AO-ADL *aspectual components*. In current work, FQAs (i.e., persistence, usability, etc.) are considered to be aspectual components. In order to incorporate these aspectual components into the base application, we need to define how to connect them to the application's existing connectors. Instead of having to manually modify the base application's connectors, the addition of the aspectual component is automated via the definition of an AO-ADL *connector template* (see XML code in Figure 7). A connector template is a connector that has one or more aspectual bindings defining the interactions between the aspectual component attached to the aspectual role and the components of the core applications — i.e., the source and target parameters (see top of Figure 7). The

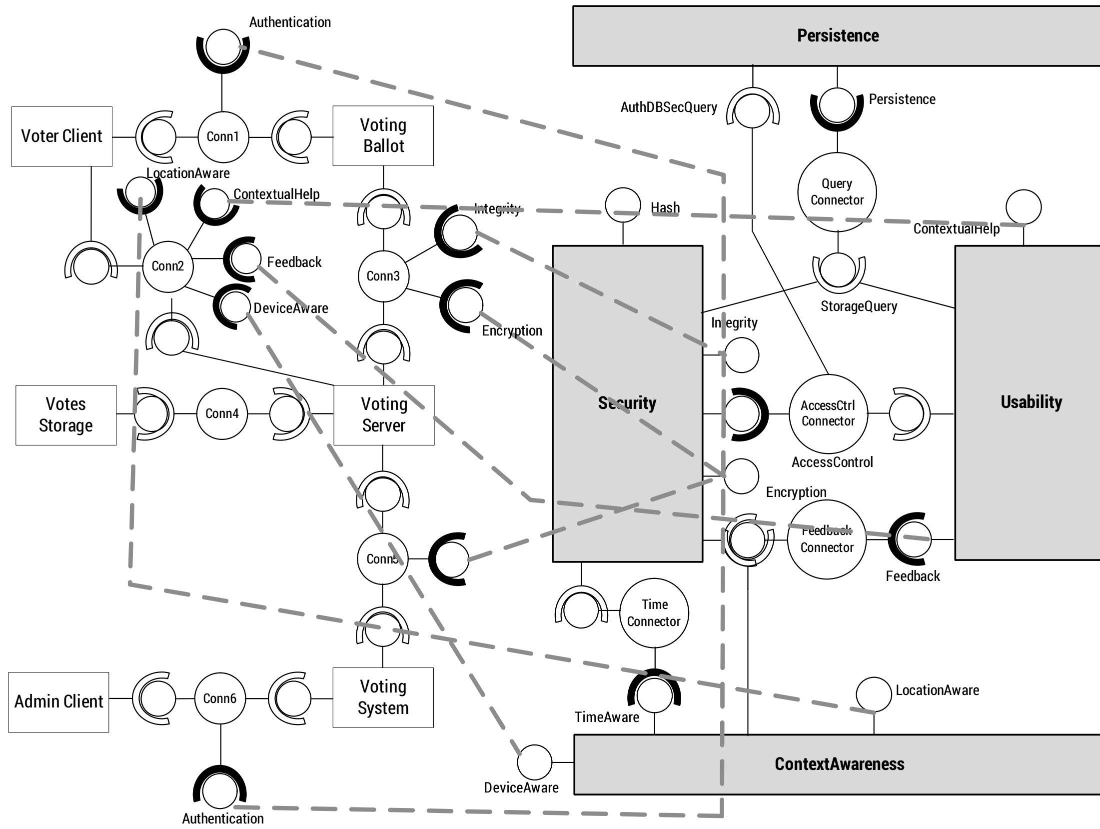


Figure 8: Complete application architecture with the FQAs woven using AO-ADL connector templates.

connector template instantiation consists of binding the source and target parameters to source and target components of the core application affected by aspectual components. Afterwards, the AO-ADL Tool Suite automatically adds a new connector (or modifies an existing one) with the aspectual role and the aspectual bindings defined in the connector template. We exploit this powerful characteristic of AO-ADL connector templates to define the weaving between the application configuration and the instantiated FQA architecture.

In our process, the variability designer must manually define a connector template for each interaction type where FQAs could be applied, according to the weaving patterns defined in Table 1. For example, authentication is incorporated using WP1 (only one advice woven around a join point), so the aspectual binding defined for the authentication connector template (see XML code in Figure 7) says that the authentication aspectual component will be attached to the `‘‘Authentication’’` `aspectual_role_name` (line 7) and the join point captured is defined by the pointcut section of this connector. Concretely, the pointcut says that any incoming message (`//operation[@name='*'` at line 3) will be intercepted by the aspectual component attached to the “Authentication” aspectual role, and the `authenticate` advice (line 8) will be injected `around` (line 6) the incoming message. At this stage of the process, the software architect has: (i) to identify those points of the base architecture where some FQAs are required; and, (ii) instantiate the appropriate connector templates defined for the required FQAs, by binding the source and target parameters to the provided and required interfaces of the affected components. One advantage of AO-ADL templates is that once they have been defined, they can be automatically instantiated and fully reused by the software architect. Moreover, if the software architect wants to make any modifications, for instance, change the advice type, this can be easily done and a new connector template will be added to the AO-ADL template repository. For example, if the software architect prefers to apply authentication “before” and not “around” a join point, he/she only has to modify the XML file of the template and substitute the term `around` with `before` (line 6).

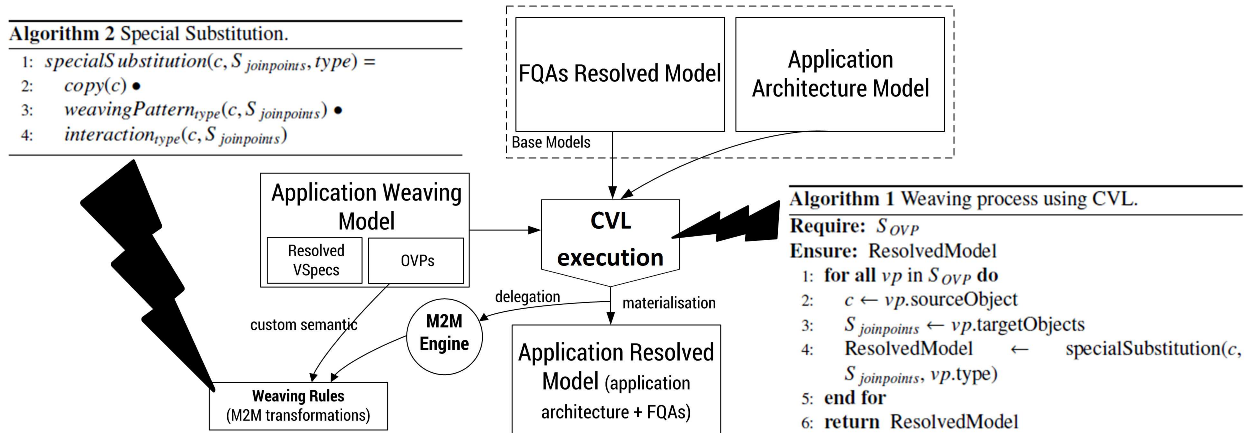


Figure 9: FQA weaving schema with CVL and model-to-model transformations.

