Information and Software Technology 95 (2018) 147–164

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Variability models for generating efficient configurations of functional quality attributes

Jose-Miguel Horcas*, Mónica Pinto, Lidia Fuentes

CAOSD Group, Universidad de Málaga, Andalucía Tech, Spain

ARTICLE INFO

Keywords:
Energy efficiency
Energy consumption
Quality attributes
Performance
Sustainability
Software product line
Variability

ABSTRACT

Context: Quality attributes play a critical role in the architecture elicitation phase. Software Sustainability and energy efficiency is becoming a critical quality attribute that can be used as a selection criteria to choose from among different design or implementation alternatives. Energy efficiency usually competes with other non-functional requirements, like for instance, performance.

Objective: This paper presents a process that helps developers to automatically generate optimum configurations of functional quality attributes in terms of energy efficiency and performance. Functional quality attributes refer to the behavioral properties that need to be incorporated inside a software architecture to fulfill a particular quality attribute (e.g., encryption and authentication for the security quality attribute, logging for the usability quality attribute).

Method: Quality attributes are characterized to identify their design and implementation variants and how the different configurations influence both energy efficiency and performance. A usage model for each characterized quality attribute is defined. The variability of quality attributes, as well as the energy efficiency and performance experiment results, are represented as a constraint satisfaction problem with the goal of formally reasoning about it. Then, a configuration of the selected functional quality attributes is automatically generated, which is optimum with respect to a selected objective function.

Results: Software developers can improve the energy efficiency and/or performance of their applications by using our approach to perform a richer analysis of the energy consumption and performance of different alternatives for functional quality attributes. We show quantitative values of the benefits of using our approach and discuss the threats to validity.

Conclusions: The process presented in this paper will help software developers to build more energy efficient software, whilst also being aware of how their decisions affect other quality attributes, such as performance.

1. Introduction

Non-functional requirements specify quality attributes (QAs) that must be realized by the system [1]. Over the last decade, energy efficiency has come to be considered as a new quality attribute that evaluates the resource consumption degree of software systems [2], being especially relevant in modern Internet of Things (IoT) applications. Energy efficiency, as well as other non-functional requirements, must be taken into account in the early stages of the software development process, because it affects the system's quality. Indeed, QAs play a critical role in the architecture elicitation phase, serving as selection criteria to choose from among a great number of alternative designs and ultimate implementations. As energy efficiency is such a critical issue, we need to make the correct decisions when choosing the configurations that consume fewer resources.

However, energy efficiency usually competes and conflicts with other non-functional requirements. For example, using the greenest implementation of a functional component reduces the energy consumption, but can penalize system performance. Therefore, if both energy efficiency and performance QAs are required, we need to find a compromised solution. Indeed, the quality degree we wish to accomplish in terms of energy efficiency and performance will, in turn, strongly influence the rationale of architectural design decisions; including the achievement of other QAs that describe behavioural properties (e.g., security, usability) [3]. For instance, to satisfy the security QA, applications may include an encryption component that modifies the system behavior to provide confidentiality. Those components that are added to the system to satisfy some QAs, may affect the performance and energy efficiency of the system.

The specific functionality that is introduced into the application to

* Corresponding author.
E-mail addresses: horcas@lcc.uma.es (J.-M. Horcas), pinto@lcc.uma.es (M. Pinto), lff@lcc.uma.es (L. Fuentes).

https://doi.org/10.1016/j.infsof.2017.10.018
Received 28 April 2017; Received in revised form 26 October 2017; Accepted 27 October 2017
Available online 31 October 2017

fulfill the desired QAs was recently coined as Functional Quality Attribute (FQA) [4,5] as it describes the functional behavior needed to satisfy specific QAs. This kind of functionality has some particular properties, which can be differentiated from the base functionality of the application. For instance, each application may require a different variant or configuration of the FQA (e.g., for confidentiality, a particular encryption algorithm and key length could be required). It is very important to investigate the relationships between FQAs and energy efficiency jointly with performance since FQAs should be designed, implemented and configured independently of the base application components [4,5]. The energy consumption and performance should therefore be analyzed independently of the application functionality.

Moreover, there are several frameworks and third party libraries that provide different implementations of FQAs ready to be reused, such as the Java Security package, the Apache Commons library, and the Spring Framework. Unfortunately, there are not enough experimental studies that show how each FQA solution addresses the energy efficiency concern and if choosing a greener solution will affect the performance of the FQA. A typical scenario is a developer who, although is able to choose between different options providing similar quality, simply decides to use the most popular framework with the default configuration. In general, developers are not aware of the implications of architectural decisions on the energy consumption [2]. They need some help to make the correct design decisions from the point of view of energy efficiency, without penalizing the system performance.

However, finding the optimum configuration of the FQAs that satisfies all the application's requirements and that additionally takes into account both energy efficiency and performance is a difficult and error-prone task to carry out manually. First, there is no catalog to tell the developer what parameters and variables present in the FQAs being considered can affect the energy efficiency and the performance. For example, the size of a message to be encrypted affects both energy efficiency and performance, but so does the algorithm used in the case of the encryption FQA. But, developers do not have enough information to choose the greenest encryption algorithm for a concrete message size, without penalizing performance so much. Moreover, the same FQA can be incorporated into different places of the application architecture. For instance, the caching component can be added in those places of the architecture where the access to the data supposes a bottleneck for the application's performance. Incorporating the same FQA in different places of the application means that the FQA may need to be configured differently at each point based on the interactions between components.

In this article, we define a process that helps application architects and developers to automatically generate optimum configurations of FQAs in terms of energy efficiency and performance for a given application. This work centers the study of energy efficiency and performance on FQAs, because FQAs can be modeled separately from applications and so they can be reused several times in different applications. The main contributions of this paper are: (1) an original process for generating optimum FQAs configurations based on experimental results on energy efficiency and performance; (2) a characterization of the FQAs in operations (e.g., encrypt/decrypt), contextual variables (e.g., the size of the object to be encrypted), configurable parameters (e.g., different algorithms), and implementations (e.g., existing framework and libraries), and the corresponding usage model for each FQA that includes all this information so it can be instantiated in the applications; (3) we consider that an FQA can be injected in different points of the application and also consider the dependencies between the usage model of different FQAs. With our process it is also possible to generate different configurations for the same implementation framework for different points of a given application, so that the overall energy consumption and/or performance are optimized; (4) a variability model schema that models the characterization of (2) and use clonable features to address (3) and that can be reused for any FQA; (5) a CSP formalization that does not model energy as an attribute of each feature as standard SPL processes do. The original work here is that we extend the CSP formulation with: (i) a set of variables that model energy consumption and performance as continuous variables that vary according to the usage model; and, (ii) the dependencies between usage models. And (6) finally, with our process, the application developer can select a partial configuration, being able to reason about the energy efficiency and performance of the different configurations. Indeed, modeling explicitly the **usage model** to conduct experiments of software energy consumption is original of this work.

This paper has been structured as follows. Section 2 outlines our approach. Section 3 characterizes the QAs, identifying the functionalities and variables of the usage models that may affect the energy efficiency and performance. Section 4 models the variability of the FQA usage models. Section 5 shows the experiment results obtained that demonstrate how the different parameters affect the energy efficiency and performance. Section 6 formalizes the variability models and the experiment information so as to automatically generate the configurations. Section 7 shows how our approach generates the best configurations based on the energy efficiency and performance. Section 8 evaluates our approach. Section 9 discusses related work, and Section 10 concludes the paper.

2. Our approach

In this section we present a general overview of our approach (Fig. 1). As discussed in the introduction, there are many variants that need to be considered for each FQA (e.g., different parameters, values of these parameters, operations, frameworks) and each variant will affect energy efficiency and performance in different ways. In order to cope with this variability, our approach is based on the definition of a family of FQAs and, for this reason, we follow the classical Software Product Line (SPL) engineering approach [6], which distinguishes between domain and application engineering processes. Within an SPL two different roles are defined: (1) *domain experts*, who are in charge of defining the assets of the domain engineering process, and (2) *application engineers*, who use those assets, reusing and instantiating them for each application in the application engineering process. Note that domain experts do their work only once. Application engineers reuse the work done by the domain experts when they use the SPL to generate concrete configurations according to application requirements.

FQAs Domain Engineering Process (top of Fig. 1). The primary aim of our domain engineering process is to specify the variability model of a set of FQAs, enriching it with energy and performance information. This information is then used during the application engineering process to generate optimum configurations of the FQAs. Our approach focuses on modeling reusable FQAs, but it does not focus on modeling the base functionality of the applications. The additional step required to integrate the generated FQAs configurations with the software architecture of the base applications is described in [4].