Figure 8 shows the result of instantiating all the connector templates for the e-voting application. The connectors that have been instantiated (automatically modified by the AO-ADL tool) are those that have an aspectual role (in bold): `Conn1` (`Authentication` was automatically added); `Conn2` (idem for `LocationAware`, `ContextualHelp`, `Feedback` and `DeviceAware`); `Conn3` (idem for `Integrity` and `Encryption`), `Conn5` (idem for Encryption) and `Conn6` (idem for `Authentication`).

6.2. *CVL: Weaving an FQA configuration with model transformations*

In the case of CVL, we have MOF-compliant diagrams (UML in this paper) for representing the application architecture and also the FQAs architecture configuration. The challenge here is to extend the UML (or MOF-compliant metamodel) application architecture by injecting new UML artifacts in specific points, without manually modifying the base architecture. In Section 5 we have already defined a set of weaving patterns that specify the different types of model injection that must be implemented to support this part of the process. In this section we will show how we have implemented the weaving patterns presented in Table 1 as model transformations following an AOM approach.

6.2.1. *CVL materialisation of the application resolved model with FQAs*

Figure 9 shows a general schema of our weaving approach. From the last phase of the process we have the FQA resolved model (FQA architecture customised for the application) that we want to inject into the application architecture model following the weaving patterns already defined. In order to accomplish this automatically, and without modifying the base architecture, we take advantage of the *Opaque Variation Points* (OVPs) defined by CVL. The way that we use the OVPs to implement the FQA weaving is the main contribution of the CVL part of our approach. OVPs allow us to define a new semantics variation point using model transformation rules. During variability resolution, the CVL engine (see Figure 9) will delegate its control to a M2M transformation engine (e.g., ATL) whenever it encounters an OVP. The M2M transformation engine executes the semantics specification associated with the underlying OVP and resolves the variability accordingly.

Thus, in our approach (see Figure 10), we: (1) bind an OVP to each of the FQAs in the resolution model (e.g., `Privacy` concern of the VSpec tree is bound to `OpaqueVariationPoint1`); (2) refer to the specific architectural element that models that concern in the resolved model of the FQAs (e.g., `OpaqueVariationPoint1` refers to the `Confidentiality` provided interface of the FQAs resolution model marking it as the `sourceObject`); and (3) indicate how the concern will be woven with the application architecture model. In order to do this, each OVP is also bound to an OVP `type`, that explicitly defines the semantics of a special substitution (e.g., `SpecialSubstitutionAuth`, the semantics of which is specified in `SemanticSpec1`). Based on this special substitution the software architect needs to create a reference to one or more join points in the base application model where the behaviour of the selected concern will be incorporated (e.g., the interface that connects the `Voter Client` with the `Voting Ballot` component is marked as the `targetObject`). In Figure 10 we only

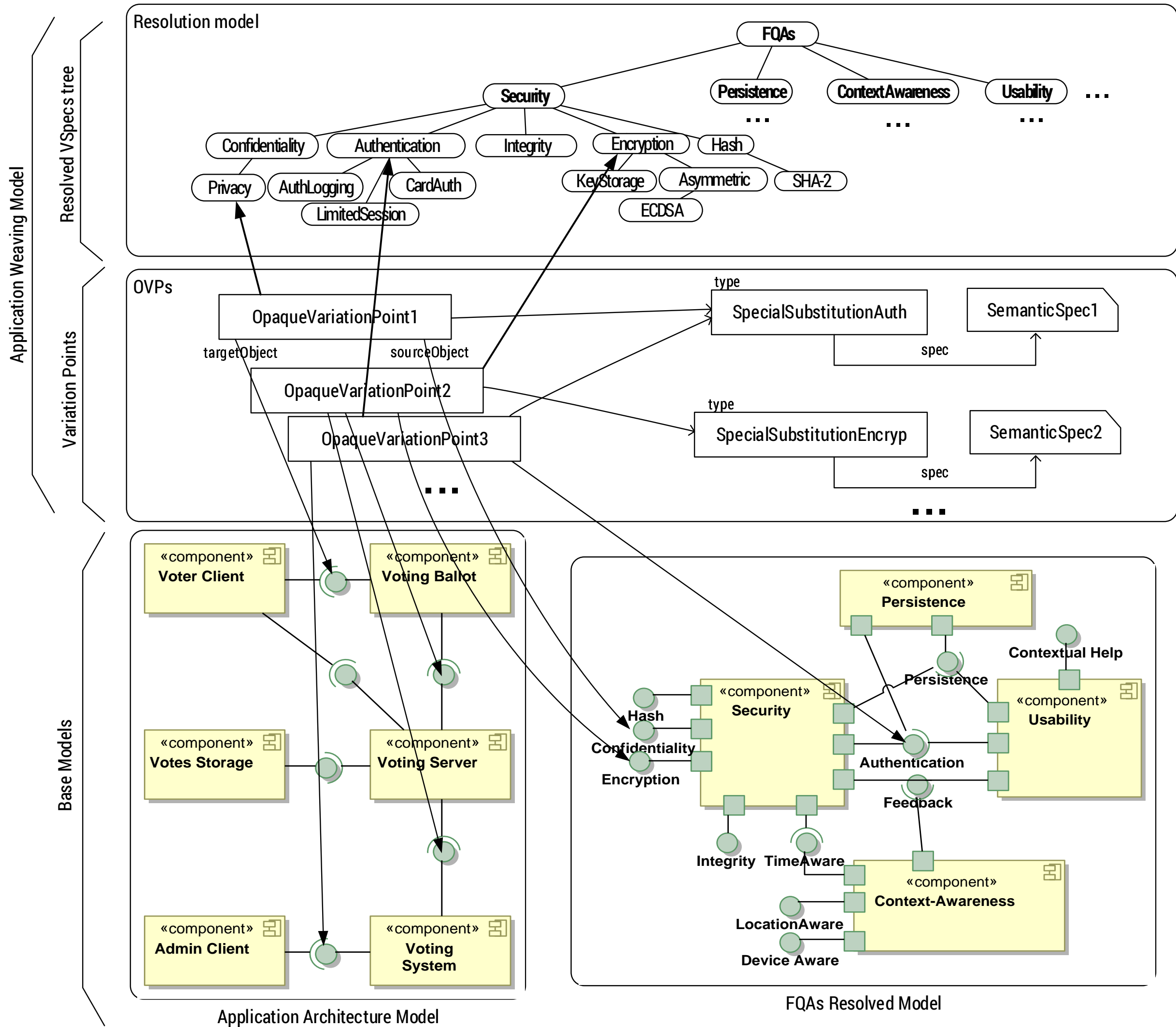


Figure 10: CVL model for the incorporation of the FQAs inside the application architecture.

show three OVPs bound to three concerns: privacy, encryption, and authentication, for reasons of simplicity.

How our transformations implementing the weaving are invoked by the CVL engine is shown in Algorithm 1 of Figure 9. It takes the set of OVPs defined (S_{OVP}) and for each OVP the semantics of the special substitution associated (`vp.type`) is executed by the M2M transformation engine (line 4). $S_{joinpoints}$ is the set of join points referenced by the OVP. The output of this algorithm is an automatically generated model representing the complete application software architecture with the custom FQAs (see Figure 11). The difference between this and the base application architecture is that those component interactions that are affected by FQAs are stereotyped as «`crosscuts`». However, the stereotype is insufficient because is not possible to express how aspectual interactions behave. In order to solve this limitation, similarly to the AOM approach we defined in another paper [23], aspectual interactions are also represented by a set of sequence diagrams. Note that these sequence diagrams are also automatically generated by the transformation rules of the special substitutions. For example, Figure 11(b) shows the sequence diagram that specifies how the authentication concern is woven when the `Voter Client` and `Voting Ballot` components interact. This sequence diagram complements the «`crosscuts`» `C1` of Figure 11(a). We also show a similar example that encrypts and decrypts the information interchanged between the `Voting Ballot` and the `Voting Server` components (Figure 11(c)).

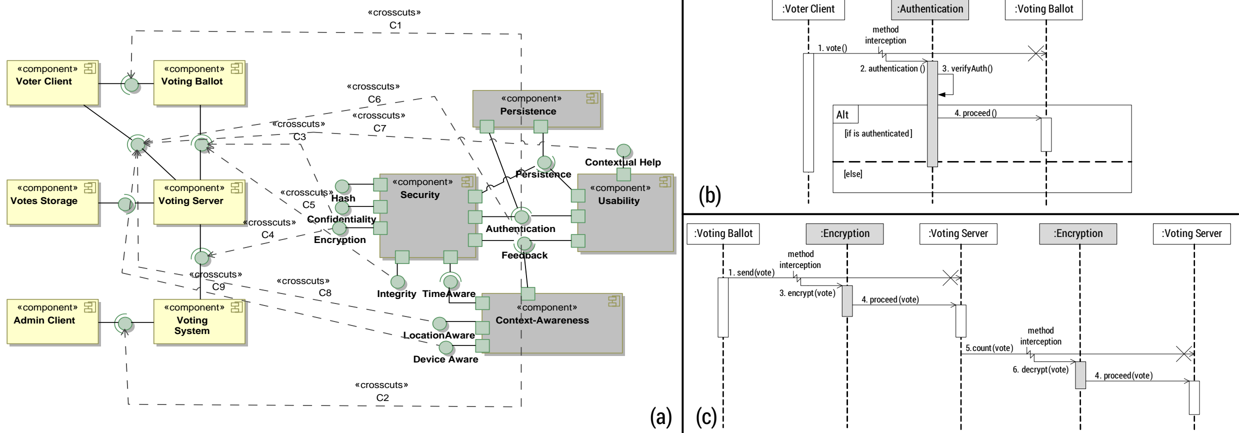


Figure 11: Complete application architecture with the FQAs woven using CVL.

6.2.2. *Implementation of the special substitutions as M2M transformations*

Now we have to define the semantics of the special substitutions for each weaving pattern defined in Table 1 by means of M2M transformations. Basically, each special substitution, implemented as a parameterised M2M transformation, consists of three main functions (Algorithm 2 in Figure 9): (1) to copy selected concerns and all related elements in the resolved model — `copy` (line 2); (2) to create the `«crosscuts»` relationships between the advice and the join points following the weaving patterns of Table 1 — $\texttt{weavingPattern}_{\texttt{type}}$ (line 3); and (3) to represent the interactions between the components defined by the weaving pattern (see Figure 11(b)(c)) — $\texttt{interaction}_{\texttt{type}}$ (line 4). Each special substitution can be represented as a sequence of the operations shown in Algorithm 2 (Figure 9). We have defined these operations using the elementary operations inspired by the MOF reflective API [8]. For example, one MOF operation is `setReference(me, r, References)` that corresponds to the assignment of `References` to the reference `r` of the model element `me`. We use it to add `«crosscut»` references to the application architecture model.

In order to illustrate the implementation of weaving patterns, we show how it works for WP5 (see Figure 12), where multiple advice functions of the same concern are woven into different join points. The selected methods representing the join points can belong to the same or to different interfaces defined by the MOF application metamodel. The FQA that typically uses this weaving pattern is the encryption concern of security that needs to encrypt the user votes before sending

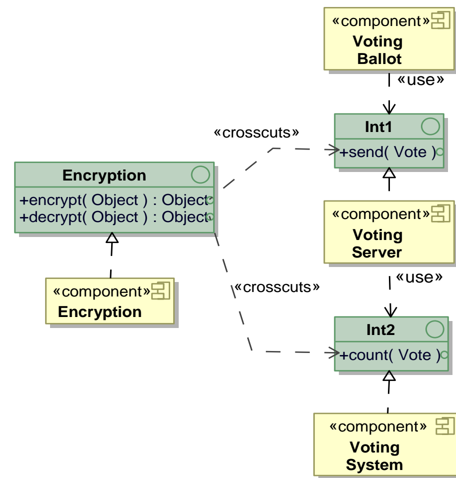


Figure 12: Result of applying Weaving Pattern 5 for Encryption.

(`encrypt(Object)` advice) and decrypt them after receiving (`decrypt(Object)` advice). To model it, the pattern specifies a `«crosscuts»` link between each pair of join point — advice. For example, WP5 specifies a `«crosscuts»` link between the `send(vote)` join point (at `Int1` interface) and the `encrypt()` advice, and another one between the `count(vote)` join point (at `Int2` interface) and the `decrypt(Object)` advice in order to decrypt the votes in the counting process. Jointly with this class diagram we define a sequence diagram representing this interaction (see Figure 11(c)).