The process is described around some research questions. Firstly, we are interested in investigating the relationships between QAs, FQAs, energy efficiency and performance. So, the first questions that arise here are, given a QA, *which are the behavioral properties of that QA that imply the addition of specific components to its software architecture? (i.e., find out the FQAs)*, and *which are the variables and parameters that affect the energy efficiency and performance of FQAs?* The answers to these questions are given in our approach during the characterization of the QAs (`QAs Characterization` in Fig. 1). Taking as input the list of QAs, domain experts characterize the QAs by identifying the relationships between QAs and the functionalities that are usually required to satisfy the QAs. For instance, to provide security for the applications, some typical functionalities that are incorporated into the applications are encryption, authentication, or hashing, among others. Each of these functionalities has its own operations, context, configurable parameters, and different implementations, that can affect the energy efficiency and performance of the applications. A usage model for each

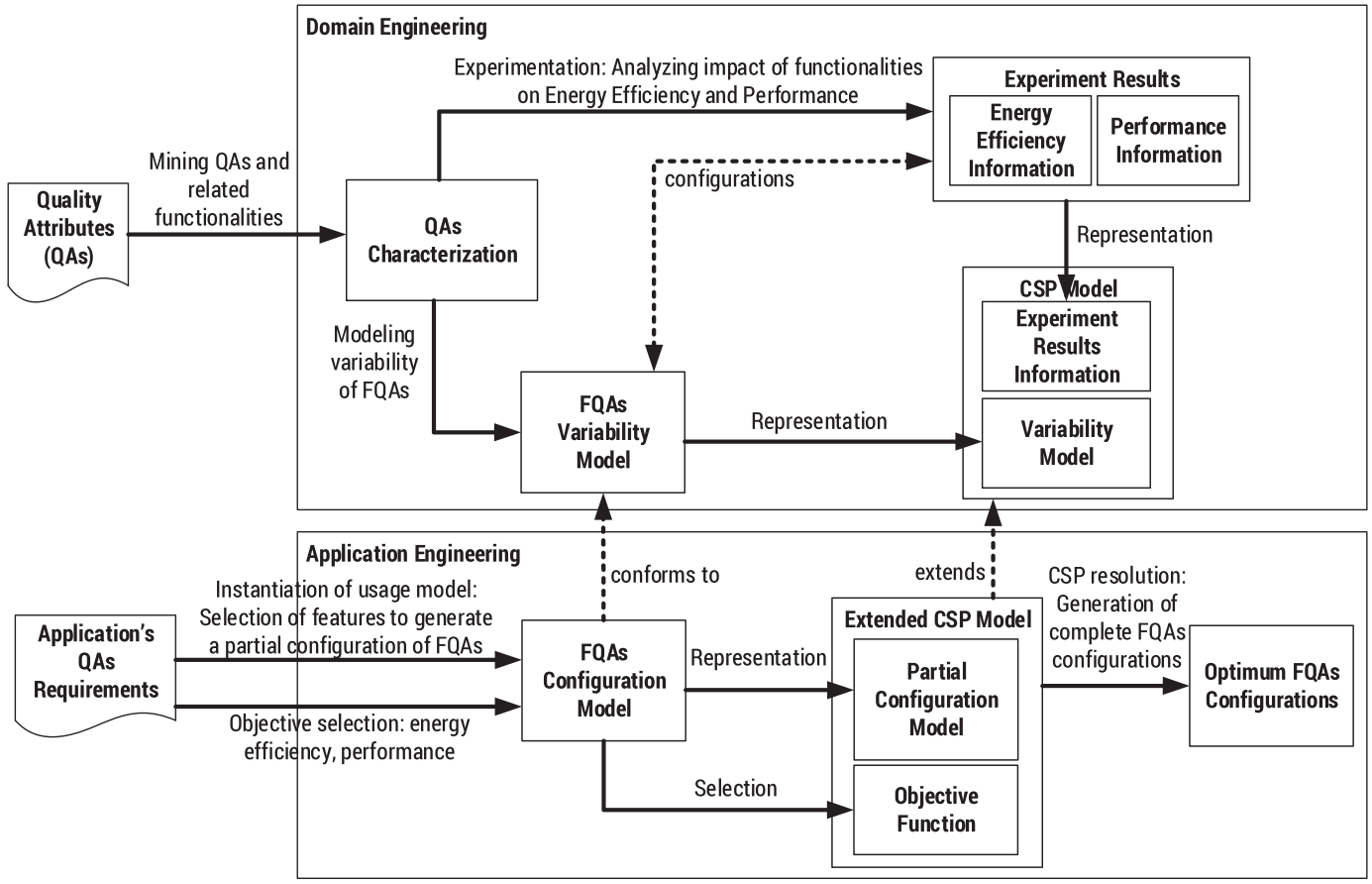

Fig. 1. Software Product Line process for generating FQAs configurations based on energy efficiency and performance.

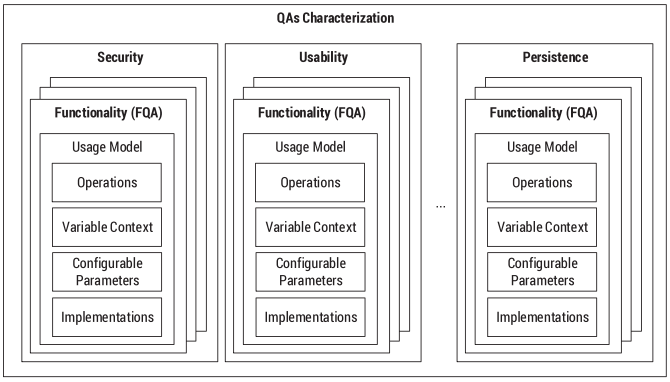

Fig. 2. QAs and the relationships with functionalities (FQAs).

FQA is defined taking into account all this information.

Now that the relevant information has been identified, the next question is *how should that information and its variability be represented in order to manage and reason about it?* In our approach, the information identified during the QAs characterization is represented through a variability model of the FQAs (`FQAs Variability Model`). The variability model defines all the features representing different possible configurations of the FQAs and their usage models, as well as the goals (objective functions) to be considered (e.g., maximizing the energy efficiency, the performance, or achieving a trade-off between both of them). This variability model is then translated to a Constraint Satisfaction Problem (CSP) in order to reason about it (`CSP Model - Variability Model` box). CSP models the features and relationships of the variability model as variables and logical constraints, so as to generate optimum configurations.

Since our final goal is to be able to generate those configurations of our variability model that are optimal with respect to energy efficiency and performance, the next question here is *how do the different variables and parameters of the usage models influence the energy efficiency and performance attributes?* The effects of each of the usage model variables on the energy efficiency and performance are analyzed in our approach through experimentation and simulation (`Experiment Results`) [7]. The experimentation corpus is determined by the valid configurations that can be generated from the FQAs variability model (discontinuos line between `FQAs Variability Model` and `Experiment Results`). Note that in order to choose between different configurations for different usage models, we only need to identify the energy consumption or performance tendencies and not the exact values.

But, *how is the information gathered through experimentation related with the FQAs variability model?* As we have previously represented our variability model as a constraint satisfaction problem, the experiment information is integrated at this level. This means that we extend the CSP model of the variability model with information about the energy consumption and the performance information of the different variants represented by the variability model. Integrating the experimental information at this level instead of directly in the variability model hides the complexity of extending the variability model's metamodel with complex structures to represent the experiment information. Also, we do not follow the classical approach of modeling QAs as attributes of features (e.g., Apache web server: *energy* = 600 Joules [8]), instead, we consider that a QA such as energy can vary depending on the usage model (e.g., number of concurrent users [8]).[1]

[1] All the information generated during the domain engineering process is available in http://150.214.108.91/code/green-fqas

Table 1
Characterization and usage models of some FQAs.

QA	FQA	Usage model	Description
Security	Encryption	Operations	Encrypt.
			Decrypt.
		Variable Context	Size of the message to be encrypted/decrypted.
		Configurable Parameters	Algorithm: AES, DES, RSA, Blowfish,...
			Mode: CBC, CFB, CTR, CTS, ECB,...
			Padding: PKCS1, PKCS5, OAEP,...
			Key size.
		Implementations	javax.crypto package.
			Bouncy Castle.
			...
	Hashing	Operations	Hash.
		Variable Context	Size of the message.
		Configurable Parameters	Algorithm: MD5, SHA-1, SHA-256, SHA-512,...,
			Block size.
		Implementations	java.security.MessageDigest.
			Guava.
			...
	Authentication	Operations	Authenticate.
		Variable Context	Number of users in the system.
		Configurable Parameters	Mechanism: user and password, digital certificate, biometric, social ID, pin, matrix,...
			Password length.
			Password type: numeric, alphanumeric, special chars,...
		Implementations	Java Authentication and Authorization Service (JAAS).
			Spring Security.
			...
Usability	Logging	Operations	Log.
		Variable Context	Size of the message to be logged.
		Configurable Parameters	Format: plain text, XML, HTML,...
			Handler: console, file, database,...
			Level: trace, debug, info, warning, error,...
			Messages encrypted or not.
		Implementations	Log4J.
			LogBack.
			java.util.logging package.
			Simple implementation of the SLF4J API.
			...
	Contextual Help	Operations	Show help.
		Variable Context	Usage frequency.
		Configurable Parameters	Type of help: tutorial, wizard,...
			Kind of user: beginner, intermediate, advanced, expert,...
			User authenticated or not.
		Implementations	Java Wizard API.
			...
Performance	Caching	Operations	Store.
			Query.
		Variable Context	Size of the objects.
			Number of elements in memory.
			Load factor of the memory.
			Access frequency.
		Configurable Parameters	Maximum size of the memory cache.
			Cache type: local, remote,...
			Maximum life seconds.
			Eternal objects.
		Implementations	Java Caching System (JCS).
			Caffeine.
			...
Persistence	File Storage	Operations	Create file.
			Modify file.
			Delete file.
		Variable Context	Size of the file.
			Size of the data to be stored.
		Configurable Parameters	Type: binary, plain text, XML, JSON,...
		Implementations	java.io package....
			...
	Database	Operations	Store.
			Query.
		Variable Context	Size of the data to be store.
			Frequency.
		Configurable Parameters	Database type: relational o no relational.
			Data type: integers, floats, strings, objects...
		Implementations	MySQL
			MongoDB
			OracleDB
			...

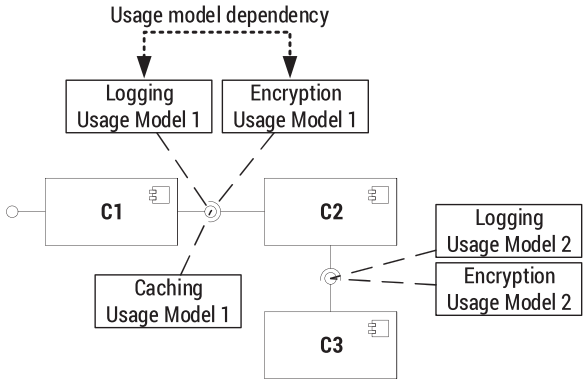

Fig. 3. Usage models applied to different interactions in an application architecture.

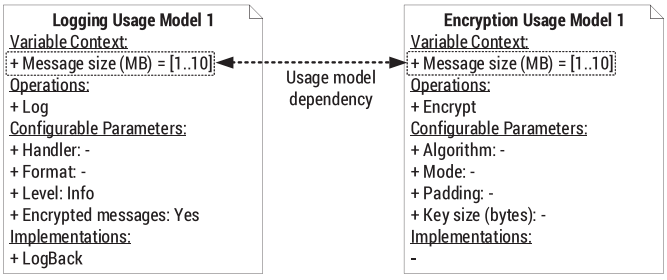

Fig. 4. Interaction between logging and encryption usage models.

FQAs Application Engineering Process (bottom of Fig. 1). The goal of the application engineering process is to generate FQA configurations based on the specific application's requirements, as well as on the selected objective function. These functions were defined during the domain engineering process and one of them is now selected to generate the optimum configuration (e.g., one that maximizes the energy efficiency). To do so, the application architect instantiates the FQAs variability model previously specified during the domain engineering process. This is done by selecting the features that satisfy the application's QAs requirements (`FQAs Configuration Model`). Often the application architect can only provide a subset of the usage model's values, so a partial configuration is generated in these cases. Both this partial configuration and the objective function are represented in an (`Extended CSP Model` that extends the CSP model of the domain engineering process with the partial configuration (`Partial Configuration Model`) and the selected objective function (`Objective Function`). Using a solver, the constraint satisfaction problem is resolved and an optimum and complete FQA configuration is automatically generated (`Optimum FQAs Configurations`), alleviating the architect's task of deciding between different configurations.

3. QAs characterization

In this section we cover the QAs characterization step. As summarized in Fig. 2, the first step of the QAs characterization is to identify, for each QA (security, usability, persistence,...), the functional behaviors (called FQAs) that must be present in the software architecture of the system in order to satisfy that QA. Then, for each FQA, a usage model is defined in terms of the variables that can affect both energy efficiency and performance. These variables can be divided into: (1) *operations* associated with the functionality (e.g., the encrypt/decrypt operations for the encryption functionality) where each operation may have a different impact on energy efficiency and performance; (2) variables that are part of the usage context of the application (e.g., object size, call frequency) and whose variation affects the energy efficiency and performance of the applications; (3) the parameters of the functionality that can be configured to be the optimum from the point of view of energy efficiency and performance (e.g., the algorithm used to encrypt/decrypt); and (5) the particular implementations in a programming language that provide the functionality (e.g., Java frameworks or libraries).

Summarizing, the usage model of an FQA is defined by means of a set of variables that can have a positive or negative effect on other QAs that are relevant for the system under development (energy efficiency and performance in this case), and the values that each variable can take. Table 1 shows the characterization for some of the most used QAs [3], their related functionalities and several variables of their usage models. We can observe that for each characterized QA, several FQAs are identified (e.g., the encryption, hashing and authentication FQAs for the security QA, or the logging and contextual help FQAs for the usability QA). We can also observe the existing variability for the usage models. For instance, the logging functionality of the usability FQA has a high degree of variability: messages can be logged in different formats (e.g., plain text, XML, HTML,...), or can be sent to different outputs (handlers) such as console, file, or database. Messages can also have different levels of severity (e.g., trace, debug, warning,...), and can be encrypted if need be. This characterization shows the large amount of information that needs to be taken into account when reasoning about the energy efficiency and performance of the system, since each characteristic can affect the energy efficiency and performance in different ways. Thus, without this characterization step and the formal representation of the information obtained (described in Section 4) it would be very difficult to generate optimum configurations based on the energy efficiency and performance QAs.

There is another important characteristic of the FQAs that needs to be considered. As stated, the energy efficiency and performance of an FQA in a particular application is determined by the usage model of the FQA. However, there are different places in a software architecture where the same FQA has to be injected, and the usage context in these places may be different. This means that the variables of the usage models can have different values to represent the different usage contexts. To cope with this, our characterization process builds a generic usage model that can be instantiated by the application architect multiple times, and in different parts in the same application. To illustrate this, Fig. 3 shows several usage models instantiated in different component interactions inside an application architecture. In the interaction between components `C1` and `C2` three functionalities have to be injected: logging, encryption and caching. Logging and encryption are also injected into the interaction between components `C2` and `C3` but here the usage models are different from those defined in the interaction between `C1` and `C2`, i.e., the values of one or more of the variables in the usage models are different.

Assigning specific values to the usage model variables can be a difficult task, since the values can be unknown to the application architect. Moreover, the application architect may want to analyze the QA level of all possible variants of the usage model. In those cases, our proposal will support the partial instantiation of the usage model, as we demonstrate in Section 7. An example of this partial instantiation is shown in Fig. 4 for the `Logging Usage Model 1` where the handler of the message (i.e., console, or file) and the format (i.e., plain text, or XML) are not provided by the application architect. The `Encryption Usage Model 1` in Fig. 4 also shows a partial instantiation where the algorithm, mode, padding, key size, and implementation framework have not been specified.

In addition, there can be dependencies between the usage models of different FQAs' functionalities. For instance, in order to encrypt the log messages, the encryption functionality has to be applied before the logging functionality, to the same messages in the same interaction (interaction between components `C1` and `C2` in Fig. 3). Thus, the values of the message size are shared in both usage models (see Fig. 4). Moreover, due to the interaction between the encryption and the logging functionality, encryption may require the values of the shared variables to be changed in the logging usage model. For instance,

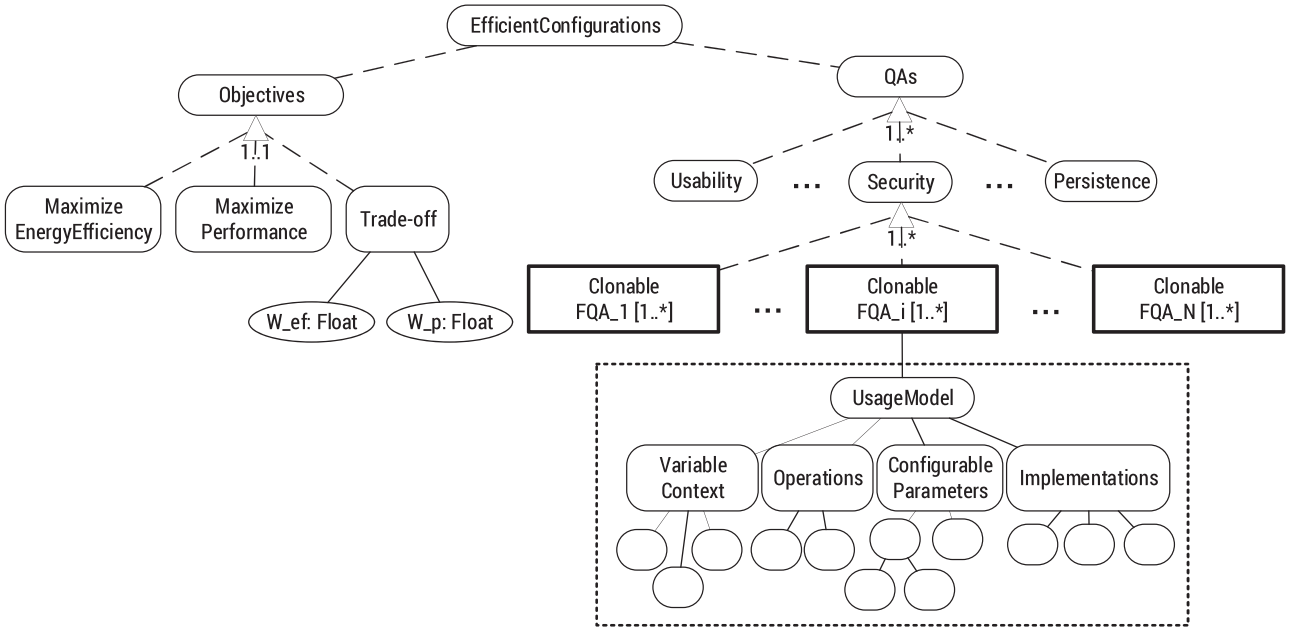

Fig. 5. Variability model schema of the QAs with their FQAs and the usage model.

depending on the encryption algorithm, the mode and the padding, the output encrypted message may vary in size compared to the original message. When this happens, the value of the message size variable in the logging usage model needs to be modified. Our proposal will manage this kind of dependency and will update the affected usage models accordingly.

4. FQAs variability modeling

Based on the domain information captured during the characterization of the QAs, in this step the domain experts model the variability of the QAs, their FQAs and the usage model as a part of an SPL.

Fig. 5 shows the schema of the resulting variability model. The variability model specifies at the second level the QAs (e.g., security, usability,...) and at the third level the functionalities of each of them (FQAs). First, each FQA is modeled as a clonable feature to allow different configurations of that functionality to be generated in different places of the application. A clonable feature (features with [1..*]) allows that feature to be instantiated multiple times, and all its sub-features can be configured differently for each instance [9]. Cardinality of the clonable feature ([1..*]) specifies the number of instances that can be generated, i.e., 1 as the minimum and * as the undefined maximum. Second, for each FQA all variables of its usage model defined in the last section are specified under the *UsageModel* sub-tree, including the `Variable Context`, the `Operations`, the `Configurable Parameters`, and the `Implementations`. Finally, the objectives that can be considered when generating a specific configuration are explicitly represented in the variability model. For instance, in our approach the application architect can select maximizing the energy efficiency over the performance, maximizing the performance, or achieving a trade-off between them by assigning a weight to each objective.

Figs. 6–8 are part of our FQAs variability model, but they are shown as separate models due to the lack of space to represent all the information in the same figure. Also, the FQA variability models shown in these figures are not complete in some of them, but readers can retrieve the complete version online.[2] Concretely, the encryption, logging, and caching functionalities are modeled by the clonable features `Encryption [1..*]`, `Logging [1..*]` and `Caching [1..*]`. This means that multiple instances of the usage model of these features can be generated straightforwardly, which in turn, enables different variable contexts, operations, configurable parameters, and implementations to be selected for each instance. The child features of any usage model feature are those variables of the FQA that affect the energy efficiency and performance. Note that some variables of the usage models can have dependencies between them. For example, in the case of the logging FQA (Fig. 7), not all frameworks provide the possibility of logging the message in XML format, such as the simple implementation of SLF4J (`Simple`). We therefore specify the constraint `Format.XML implies NOT Implementations.Simple` to prevent that framework from being selected when the format is XML. Similarly, we explicitly define the dependency with the encryption functionality when the log messages need to be encrypted (`EncryptedMessages implies Security.Encryption`). This means that by selecting the `EncryptedMessages` feature, an additional instance of the encryption functionality needs to be created in order to be configured, sharing the dependent variables of the usage model.