Each M2M transformation can be implemented in any transformation language, but we have used ATL. Like the AO-ADL connector templates, these transformations are written once and can be reused in any application by instantiating them with the specific applica-

[8] `http://www.omg.org/spec/MOF/2.0/`

tion's parameters such as the name of the target method and the components involved in the relationships. We consider that the effort required in using the two approaches (i.e., AO-ADL and CVL) is quite similar, since in both cases the software architect only has to specify the join points of the application where the FQAs have to be woven and then he/she just has to execute the supporting tool (AO-ADL tool or CVL and M2M engines). Also, in both cases the connector templates or the M2M transformations can be reused and are easily modifiable by the application engineer. In case of CVL, our approach enables the weaving process to be performed over multiple views (e.g., state diagrams, sequence diagrams, etc.) by using the CVL extension mechanism to integrate different model-to-model transformations as part of the weaving step — i.e., using the Opaque Variation Points (OVPs). The FQA weaving schema with CVL (shown in Figure 9) and our weaving patterns (described in Table 1) have been defined generically for the different views, and only the weaving rules (M2M transformations) need to be defined differently for each view. In this paper, the same weaving process and the same weaving patterns have been applied both to do the weaving with the structural view and to automatically generate the sequence diagram. This means that the M2M transformations that automatically generates the sequence diagram can be reused, slightly modified to perform the aspect weaving in an already existing sequence diagram.

7. Evaluation

In this section we evaluate our work quantitatively by using metrics to quantify the benefits provided by our approach, and qualitatively, where the use of a metric is unsuitable.

7.1. Degree of Modularity and Reusability

The initial hypothesis of this approach was that separating FQAs has many advantages: automatic generation of highly cohesive architectures with low coupled components and reusability increment of both FQAs and application components. In [24] it is demonstrated that coupling and cohesion metrics are the main predictors of reusability. Therefore, in order to provide evidence of this hypothesis we use several coupling and cohesion metrics (see Table 2). We evaluate these metrics only considering the FQAs concerns, and not other functional concerns specific to the application. We need to calculate these metrics on two levels: (1) at the FQA product line architecture level, and, (2) at the application architecture level with the FQAs woven. Also, since these metrics depend on the elements used to model the architecture, the results are different for AO-ADL and MOF-based architectures. In [21] we demonstrated with several metrics that non-AO architectures are more coupled and less cohesive than AO-ADL architectures. So, in this section we focus on the case of MOF-based architectures, showing that coupling and cohesion levels are acceptable.

Coupling is measured by Fan-in and Fan-out and cohesion by LCC metrics (Table 2) [25]. Regarding the FQA product line architecture (1), we can calculate these metrics considering: (i) connections between composite components modelling entire FQAs; and, (ii) also inside each composite component. The Fan-in is calculated by counting the number of components, which require services from the assessed component. The higher the Fan-in for a given component, the more reusable it is. Therefore, it makes no sense to use this metric at the FQA product line architecture level since the reusability degree inside the FQA product line architecture is not significant. The Fan-out counts the number of components required by the assessed component, or in other words, the number of dependency relationships between the component assessed and the rest of the components. Inside FQA composite components, this result is exactly the number of intraFQA-dependencies modelled by the domain experts as provided-required interfaces. Looking at Figure 5 it is easy to see that there are some concerns with a Fan-out greater than 1, which requires many other concerns (e.g., `Encryption Manager` or `Session Manager` at the `Security` composite component) but this normally happens when there are many implementations for the same concern (i.e., modelled as optional features, such as `DSA`, `ECDSA`, ..., and other encryption algorithms). But, note that in the resolved model, if only one of the possible implementations is permitted, then this number plummets to one. Likewise, calculating the Fan-out of the FQA composite components is the same as calculating the number of interFQA-dependencies modelled by the domain experts. With respect to the cohesion LCC metric, it counts the number of concerns addressed by both the functionality and the required interfaces of the assessed component. The higher this number is, the lower the cohesion. In this case, the LCC metric is greater than one for those components which require other components, so it is also related with the dependency number defined. We calculate the dependency degree in Subsection 7.3. How well we have decomposed the FQA in components is measured by the CDAC metric [25], which measures the diffusion of a concern between several components

Table 2: Metric Suite.

Modularity and Reusability Metrics	**Fan-in coupling metric:** It measures the number of components that require one service. **Fan-out coupling metric:** It measures the number of services required by a component. **Lack of concern-based cohesion (LCC):** It measures the number of concerns tangled in a particular component. **Concern diffusion over architectural components (CDAC):** It measures the number of components in which a concern is scattered.
Variability Metrics	**#choices:** It measures the total number of choices in a VSpec. **#resolutions:** It measures the total number of valid resolutions that can be generated from a VSpec. **Variability level:** Expressed as the ratio `#choices:#valid resolutions`.
Dependency Metrics	**#intraFQA-dependencies:** It measures the number of dependencies between the concerns of a FQA. **#interFQA-dependencies:** It measures the number of dependencies between different FQAs.

in the software architecture. In our case, we deliberately encapsulate only one concern in each component, so CDAC is always one for both concern components and FQA composite components.

At the application architecture woven with FQAs level (2), these metrics depend on the number of concerns required by the application and on the number of join points where the FQA components are woven (i.e., the Fan-in of FQA components from the point of view of the application). Table 3 shows the number of concerns required by several industrial case studies. Considering an average of 7.83 concerns and the Fan-in metric of each FQA, it is possible to calculate the reuse rate. Comparing all the case studies, the highest reusability degree is for the encryption and integrity concerns (83%), which are reused several times in five case studies. Regarding the Fan-out and LCC metrics, since dependency relationships between application architecture and FQA configuration are specified outside the component interfaces; injecting FQAs, therefore neither increment the coupling, nor the cohesion of base application. Therefore, we consider our initial hypothesis to be proven.

7.2. Degree of Variability

The first challenge posed by this approach was to manage FQA variability (Challenge 1). In order to evaluate how well we address this challenge we calculate the degree of variability of the FQAs considered in the EV case study. Table 4 shows the number of choices the software architect has to select, initially, in the application engineering phase and the number of components that could be injected per selected FQA. The highest number corresponds to the Security FQA with 23 choices and 25 components. We have tried to model all possible variation points for each FQA, in order to cover a wide range of concerns. This table also shows the number of different "valid" resolutions (configurations) that can be generated using our approach.