5. Energy efficiency and performance experimentation

The effect of each variable of the usage model on the energy efficiency and performance of the FQA is pre-calculated through experimentation [7]. As part of our proposal we measure the energy consumption of the different configurations of the FQAs. An example of this experimentation is presented in this section. The set-up of the experiments is detailed in Section 8.3.1.

Note that the goal of our approach is to analyze the energy consumption and performance trends of the FQAs when the values of the usage models' variables change. Thus, in our approach it is not relevant how accurate the energy consumption and performance values of FQAs are, but rather assessing whether one configuration consumes more or less than another, or has a lower or higher performance. For this reason, in this paper we assume that the FQA characterization and the analysis of the energy efficiency and performance can be performed at the domain engineering level, independently of a specific application. However, any experimental measure of energy consumption and performance must be considered as an estimation only, since the power and time consumed by an application will be different in successive

[2] Complete variability models are available in http://150.214.108.91/code/green-fqas

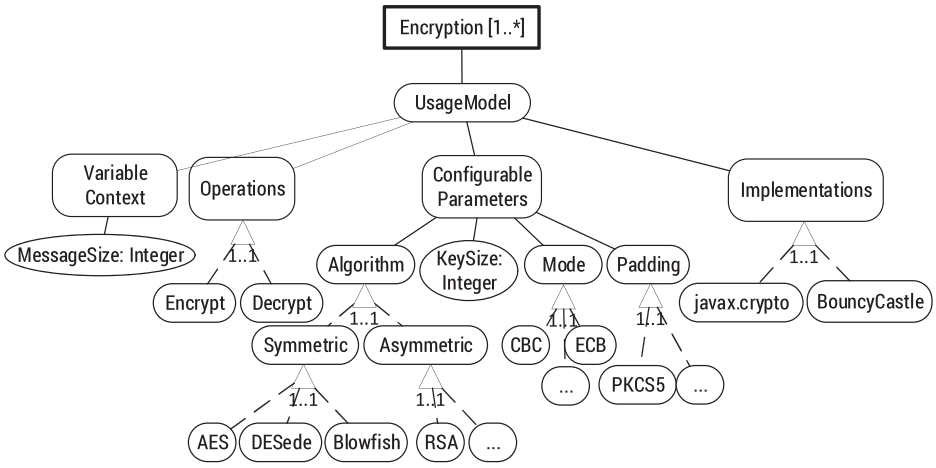

Fig. 6. Encryption FQA variability.

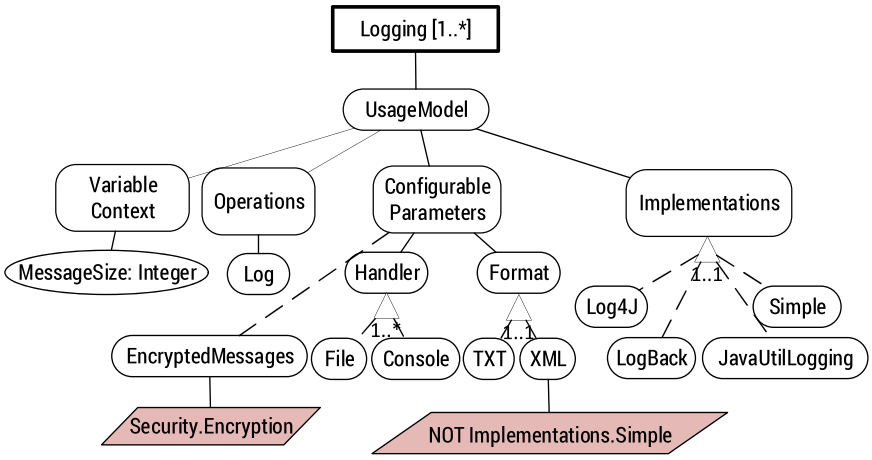

Fig. 7. Logging FQA variability.

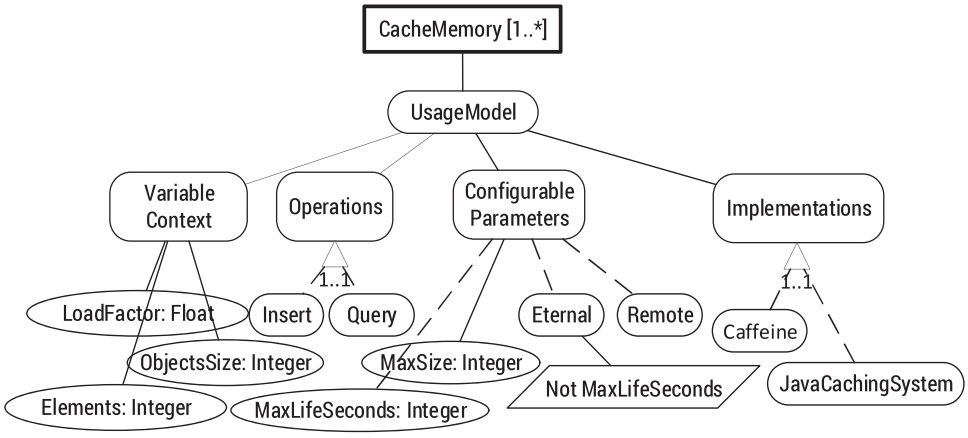

Fig. 8. Caching FQA variability.

executions. This is because there are many uncontrollable factors that affect power consumption and computational time (e.g., temperature, garbage collection), so we need to identify the power consumption and computational time tendencies.

Part of our experiment results for the encryption, logging, and caching FQAs are shown in Figs. 9–11, respectively. They show the energy consumption and execution times for a subset of the variants generated from the variability models specified in Figs. 6–8. For the encryption functionality we encrypted messages, varying the message size. We also used three different encryption algorithms of symmetric keys: AES, Triple DES (DESede), and Blowfish, provided by two different implementations: the javax.crypto package and the Bouncy Castle[3] encryption library. Similarly, experiments to characterize the logging functionality consisted of delivering a set of log messages using four different implementations of the SLF4J (Simple Logging Facade for Java)[4] logging API: the java.util.logging package directly provided by the Java JDK, the Log4J[5] and the LogBack[6] frameworks, and a simple

[3] https://www.bouncycastle.org/
[4] https://www.slf4j.org/
[5] http://logging.apache.org/log4j/1.2/index.html
[6] https://logback.qos.ch/

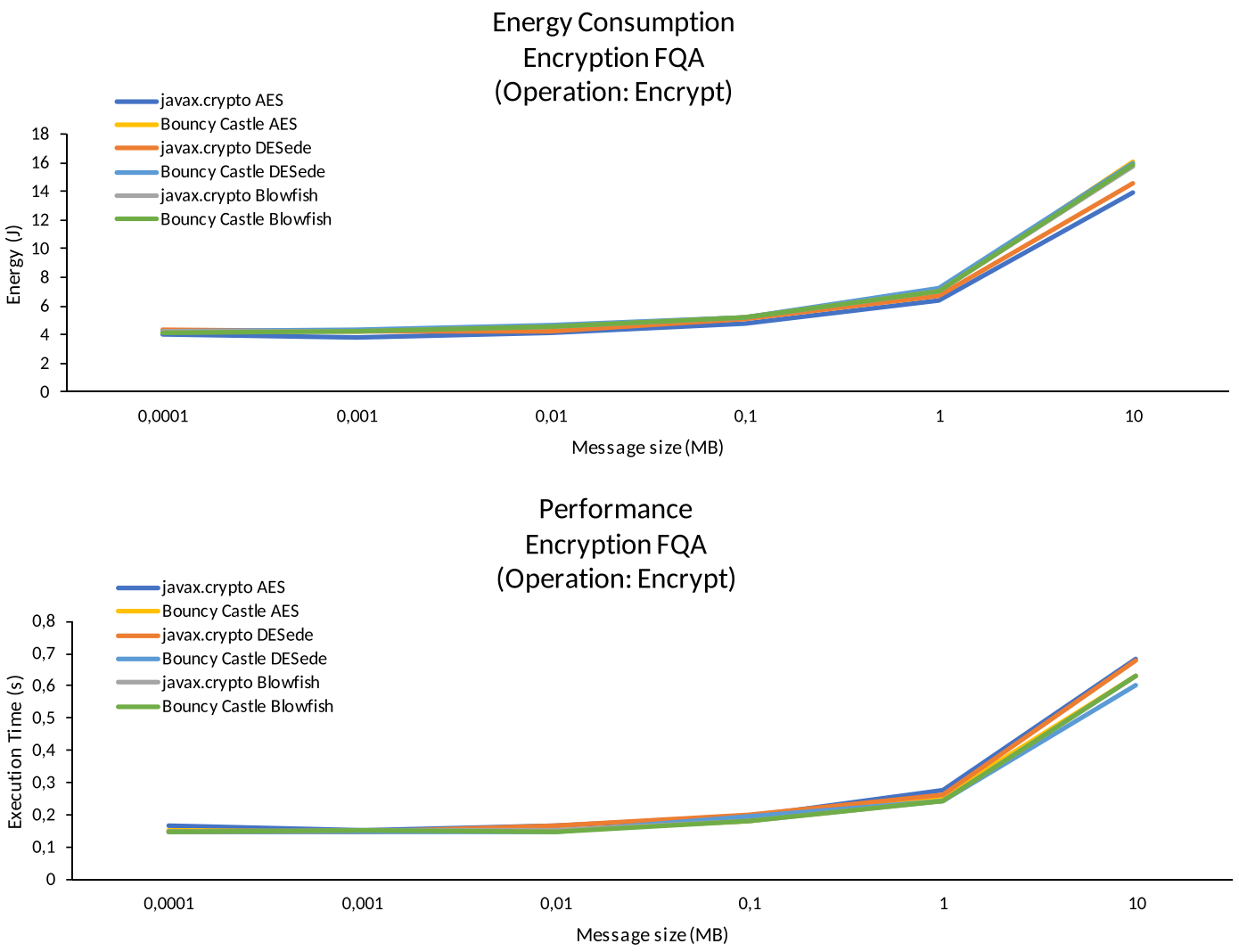

Fig. 9. Energy efficiency and performance: encryption FQA.

implementation of SLF4J[7]. We logged plain text messages varying the message size from 100 Bytes to 10 MBytes in powers of 10. For the caching functionality, experiments consisted of filling a memory cache with a specified maximum size until a specific load factor, and then inserting a string object of 1 MB in a cache with that load factor, i.e., we study the effect of the load factor parameter. We use an eternal, not remote cache, with different configurations for the maximum size parameter. To design the experiments, we automatically generate all the valid configurations from the variability models (see Section 8.4) and implement different tests by varying the variables and parameters until covering all generated configurations. Each test (or trial) in the experimentation considers a specific complete configuration, and is evaluated to measure its energy consumption and computational time (see Section 8.3.1).