What we wish to highlight with this metric is that as the number of choices increases, the number of architecture configurations also increases exponentially. For example, the number of security configurations (or resolutions) is 16589, which means that the software architect could obtain any of these architecture configurations. But note, that the software architect does not need to be aware of this high variability, they only have to focus on choosing those concerns that fit application requirements. The variability level is lower for context awareness (12 choices: 191 resolutions), usability (10 choices: 79 resolutions), and persistence (7 choices: 19 resolutions) than for security. These results show the benefits of defining a variability model of FQAs in our generic process: (i) it is applicable to real case studies (as those in Table 3) since it covers a wide variability spectrum of FQAs and their concerns; (ii) the number of components that can be automatically injected into the application architecture is high (around 55 components as Table 4 shows); (iii) the software architect can reason about possible valid architecture configurations that match application requirements, one by one, by selecting/deselecting different variation points.

7.3. Degree of Dependency

Regarding Challenge 2, we evaluate the degree of dependency calculated for the FQAs considered in the EV case study (Table 5). We can observe the number of intraFQA-dependencies, interFQA-dependencies and the total number of dependencies of each FQA. Despite the fact that modelling the dependency relationships implies more complexity in the definition of the FQAs, this complexity is transparent to the software architect since it is tackled by the domain experts, at the beginning of our process.

Our hypothesis here is that the interFQA-dependencies are not usually considered by the software architect; and, in order to consider the intraFQA-dependencies for each FQA the software architect should be an expert in a wide range of FQAs, which is not always the case. Let us imagine that the software architect does not use our approach. In this case, he/she would have to explicitly identify and model 15 dependencies, where 33% of these dependencies

Table 3: Case studies.

Case study	FQAs	#concerns	Required concerns
E-Voting (EV)[1]	Security	6	privacy, confidentiality, encryption, integrity, hashing, and authentication
	Context awareness	3	location aware, time aware, and device aware
	Usability	2	contextual help and feedback
	Persistence	2	database storage and files storage
Intelligent Transportation (IT)[1]	Security	7	privacy, confidentiality, integrity, encryption, authentication, hashing, and digital signature
	Context awareness	2	location aware and environment aware
	Usability	1	contextual help
File Sharing (FS) [26]	Security	2	authorisation and encryption
	Context awareness	2	location aware and device aware
	Persistence	1	files storage
Health Watcher (HW) [27]	Security	5	integrity, hashing, encryption, authentication, and authorisation
	Usability	1	contextual help
Toll (TS)[2]	Security	3	integrity, hashing, and encryption
	Usability	1	feedback
Crisis Management (CM) [28]	Security	5	integrity, hashing, recovery, authentication, and authorisation
	Usability	1	feedback
	Persistence	1	files storage
	Error Handling	2	error checking and exception handling

[1] `http://www.inter-trust.eu//`
[2] `http://www.infolab21.lancs.ac.uk/docs/aosd.pdf`

Table 4: Degree of variability.

	Security	Context awareness	Usability	Persistence	Total
#choices	23	12	10	7	52
#components	25	15	10	5	55
#resolutions	16589	191	79	16	5628368

Table 5: Degree of dependency.

Dependencies	Security	Context awareness	Usability	Persistence	Total
#intraFQA	7	2	1	0	10
#interFQA	2	1	1	1	5
#total	9	3	2	1	15

are interFQA-dependencies, and 67% are intraFQA-dependencies[9]. This could be an error-prone task, since some of these dependencies are not so evident or can be forgotten, as we have already shown in Section 4.3. Since these dependencies are formally modelled as tree relationships (e.g., cross-tree constraints), we can ensure that if the domain experts have done their job correctly, the resulting architecture contains all the components required to implement all application requirements with regard to FQAs; and furthermore the components are correctly connected.

7.4. Degree of Automation

Challenge 3 is addressed by defining reusable architecture weaving patterns that allow customised FQAs to be injected in an AO modelling way. We have defined seven weaving patterns that are used in several FQAs (Table 1). We have shown that these patterns have been implemented in AO-ADL and for MOF-compliant architectures. But, the contribution here is the automatic process that instantiates the WPs, by connecting FQAs architecture configurations in certain points of the application architecture. We assess how well our approach addresses this challenge by showing the saving in effort that this automatic weaving represents compared with doing the same thing manually, using the degree of automation metric.

The degree of automation is a measure that allows the comparison between the software elements (e.g., number of requirements at the specification level, architectural elements at the architectural level or lines of code at the implementation level) that need to be defined manually and the software elements that are generated automatically using a specific approach. The degree of automation of an approach also lends itself to assessing the development effort that is needed in order to use a particular approach, and the degree of reuse and the gain in productivity that can be achieved.

For the software architect, the main effort is in modelling the main functionality of the application. Thus, in order to specify the degree of automation of our approach, we compare the number of architectural elements that are manually created in the specification of the software architecture (this includes components, interfaces, ports, interfaces realisations, interfaces usages, and other relationships such as dependencies) with the number of architectural elements that are automatically generated by our process, as we show in Equation 1 [29]. We only include the results of the CVL approach, since the results of the AO-ADL approach are

[9] We count parent-child relationships and tree constraints also as intraFQAs-dependencies.

Table 6: Degree of automation.

Case study	Specified elements manually	Specified elements automatically	Degree of automation
EV	26	118	81.94%
IT	25	102	80.31%
FS	24	23	48.94%
HW	62	47	43.12%
TS	126	30	19.23%
CM	39	56	58.95%

very similar as shown in [12].

$$\text{Degree of Automation} = \frac{\#elements_FQAs}{\#elements_core + \#elements_FQAs} \quad (1)$$

We observe that, in the case study of this paper, the core application is composed of 26 architectural elements as shown in the UML `Application Architecture Model` in Figure 10: 6 external components + 6 interfaces realisations + 6 interfaces usages + 8 additional internal elements (e.g., subcomponents,...) that we have omitted in the figure for simplicity. When the FQAs are incorporated, 118 new elements are added to the architecture (see Figure 6): 32 components and subcomponents + 18 ports + 26 interfaces realisations + 33 interfaces usages + 9 additional «crosscuts» relationships as shown in Figure 11; obtaining a degree of automation of 81.94%. This value is higher than in the other case studies because the EV application requires many FQAs concerns, with many dependencies between them. The higher the number of FQAs required, and the higher the number of dependencies between them, the greater the benefit obtained with our process. For instance, the IT case study also has a high degree of automation (80.31%). The degree of automation in the other case studies is lower because they require fewer FQAs.