Fig. 9 shows the results of the energy measurements (in Joules) and the computational time (in milliseconds) of the encryption functionality for different message sizes. In all cases, the mode selected is Cipher Block Chaining (CBC) and the padding is PKCS5. AES and Blowfish algorithms use an encryption key of 16 bytes, while the DESede algorithm uses an encryption key of 24 bytes. We observe similar energy consumption for the three algorithms with no significant variations. A very similar curve is generated for performance. Here, we can observe that the energy consumed by the encryption FQA is co-related with the execution time. However, even in this case where both curves have the same tendency we can observe that the greenest algorithm (AES with the javax.crypto implementation) has the worst performance. So, it is not trivial to select the best configuration to improve both energy efficiency and performance in the case of the encryption FQA.

Regarding the logging experiments, the results allow a more interesting analysis. Fig. 10 shows the experimentation results for different message sizes, using two different handlers: console and file. On the one hand, we can observe that the most eco-efficient configurations are those that use the simple implementation of the SLF4J API, while the configurations with the higher energy consumptions are, in all cases, those that use the LogBack framework. In general, Log4J and LogBack frameworks consume more than others because their purpose is to provide full functionality and a great performance regardless of energy consumption. The tendency for the energy consumption is more or less the same for all the alternatives evaluated, for logging the messages to console or to file. However, considering the performance, there are differences, depending on the message size, between logging to console or to file that are worth analyzing. Configurations that log the messages to console demonstrate the lowest performance as the message increases in size. The simple implementation of the SLF4J API (the greenest framework) is one of the frameworks with the worst performance for huge messages (10 MB), but the most efficient for short messages (less than 10 KB). Similar behavior is observed for the java.util.logging and Log4J implementations using the console handler.

[7] https://www.slf4j.org/apidocs/org/slf4j/impl/SimpleLogger.html

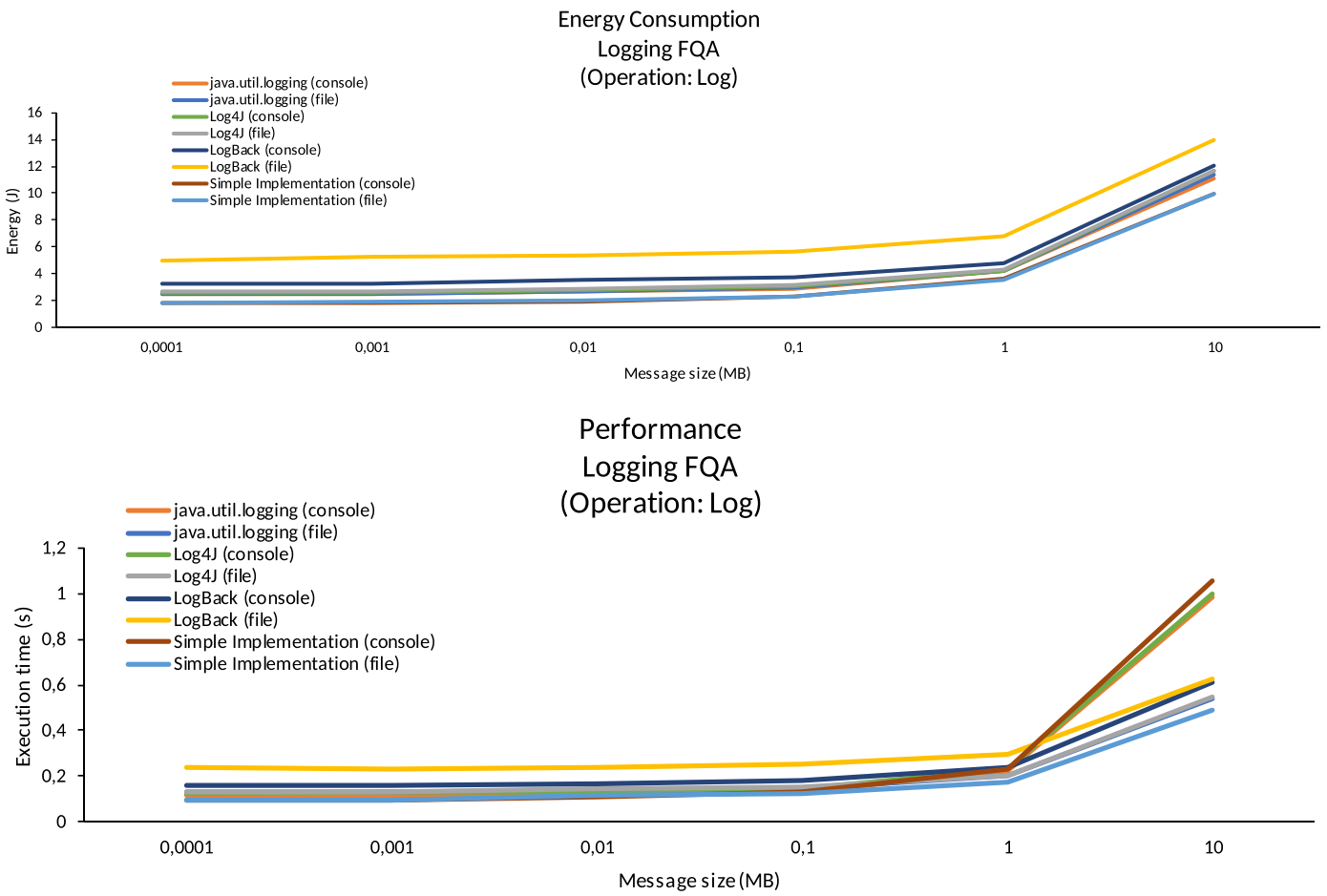

Fig. 10. Energy efficiency and performance: logging FQA.

The experiment results for the caching FQA show that in general for both energy efficiency and performance, it is better to use the minimum possible size for the cache, i.e., a smaller cache. However, identifying the best configuration is not an easy task, since tendencies of both energy efficiency and performance are similar but not trivial. For instance, if we have a cache with the capacity for 1000 elements and the load factor is actually 20% (i.e., 200 elements), filling the cache and inserting an element consumes 126.02 J of energy and takes 5.9 s, while in a cache with a maximum capacity for 400 elements, where the load factor is 50%, the operations consume 104.75 J and take 6.5 s. Thus, in these conditions, choosing a cache with a capacity of 400 elements will improve the energy efficiency by around 17%, but the performance will deteriorate by approximately 10%.

The effects of the usage model dependencies between FQAs on the energy and time consumption have also been measured as part of our experiments. For instance, for the existing dependency between the usage models of the logging and the encryption FQAs, our experimentation results show that the size of messages to be logged increases by up to 33% when they are encrypted. Both logging and encryption consider message size as part of the usage model, so this means that the usage models are automatically updated by our approach when the application architect selects interacting FQAs.

6. Representation in CSP

Our goal is to generate optimum configurations considering energy consumption and/or performance, we therefore need to formally represent the variability model and the experimentation results. Different approaches have been used to generate optimum product line configurations such as constraint satisfaction problems [10], or evolutionary algorithms [11], among others. In this section we will show how we represent the variability model and configurations as a Constraint Satisfaction Problem (CSP) [12], since CSP is one of the most used approaches to formalize variability models [10].

Our contributions with respect to other approaches, which also reason about variability models using CSP is that we integrate the information obtained from the experiments, as well as the dependencies between the usage models into the CSP model. With CSP we avoid the need to extend the metamodel of the variabiblity model or classical feature models or enrich it with attributes [6] to represent the experiment information and associate it with the different configurations. In contrast to other approaches such as evolutionary algorithms [11], CSP allows reasoning about variability, as well as other capacities of variability models and feature models like the generation of valid product configurations, or the quantification of the number of possible configurations [10]. In the case of evolutionary algorithms, generated configurations may not be optimal nor even correct since solutions are randomly generated by applying specific operators (e.g., selection, crossover, mutation), and configurations may not satisfy the tree and cross-tree constraint defined in the variability model. In those cases, an additional step to fix the generated configurations is required. Moreover, integrating the experiment information within the configurations is not an easy task because configurations are previously unknown.

A CSP problem is defined by a triplet (X, D, C), where X is the set of *variables*, D is a set of *domains* for the variables, and C is the set of *constraints* that must be satisfied.

6.1. Representation of the variability model and configurations

We formalize the variability model in CSP as follows:

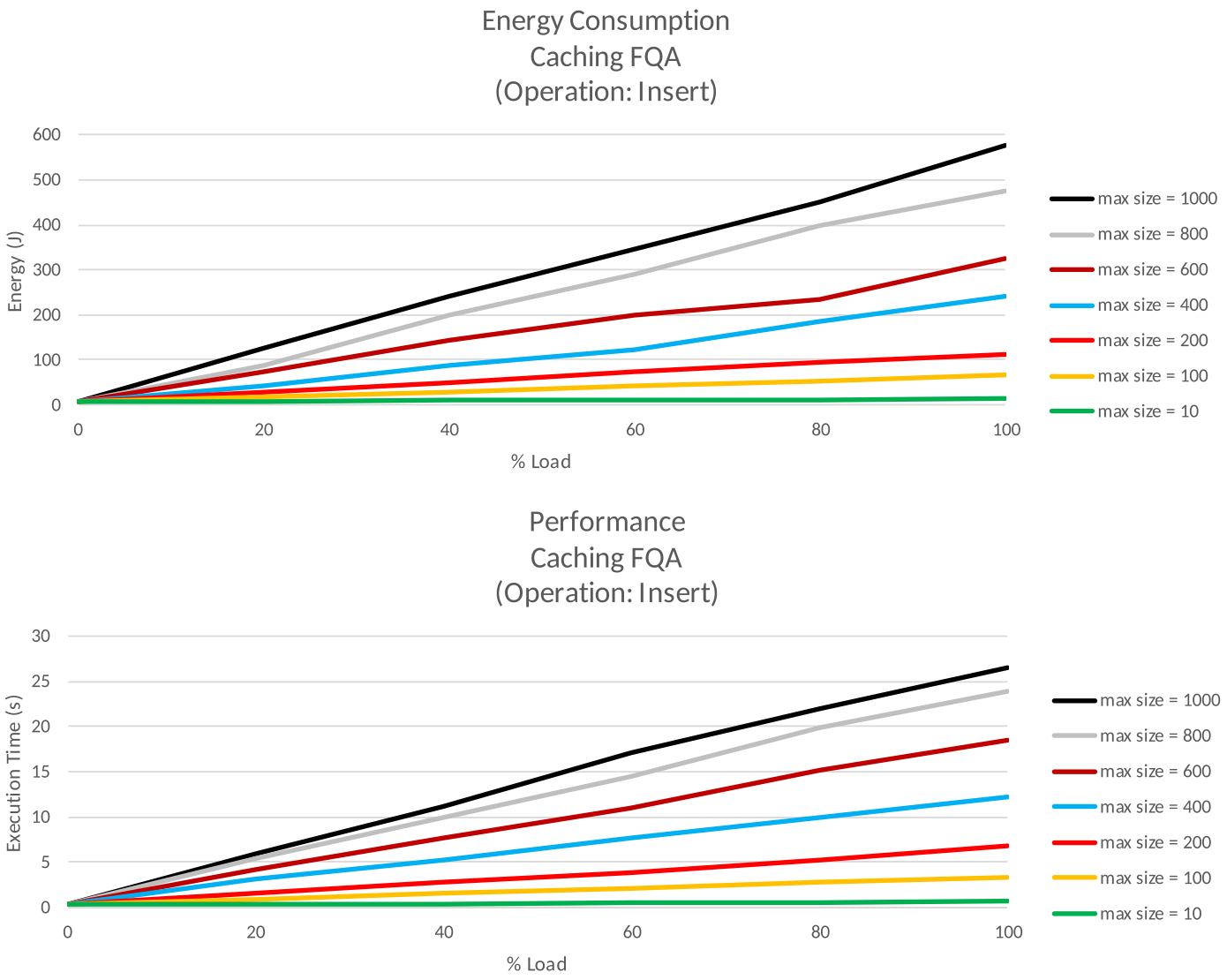

Fig. 11. Energy efficiency and performance: memory cache FQA.

- **Variables (X_{VM}):** X_{VM} is a proper subset of X ($X_{VM} \subset X$) and is composed by the following subsets of variables: $X_{VM} = X_F \cup X_V$.
  - X_F is the set of variables that represents the features of the tree in the variability model. For example, for the logging FQA we define the following set of variables: $X_{F_{logging}} = \{x_{usagemodel}, x_{variablecontext}, x_{messagesize}, x_{operations}, x_{log}, x_{configurableparameters}, x_{handler}, x_{file}, x_{console}, x_{format}, x_{txt}, x_{xml}, x_{implementations}, x_{log4j}, x_{logback}, x_{javautillogging}, x_{simple}\}$
  - X_V is the set of variables that represents those features which represent attributes (i.e., they take specific values such as the size of the message in the logging functionality). An example of this set for the logging FQA is: $X_{V_{logging}} = \{v_{messagesize}\}$.
- **Domain (D_{VM}):** $D_{VM} \subset D$, $D_{VM} = D_F \cup D_V$.
  - $D_F = \{0, 1\}$. The domain of the variables in X_F is $\{0, 1\}$ to indicate whether the feature is selected or not in a configuration. Formally: $\forall v \in X_F,\ v \in \{0, 1\}$.
  - The domain of the variables in X_V is the domain of the specific type defined in the variability model for each variable (e.g., the domain of the variable $v_{messagesize}$ is the set of natural numbers $\mathbb{N}$).
- **Constraints (C_{VM}):** $C_{VM} \subset C$, where C_{VM} is the set of constraints that models the relationships of the variability tree, as well as the cross-tree constraints. An example of this kind of constraint is $x_{log4j} + x_{logback} + x_{javautillogging} + x_{simple} = 1$, which guarantees that only one framework of logging is selected in a specific configuration. Another example is $x_{variablecontext} = x_{messagesize}$, which models the mandatory relationship (parent-child) between the `VariableContext` and the `MessageSize` features of the variability model (see Fig. 7). Finally, the constraint $x_{encryptedmessages} - x_{encryption} < 1$ models the dependency between the `EncryptedMessage` and `Encryption` features.

Configurations of the variability model are defined by the following set of constraints:

- **Constraints (C_C):** $C_C \subset C$, where C_C is the set of constraints that models the selections made by the software architect in a configuration of the variability model. For example, the following constraints model a partial configuration of logging that correspond to the `Logging Usage Model 1` of Fig. 4 (and the `Logging1` instance in Fig. 13)[8]:
  (1) $x_{usagemodel1} = 1$, (2) $x_{variablecontext1} = 1$, (3) $x_{operations1} = 1$, (4) $x_{configurableparameters} = 1$, (5) $x_{implementations} = 1$, (6) $x_{messagesize1} = 1$, (7) $v_{messagesize1} \geq \wedge v_{messagesize1} \leq 10$. (8) $x_{encryptedmessages} = 1$, (9)

[8] For simplicity we have omitted features/variables not selected in the configuration, the values of which are 0.

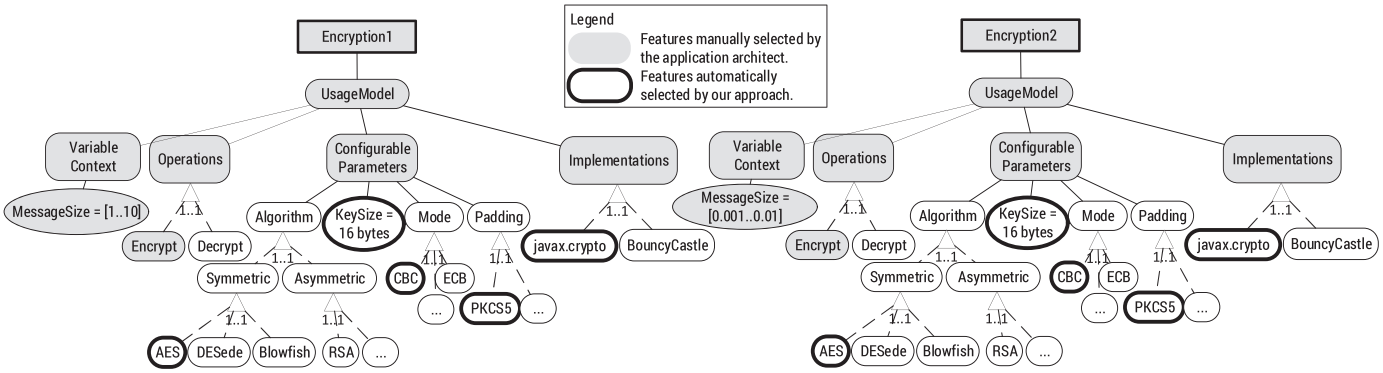

Fig. 12. Configurations of the variability model for the encryption FQA instances.

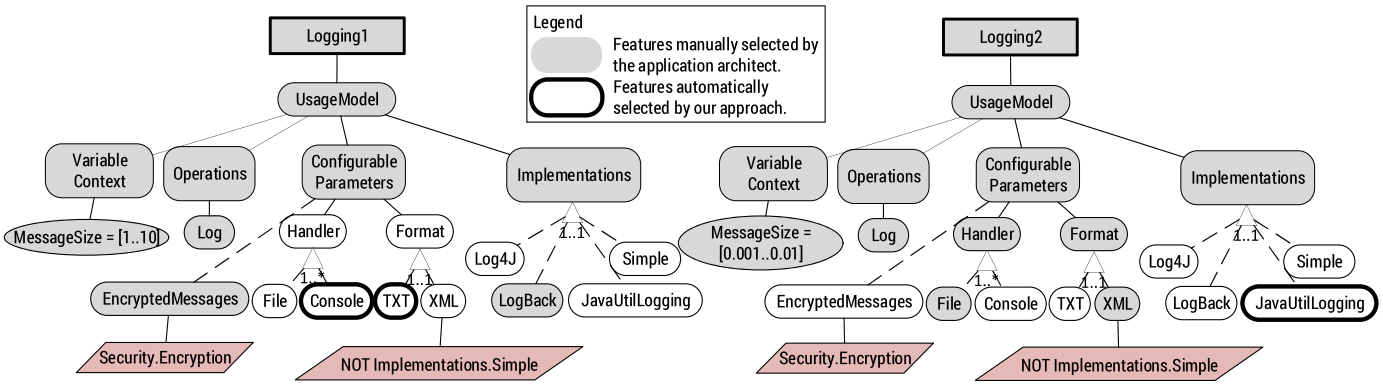

Fig. 13. Configuration of the variability model for the logging FQA instances.