7.5. Tool support

Both instantiations of the generic process are supported by tools. For the first instantiation, FM specification is supported by our own feature modelling tool (Hydra), but any other FM tool could be used instead. The rest of the process is supported and totally implemented as part of the AO-ADL Tool Suite. We have already shown the correct functioning of this tool in [21]. The CVL instantiation is covered by the industrial CVL tool[10]. In this case, we also implemented the seven weaving patterns as M2M transformations. Since, we are not responsible for the correct functioning of the CVL tool, we limit ourselves to discussing the correctness of the M2M transformations. The M2M transformations are not so complex since they consist basically in instantiating the parameters defined by the pattern. Concretely, the M2M transformations bind the join points with the FQA components that define the advice to be applied in these join points. This means that our M2M transformations do not modify any components and connections of the MOF-compliant architecture. These transformations only add `«crosscuts»` relationships where the software architect has said to add them. We can ensure that the generated architecture works well, since is very easy to visually compare the previous and generated architectures by simply taking a look at the `«crosscuts»` relationships defined at the desired join points. Regarding the sequence charts, they are automatically generated once the software architect has specified the OVPs. These message sequence charts are pre-defined in the M2M transformations and simply have to be instantiated with the required parameters: source and target components, selected concern and advice, and the advice type (i.e., before, after or around) if needed.

[10]We are developing a CVL tool to completely support our approach. A prototype is available in `http://150.214.108.91/code/cvl` and `http://150.214.108.91/code/cvltool`.

8. Discussion

The evaluation results obtained here show that the effort of separately defining FQAs forming an SPL family has many advantages: (i) helps the software architect to identify the interFQA- and intraFQA-dependencies; (ii) helps the software architect to identify the variability degree of the solution domain; (iii) the resulting application resolved model is less coupled and very cohesive. But, these benefits are almost the same regardless of the SPL approach used, so in this section we discuss the specific advantages of using either AO-ADL or CVL in our approach.

There are many differences between the two approaches: (1) the first difference is the variability language, FM or VSpec tree. The advantage of VSpec trees is that they can be modularised, while FMs require all features in a single tree to be included; (2) regarding the modelling of software architectures, the CVL approach is more generic than the AO-ADL approach. The reason is that CVL allows the use of any MOF-compliant model, and not an ADL defined by the academia as is the AO-ADL language; (3) with regard to the modelling of the variability of the FQAs, CVL separates the specification of the software architecture from the specification of its variation points. However,

using AO-ADL, the variability information and the architecture information is tangled in the VML file. The VML file stores the links between variation points and architecture components textually, being difficult to see the specified connections or check the correctness of them; (4) with respect to the weaving stage, in AO-ADL the software architecture of the base application and the software architecture of the FQAs was not automatically woven. Only the pointcuts for selecting the points of the base architecture where the FQAs need to be woven are defined. The advantage of this is that FQAs and base architecture remain separate, but the disadvantage is that it is difficult for the software architect to reason about the final architecture. In the CVL case, the CVL execution engine automatically performs the weaving and we are able to automatically generate the complete software architecture of the application, including the FQAs; (5) with CVL our approach enables the weaving process to be performed over multiple views (e.g., state diagrams, sequence diagrams, etc.) by using the CVL extension mechanism to integrate different model-to-model transformations as part of the weaving step (i.e., the OVPs); finally (6) the main advantage of CVL compared with AO-ADL is that CVL is a proposed standard, being widely known and accepted by both industry and academia.

Although CVL seems to be the most adequate option of our generic process, it also has some counterparts. On the one hand, with MOF a software architect can define custom meta-models, but our CVL approach assumes that the MOF meta-models used to describe the application architecture and the FQA architecture are compatible. Compatible here means that the application's architectural meta-model expresses, at least, the same constructs as the meta-model of the initial FQAs model. But, this is not always possible, as a potential user could use a non-compatible Domain-Specific Language. This shortcoming could be easily solved through an additional model transformation step before performing the weaving process. On the other hand, when several FQAs are applied at the same join point of the application, the order of the advice is the same as how the software architect defined the OVPs in the Application Weaving Model and cannot be explicitly specified. However, in AO-ADL the software architect can specify the order of the advice functions in the AO-ADL connector template — e.g., binding order (line 6) in Figure 7. Finally, the CVL tool[11] is not mature enough and does not completely support the CVL specification. Currently, it can be considered an out-of-date prototype implementation of the CVL specification that is no longer under active development.

[11] CVL Tool from SINTEF: `http://www.omgwiki.org/variability/doku.php?id=cvl_tool_from_sintef`

9. Related Work

In this section, we provide an overview of some of the work in the field of SPLs, variability of quality attributes, and weaving models.

Most of the approaches that model QAs variability focus on the analysis of the QAs as non-functional requirements (e.g., cost, maintenance, performance, availability) in the final product of an SPL, and how the variations in the functional components of the application affect those QAs. For example, the approaches presented in [5, 6, 30, 31, 7, 32] model variations of QAs by extending feature models in different ways: in [5], Benavides et al. deal with extra functional features using attributes, characteristics of a feature that can be measured (e.g., latency) and relationships between attributes. In [6] González-Baixauli et al. focus on the variant analysis of non-functional requirements by introducing a goal/softgoal paradigm and relating it with feature modelling and use case modelling. Jarzabek et al. [30] propose an integrated modelling framework (F-SIG, Feature-Softgoal Interdependency Graph) that extends feature modelling with concepts of goal-oriented analysis in two ways: (1) records design rationale in the form of inter-dependencies among variant features and QAs during the design of an SPL architecture, and (2) evaluates the impact of variant features selected for a target system during its construction. Zhang et al. [31] use feature models to capture functional requirements of an application while using Bayesian belief models to capture the impact of functional variants on the QAs. Sinnema et al. [7] propose COVAMOF, which is a framework to model variability on all layers of abstraction of an SPL. COVAMOF captures the variability of FQAs in terms of variation points and dependencies by using associations. Dependencies specify properties that define values of the QAs such as performance or memory usage. Finally, in [32], George et al. analyse the impact of security properties on other functional concerns of the base application using an AO approach. In contrast to our proposal, none of these approaches address the variation of the functional part of the QAs themselves. Moreover, QA variability is modelled jointly with the variability of the application.