$x_{logback} = 1$,

Sufix '1' after the name of the variables identifies the instance '1' of the *logging* clonable feature. This sufix number is incremented for each clone feature.

6.2. Representation of the energy efficiency and performance information

We extends the CSP model with the following definitions:

- **Variables (X_{QA}):** $X_{QA} \subset X$, $X_{QA} = QA_E \cup QA_P$, where:
  - QA_E is the set of variables that represents the energy consumed for each possible full configuration of an FQA. For instance, for logging, we have a variable e for each different configuration of the logging FQA. $QA_E = \{e_{configlog1}, e_{configlog2}, e_{configlog3}, ...e_{configlogN}\}$
  - QA_P is the set of variables that represents the execution time for each possible full configuration of an FQA. Similarly to the energy efficiency variables, for logging, $QA_P = \{p_{configlog1}, p_{configlog2}, p_{configlog3}, ...p_{configlogN}\}$
- **Domain (D_{QA}):** $D_{QA} \subset D$, $Dom(QA_E) = Dom(QA_P) = \mathbb{R}$. The domain of the variables in QA_E and QA_P together form the set of real numbers $\mathbb{R}$ that represent the energy consumed (in Joules) and the execution time (in seconds), respectively, of a configuration.
- **Constraints (C_{QA}):** $C_{QA} \subset C$, where C_{QA} is the set of constraints that defines the values of a particular configuration obtained in the experimentation (see Section 5). The C_{QA} constraints relate the variables in X_{QA} with each specific configuration. An example of this kind of constraint is: $x_{console} \wedge x_{txt} \wedge x_{log4j} \wedge x_{messagesize} \wedge v_{messagesize} \geq 0.01 \wedge v_{messagesize} \leq 10 \Rightarrow e_{configLog1}$, which together with the constraint $e_{configLog1} = 11,6736$ represents a complete instance of the usage model, assigning an energy consumption value to a configuration. Note that to simplify the CSP model we have chosen the worst case in the energy consumption for each interval of values in the message size.

6.3. Representation of dependencies between usage models

Initially we can suppose that by resolving the CSP for each partial configuration of each FQA, we can obtain the most eco-efficient configuration of each FQA. However, this reasoning cannot be performed in isolation for every FQA, because the configuration of one FQA can have a collateral effect on the energy consumption of another. For instance, an encryption algorithm may modify the size of the messages or the objects before storing them in a database (i.e., persistence FQA). Note that different configurations of the encryption algorithms produce encrypted objects of different sizes, and therefore the energy consumed by the persistence FQA will be different depending on the configuration of the encryption algorithm previously used.

To take into account these kinds of dependencies in our proposal, we formalize them using a new set of variables A:

- $A \subset X_V$ is the set of variables of the usage model whose values may be affected when a specific functionality of an FQA is applied (e.g., encryption functionality changes the size of the message). An example of the set A for the logging FQA is $A_{logging} = \{x_{messagesize}\}$. Note that the domain of the variables in A is the same as the domain of the variables in X_V.

Let us define UM_1 as the set of variables that compose the usage model of an FQA's functionality (e.g., logging), and UM_2 as the set of variables of the usage model of a second FQA's functionality (e.g., encryption). When there is a dependency between two FQAs (e.g., logging implies encryption) the intersection of the sets UM_1, UM_2, and A is not empty ($UM_1 \cap UM_2 \cap A \neq \emptyset$). In this case, when generating the configurations, our proposal: (1) uses the usage model's variables of that intersection, that is, uses the values of UM_1 in UM_2 for those common variables in A (e.g., the message size for encryption and logging); (2) updates those variables of A in the usage model UM_1 with the appropriate values obtained from the experimentation after applying the first

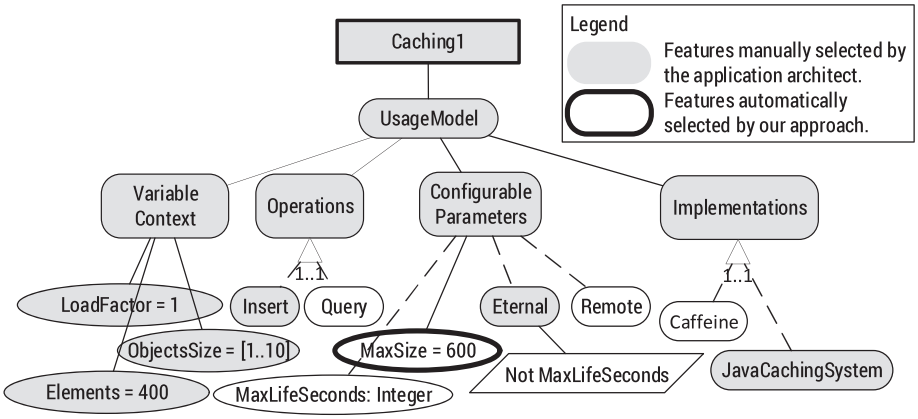

Fig. 14. Configuration of the variability model for the caching FQA instance.

functionality, i.e., an increase of 33% in the size of the message when it is encrypted; and (3) generates the configuration for the logging using the original usage model (UM_1) with the variables of A appropriately updated.

To summarize, our CSP model is formally defined as a tuple (X, D, C), where $X = X_{VM} \cup X_{QA}$, $D = D_{VM} \cup D_{QA}$, and $C = C_{VM} \cup C_C \cup C_{QA}$.

6.4. Representation of the objective functions

Defining the objective functions specified in the variability model (Fig. 5) in CSP is straightforward using the previously defined variables in X. The three objective functions are defined as follows:

- **Maximize energy efficiency.** This corresponds to minimizing the energy consumption of all FQA configurations:

$$\text{Minimize} \sum_{i=1}^{|QA_E|} e_i, \quad e_i \in QA_E \tag{1}$$

- **Maximize performance.** This corresponds to minimizing the computational time of all FQA configurations:

$$\text{Minimize} \sum_{i=1}^{|QA_P|} p_i, \quad p_i \in QA_P \tag{2}$$

- **Trade-off between energy efficiency and performance.** We achieve the trade-off by minimizing both the energy consumption and the computational time of all FQA configurations giving a weight (w_e and w_p) to each QA:

$$\text{Minimize} \sum_{i=1}^{|QA_E|} w_e \cdot \frac{e_i}{N_E} + \sum_{i=1}^{|QA_P|} w_p \cdot \frac{p_i}{N_P}, \quad e_i \in QA_E,\ p_i \in QA_P, \tag{3}$$

where N_E and N_P are normalization constants. By assigning different weights to each objective, all possible optimum configurations from the Pareto optimal/efficient solutions can be generated [13].

7. Generating optimum configurations

In this section we detail the application engineering process of our proposal. The goal is to generate optimum configurations for the FQAs required in concrete applications. By reusing the domain knowledge already identified and modeled by the domain experts during the domain engineering process, the application architects would have to identify the QAs required by their applications and instantiate the already existing usage models with those values specified in the requirements. In the FQAs variability model the architect also selects the objective function that will determine the best configurations based on energy efficiency and/or performance.

As illustrated in Fig. 3, first, for each FQA required by the application, the software architect needs to identify the join points (or interactions) in the application architecture where the FQA will be incorporated. Second, for each join point the architect creates an instance of the usage model for the injected FQA. This implies having to provide the values of the variables that compose the usage model, i.e., specifying how the FQA will be used in that specific part of the application. This is done by selecting, in the variability model, those features specified in the application's requirements and providing the required values. The selections made by the software architect can generate a complete or a partial configuration of the usage models. When the software architect provides a complete configuration for an FQA (i.e., values for all the parameters of the usage model are provided), the configuration generated may not be the best according to the objective function, even though it fulfills the specified requirements. So, in our approach it is more useful to partially instantiate the usage models, so that the most efficient configuration can be automatically completed. In many cases, the software architects' knowledge allows him to provide only a subset of the usage model's values, so only a partial configuration can be generated.

The CSP model defined in the domain engineering process is completed with the partial configuration and the objective function selected by the application architect. The CSP model, in the application engineering process, includes: (1) the selections made by the application architect represented as the set of constraints C_C defined in Section 6; and (2) the selected objective function. Using a CSP solver, the constraint satisfaction problem is resolved and an optimum and complete FQA configuration is automatically generated (`Optimum FQAs Configurations`), alleviating the architect's task of deciding between different configurations.

Let us suppose the application architect wants to maximize the energy efficiency of the configurations over the performance, and thus, has selected that objective function. Figs. 12–14 show, respectively, a configuration of the variability model with two instances for the logging functionality, two instances for the encryption functionality, and one instance for the caching functionality. All instances are customized based on the usage models described in Fig. 3 in Section 3.

In the first interaction between components C1 and C2, the application architect has provided, for logging (see `Logging1` instance in Fig. 13), a partial configuration of the `Logging Usage Model 1` presented in Fig. 4), specifying a message size between 1 MB and 10 MB, and selecting the option for encrypting the message and a specific framework: LogBack. Taking into account the dependency between logging and encryption, a new instance for the encryption FQA needs to be generated (see `Encryption1` instance in Fig. 12), based on the values of the variables shared with the logging usage model (`Encryption Usage Model 1` in Fig. 4). Concretely, using a message size of between 1 and 10 MB. With these values, the best configuration based on maximizing the energy efficiency (one of the objectives), is

using the AES algorithm provided by the javax.crypto package with an energy consumption around 13.98 J in the worst case (10 MB). This configuration creates encrypted messages of size 13 MB (i.e., an increment of 33%). Thus, our proposal updates the original usage model for logging with the new value of the message size variable and then generates the optimum configuration for logging. In this case, our proposal will select the console handler and the text plain format (`TXT`) that is the most efficient configuration (in terms of energy) for the updated usage model when the LogBack framework has been selected.