Another technique to model variability in SPLs instead of feature models is annotating the base model by means of extensions to the base modelling language. In [33, 34], Tawhid and Petriu propose a technique

to model the commonality and variability in structural and behavioral SPL views using Model-Driven Development (MDD). They add generic annotations related to a QA (e.g., performance) to 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) of the QA is derived, and a model for the given product is generated. Annotating the base model makes this closely related with variability specifications and prevents the reuse of both the base model of the application and the variability model of the QAs. In contrast, using a separate variability language such as CVL allows the independence of the variability language and the modelling language to be maintained. In addition, this proposal also models non-functional QAs such as performance instead of FQAs, and introduces the variability at the design level (e.g., within sequences diagrams) while we model the variability of the FQAs earlier on in the development process, at the architectural level.

Existing work that addresses FQAs variability considers that they are part of an SPL. For instance, QADA [8] (Quality-driven Architecture Design and quality Analysis) is a specific method to design SPL architectures by transforming systematic functionality and QAs into software architectures, but this proposal does not take into account the quality requirements explicitly. The RiPLE-DE [9] (RiSE Product Line Engineering - Design Engineering) process is a domain design process for SPL that can be extended to model the FQAs variability as part of a family of products. The QAs variability is represented in feature model diagrams and in order to achieve desired quality levels, the QAs are complemented with information about the base application (e.g., the system's response measure). The variation of the attributes is given by that information which is usually represented in numerical values and the architecture is evaluated in order to achieve the necessary variation of the QAs. Thus, the variability of the FQAs directly depends on the base application, avoiding the reuse of the FQAs.

Another approach that deals with similar challenges as our approach is the concern-oriented reuse (CORE) process [35]. The main differences between the CORE approach and our approach are that (1) they model the variability of the interfaces of the concerns (e.g., interfaces of frameworks or components) instead of modelling the variability of the internal functionality of the components as we do; (2) they also focus on the impacts of the concerns on non-functional qualities (e.g., access time, efficiency, etc.) by specifying goals using goal models; while we focus on the functional part of the quality attributes (e.g., the implementation of a particular encryption algorithm and its variants). (3) Their approach depends on the Reusable Aspect Models (RAM) weaver. RAM is an AO multi-view modelling approach [36] for software design modelling that consists of a UML package specifying the structure and the behaviour of a software design using class, sequence, and state diagrams. So, the RAM weaver is specific for UML models and makes difficult to apply the approach to others ADLs. Our approach, instead, is independent of the language to model the architectures and in the case of CVL, our approach is suitable for using with any MOF-compliant language. Additionally, CVL provides the advantages of MDD by allowing us to define custom model transformations to apply any kind of modification to the architecture.

Recently, CVL has been applied in multiple approaches. For instance, CVL is used to manage the variability in the context of software processes [37], business processes [38], or even for synthesising an SPL using model comparison [39]. In [40] the CVL approach is adopted to specify and resolve the variability of software design, such as in workflows. The authors compose the detailed structural and behavioural design models of the chosen variants by using, as in [35], the RAM weaver. However, contrary to our proposal, this external weaver is responsible for composing the reusable aspects instead of implementing the weaving process by using CVL and the transformation engine as we do. Additionally, they apply the CVL approach at the design level while we focus at the architectural level (e.g., component diagrams).

Model Driven Engineering (MDE) has also been used in the field of SPLs [41]. Sijtema proposes a strategy to let ATL handle the variability by extending the concrete syntax of ATL with the concept of variability rules. Variability rules are used in the context of a transformation sequence which successively refines models. However, they first model the variability separately in a feature diagram and they have to make the feature selections and the realisation of the artifacts correspond. In comparison with our proposal, using CVL we model the variability and bind the features directly to the elements in the software architecture. We use the basic ATL without the need to extend it, but our proposal can also be used with other transformation languages such as QVT or ETL.

10. Conclusions

We have proposed a generic process for modelling the variability of the FQAs independently of the application affected by them. Separating the modelling of the FQAs and the software architecture of the application we improve the separation of concerns and the modularisation of the FQAs from the early stages of the development process. We manage the dependencies and interactions between the concerns of the FQAs. The subsequent incorporation of the FQAs into the software architecture of the application is performed automatically. For that, we have proposed a set of weaving patterns covering all the possible types of weaving and we have mapped the FQA concerns to those weaving patterns. We have demonstrated the feasibility and advantages of using our approach by two instantiation of this generic process with different technologies, academic ones (i.e., FM, AO-ADL and VML) and industrial ones (i.e., CVL and MOF-compliant architectures). We have also validated the different challenges posed by our approach by using different metrics with very good results.

Acknowledgements

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

References

[1] M. Barbacci, M. Klein, T. Longstaff, C. Weinstock, Quality Attributes, Technical Report CMU/SEI-95-TR-021 ESC-TR-95-021, Carnegie Mellon University, Software Engineering Institute (1995).

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

[3] K. Pohl, G. Böckle, F. J. v. d. Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., 2005.

[4] L. Etxeberria, G. Sagardui, L. Belategi, Quality aware software product line engineering, Journal of the Brazilian Computer Society 14 (1) (2008) 57–69. doi:10.1007/BF03192552.

[5] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, in: Advanced Information Systems Engineering, Vol. 3520 of LNCS, Springer Berlin Heidelberg, 2005, pp. 491–503. doi:10.1007/11431855_34.

[6] B. González-Baixauli, J. Prado Leite, J. Mylopoulos, Visual variability analysis for goal models, in: 12th IEEE International Requirements Engineering Conference, 2004, pp. 198–207. doi:10.1109/ICRE.2004.1335677.

[7] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, Modeling dependencies in product families with COVAMOF, in: 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS, 2006, pp. 9 pp.–307. doi:10.1109/ECBS.2006.49.

[8] M. Matinlassi, E. Niemelä, 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.

[9] R. d. O. Cavalcanti, E. S. de Almeida, S. R. Meira, Extending the RiPLE-DE process with quality attribute variability realization, in: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS, QoSA-ISARCS, ACM, 2011, pp. 159–164. doi:10.1145/2000259.2000286.

[10] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wąsowski, Cool features and tough decisions: A comparison of variability modeling approaches, in: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS, ACM, 2012, pp. 173–182. doi:10.1145/2110147.2110167.

[11] Ø. Haugen, A. Wasowski, K. Czarnecki, CVL: Common Variability Language, in: 16th International Software Product Line Conference, Vol. 2 of SPLC, 2012, pp. 266–267. doi:10.1145/2364412.2364462.

[12] R. Lence, L. Fuentes, M. Pinto, Quality attributes and variability in AO-ADL software architectures, in: Proceedings of the 5th European Conference on Software Architecture: Companion Volume, ECSA, ACM, 2011, pp. 7:1–7:10. doi:10.1145/2031759.2031768.

[13] J. M. Horcas, M. Pinto, L. Fuentes, Variability and dependency modeling of quality attributes, in: 39th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA, 2013, pp. 185–188. doi:10.1109/SEAA.2013.20.

[14] J. M. Horcas, M. Pinto, L. Fuentes, Injecting quality attributes into software architectures with the common variability language, in: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE, ACM, 2014, pp. 35–44. doi:10.1145/2602458.2602460.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University (1990).

[16] M. Fontoura, W. Pree, B. Rumpe, The UML Profile for Framework Architectures, Addison-Wesley Longman Publishing Co., Inc., 2000.

[17] H. Gomaa, Designing software product lines with UML 2.0: From use cases to pattern-based software architectures, in: Reuse of Off-the-Shelf Components, Vol. 4039 of LNCS, Springer Berlin Heidelberg, 2006, pp. 440–440. doi:10.1007/11763864_45.

[18] N. Loughran, P. Sánchez, A. Garcia, L. Fuentes, Language support for managing variability in architectural models, in: Software Composition, Vol. 4954 of LNCS, Springer Berlin Heidelberg, 2008, pp. 36–51. doi:10.1007/978-3-540-78789-1_3.

[19] IEEE standard for a software quality metrics methodology, IEEE Std 1061-1998.

[20] B. Boehm, Characteristics of software quality, TRW series of software technology, North-Holland Pub. Co., 1978.

[21] M. Pinto, L. Fuentes, J. M. Troya, Specifying aspect-oriented architectures in AO-ADL , Information and Software Technology 53 (11) (2011) 1165–1182. doi:http://dx.doi.org/10.1016/j.infsof.2011.04.003.

[22] M. Pinto, L. Fuentes, Modeling quality attributes with aspect-oriented architectural templates, J. UCS 17 (5) (2011) 639–669.

[23] M. Pinto, L. Fuentes, L. Fernández, J. Valenzuela, Using AOSD and MDD to enhance the architectural design phase, in: On the Move to Meaningful Internet Systems: OTM 2009 Workshops, Vol. 5872 of LNCS, Springer Berlin Heidelberg, 2009, pp. 360–

369. doi:10.1007/978-3-642-05290-3_48.

[24] G. Gui, P. D. Scott, Measuring software component reusability by coupling and cohesion metrics, Journal of Computers 4 (9) (2009) 797–805. doi:10.4304/jcp.4.9.797-805.

[25] C. Sant'Anna, E. Figueiredo, A. Garcia, C. Lucena, On the modularity of software architectures: A concern-driven measurement framework, in: Software Architecture, Vol. 4758 of LNCS, Springer Berlin Heidelberg, 2007, pp. 207–224. doi:10.1007/978-3-540-75132-8_17.

[26] G. G. Pascual, M. Pinto, L. Fuentes, Component and aspect-based service product line for pervasive systems, in: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE, ACM, 2012, pp. 115–124. doi:10.1145/2304736.2304757.

[27] S. Soares, E. Laureano, P. Borba, Implementing distribution and persistence aspects with AspectJ, SIGPLAN Not. 37 (11) (2002) 174–190. doi:10.1145/583854.582437.

[28] J. Kienzle, N. Guelfi, S. Mustafiz, Crisis management systems: A case study for aspect-oriented modeling, in: Transactions on Aspect-Oriented Software Development VII, Vol. 6210 of LNCS, Springer Berlin Heidelberg, 2010, pp. 1–22. doi:10.1007/978-3-642-16086-8_1.

[29] A. Harrington, V. Cahill, Model-driven engineering of planning and optimisation algorithms for pervasive computing environments, in: IEEE International Conference on Pervasive Computing and Communications, PerCom, 2011, pp. 172–180. doi:10.1109/PERCOM.2011.5767582.

[30] S. Jarzabek, B. Yang, S. Yoeun, Addressing quality attributes in domain analysis for product lines, Software, IEE Proceedings - 153 (2) (2006) 61–73.

[31] H. Zhang, S. Jarzabek, B. Yang, Quality prediction and assessment for product lines, in: Advanced Information Systems Engineering, Vol. 2681 of LNCS, Springer Berlin Heidelberg, 2003, pp. 681–695. doi:10.1007/3-540-45017-3_45.

[32] G. Georg, R. France, I. Ray, An aspect-based approach to modeling security concerns, in: Proceedings of the Workshop on Critical Systems Development with UML, 2002, pp. 107–120.

[33] R. Tawhid, D. Petriu, Integrating performance analysis in the model driven development of software product lines, in: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS, Springer-Verlag, 2008, pp. 490–504. doi:10.1007/978-3-540-87875-9_35.

[34] R. Tawhid, D. Petriu, Automatic derivation of a product performance model from a software product line model, in: 15th International Software Product Line Conference, SPLC, 2011, pp. 80–89. doi:10.1109/SPLC.2011.27.

[35] O. Alam, J. Kienzle, G. Mussbacher, Concern-oriented software design, in: Model-Driven Engineering Languages and Systems, Vol. 8107 of LNCS, Springer Berlin Heidelberg, 2013, pp. 604–621. doi:10.1007/978-3-642-41533-3_37.

[36] J. Kienzle, W. Al Abed, J. Klein, Aspect-oriented multi-view modeling, in: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD, ACM, 2009, pp. 87–98. doi:10.1145/1509239.1509252.

[37] E. Rouille, B. Combemale, O. Barais, D. Touzet, J.-M. Jezequel, Leveraging CVL to manage variability in software process lines, in: 19th Asia-Pacific Software Engineering Conference, Vol. 1 of APSEC, 2012, pp. 148–157. doi:10.1109/APSEC.2012.82.

[38] C. Ayora, V. Torres, V. Pelechano, G. H. Alférez, Applying CVL to business process variability management, in: Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone, VARY, ACM, 2012, pp. 26–31. doi:10.1145/2425415.2425421.

[39] X. Zhang, O. Haugen, B. Moller-Pedersen, Model comparison to synthesize a model-driven software product line, in: 15th International Software Product Line Conference, SPLC, 2011, pp. 90–99. doi:10.1109/SPLC.2011.24.

[40] B. Combemale, O. Barais, O. Alam, J. Kienzle, Using CVL to operationalize product line development with reusable aspect models, in: Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone, VARY, ACM, 2012, pp. 9–14. doi:10.1145/2425415.2425418.

[41] M. Sijtema, Introducing variability rules in ATL for managing variability in MDE-based product lines, Proc. of MtATL 10 (2010) 39–49.

View publication stats