In the second instance of logging (`Logging2` in Fig. 13), the architect has provided a total configuration, but has not selected the implementation to be used. In this case our proposal will select the library java.util.logging because this is the most energy efficient option that fulfills the constraints of the variability model, i.e., it is the most energy efficient framework that allows logging message between 1 and 10 KB to file in XML (see constraint in Fig. 7). A similar configuration to the first has been chosen for the second instance of encryption (`Encryption2` in Fig. 12), but in this case, there is no dependency between the logging and encryption usage models for the second instance of the encryption functionality

Finally, in the instance of the caching functionality (Fig. 14), the architect has configured the caching FQA as follows: it is expected to store around 400 elements between 1 and 10MB of size, and the cache will normally be full (i.e., the load factor is 100%). The cache will be eternal (elements do not have a maximum life time), and local (i.e., not remote). In this context and with these requirements, the most energy efficient configuration is having a cache with a maximum size of 600 elements. The cache with a maximum size (600 elements), should be used which implies that the load factor will be 66.67%. Filling the cache and inserting an element in this case consumes around 217.28 Joules, instead of using a always full (i.e, maintaining the load factor at 100%) cache with a maximum size of 400 elements, which consumes 240.28 Joules. This supposes improving the energy efficiency by 9.58%, which is the best configuration in the current context. Another configuration with a capacity of 800 or 1000 elements does not benefit energy efficiency as shown in Fig. 11.

8. Evaluation and threats to validity

In this section we evaluate the different steps of our proposal, and discuss the threats to validity and lessons learnt from our proposal and evaluation.

8.1. Characterization of the QAs

We discuss the completeness and correctness of the characterization of the QAs. The completeness intends to explore whether all behavioral properties of the QAs and all variables and parameters that affect the energy efficiency and performance have been identified. The correctness determines whether the characterization is done for the appropriate QAs and the identified functionality and parameters are appropriate.

Table 1 in Section 3 only shows a subset of the characterization of 4 QAs and a total of 8 FQAs, obtained from different studies [3,14]. Completeness of the characterization not only depends on the knowledge available for QAs from the domain experts, but also on the variables and parameters that the different frameworks, which implement the FQAs, provide, and can affect the energy efficiency and performance. Normally, any parameter related to the functionality being characterized is considered (i.e., we consider all parameters that appear in the API or framework documentation), even when the parameter does not directly affect the energy consumption or performance. This is because, until the experimentation has been carried out, the effects of a given parameter on the energy consumption or performance are unknown; and although some parameters may not greatly affect on energy consumption or performance, they may influence other parameters that do.

One threat to the internal validity of QA characterization is that the information we have used to characterize QAs may not be accurate. For this we have consulted papers written by domain experts and also the documentation provided by the APIs and frameworks that implement QAs. Now, let us consider that domain experts do not provide a complete and correct characterization of QAs or that the documents we have used to characterize QAs have errors. In this case, our approach will not consider that parameter as part of the variability model and all the generated configurations will have that parameter with a fixed value provided by the API or framework (e.g., a default value). Thus, keeping that parameter in the experimentation fixed for all configurations does not affect the results presented in this paper. The threats to external validity are determined by the lack of experimentation in industrial settings. We try to mitigate this threat by considering those APIs and frameworks that are most used in industrial case studies.

8.2. Variability modeling and CSP formalization

We have specified the variability model using the Common Variability Language (CVL) [15], which allows not only the variability to be specified as feature models do [16], but also resolves the variability over MOF-compliant models. CVL enhances feature modeling and automates the production of product models from an SPL [17]. Apart from the advantages it offers (orthogonal variability, architecture variability resolution, MOF-compliant, variable and clonable features,...) [4,18], we have chosen CVL because all characteristics of feature models can be expressed in CVL [16] and it was recommended for adoption as a standard by the OMG.

Moreover, CVL as feature models, can be easily mapped to logical constraints, as we formalize in CSP in Section 6 [10]. The formalization allows us to resolve the CSP problem using a CSP solver to guarantee that our proposal generates the same valid configurations. It also guarantees that those configurations are the best, based on the experimental information that we have integrated within the CSP model. Here, we have used Clafer,[9] a general-purpose lightweight modeling language, and CHOCO,[10] a Java library for constraint programming, to implement the variability models and configurations as CSP problems, and resolve them. Configurations generated by CHOCO satisfy the partial instantiation of the usage models introduced by the software architect. This ensures that the configurations generated by our proposal are the most efficient in terms of energy and performance, depending on the objective function selected by the architect.

One threat to internal validity is that the correctness of our process principally relies on the variability model. If this model is not correct then it may affect the causality of our conclusions. We have checked with Clafer and CHOCO that the variability model is structurally correct as it does not contain dead features, false optional features or wrong group cardinality, the typical defects of variability models, as shown in [10]. We have not identified any threat to external validity regarding our experimental environment, because for the CSP formalization we have used the latest versions of Clafer and CHOCO.

8.3. Experimentation

In this section we first discuss the reliability of the experiments discussed in Section 5 and whether they can be replicated with the same results.[11] Then we discuss the internal and external validity of the experimentation. The internal validity examines whether the experiment results are influenced or not by other factors apart from those considered in the experiments. The external validity analyzes whether the

[9] http://www.clafer.org/

[10] http://choco-solver.org/

[11] All resources (models, code, and experimentation results) all available in http://150.214.108.91/code/green-fqas

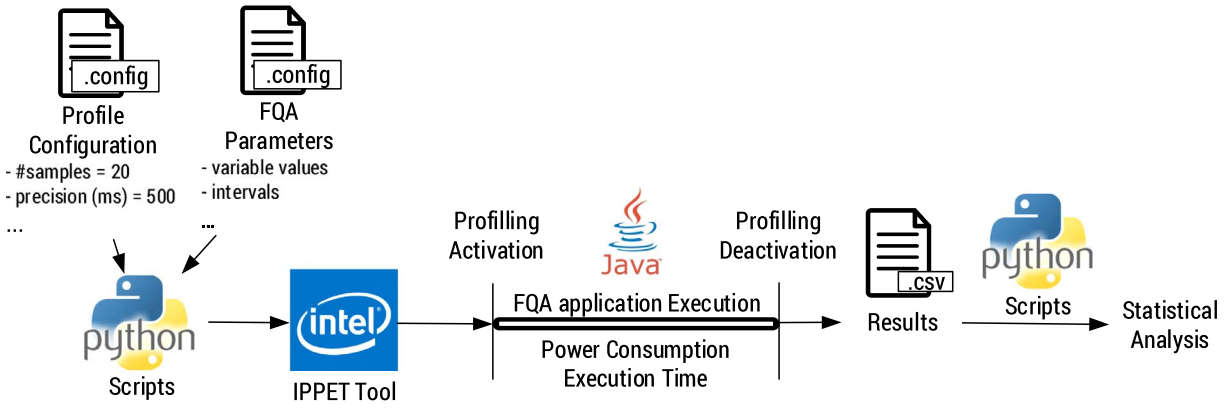

Fig. 15. Profiling platform designed for experimentation.

results obtained in the experimentation can be generalized or not.

8.3.1. Experimentation set-up

The experiments were performed on a desktop computer with Intel Core i7-4770, 3.40 GHz, 16GB of memory, Windows 10 64 bits with the high performance option activated, and Java JDK 1.8. The profiling platform designed to gather energy consumption and performance information in our approach is shown in Fig. 15. We use the IPPET (*Intel Platform Power Estimation Tool*)[12] tool that monitors the energy consumption and the computational time of the CPU at process level using the Intel Model-Specific Registers (MSRs). As a result, IPPET generates a CSV file with the estimated power (in Watts) for each timestamp (with the specified precision in milliseconds — 500 ms) of the execution of the process, which is repeated 20 times. We have implemented some Python scripts[13] to run IPPET over all our experiments, extract the information, and calculate averages and standard deviations of energy consumption (in Joules) and execution times (in milliseconds).

8.3.2. Accuracy of the results

We choose the IPPET tool for two reasons: (1) we need to perform the measurement at the code and/or process level (e.g., measures of operations, parameters) and not at the device level, so taking the measurements with more precise hardware-based tools is more difficult because it is not be easy to identify the part of the software responsible of this consumption; and (2) IPPET can be executed as a command line process apart from its graphical user interface, and thus, it is possible to automatize the experiments and repeat them multiple times. This obtains more precise values than manually measuring them with other graphical tools such as Joulemeter[14] or pTop.[15]

Although hardware solutions provide more precise measurements, note that our goal is not to calculate the exact values obtained for each different configuration of FQAs. As we discussed in Section 5, we are interested in identifying energy consumption and performance variations and tendencies when the usage models vary.

8.3.3. Generalization of results

By performing the same experiments in another environment (e.g., another computer), the actual experimental values may vary due to many factors (mainly hardware), but the comparative results should remain correct. In specific cases where the execution environment is very different from a general-purpose computer, such as supercomputers or special-purpose computers designed to perform some processing faster, the results of the experimentation may be very different. In that case our proposal will not generate the same configurations. However, our approach will generate the best configurations based on the energy efficiency and performance values obtained for that specific environment. To mitigate the external validity we have performed a subset of the experiments on a different computer, and have obtained similar results.

8.4. Generation of configurations

In this section the scalability of our proposal from the point of view of both the domain and the application engineering processes is discussed. In addition, the benefits of our proposal are discussed by analyzing the improvements in energy efficiency and performance of the FQA configurations when they are generated with our proposal.

8.4.1. Scalability

In this section we study the scalability of our proposal. Generating all the configurations and performing all the experiments within a reasonable time may sound an intractable task for real world sized systems. However, it is important to clarify that our approach is not applied to complete systems or applications, but rather to real world implementations of the FQAs. The degree of variability of FQAs and their usage models (variables, operations, frameworks, and parameters) is usually much smaller than the variability found in complete systems. For example, we did not find any FQAs with more than 5 contextual variables and 5 operations, while 10 is a high number of frameworks to be studied, each one with around 15 or 20 different configurable parameters. Note that the size of the application does not affect our approach, but the size of the FQAs, which is often very limited as shown in FQAs' characterization of Section 3 does affect it. The evaluation was carried out on the same computer as the experimentation (see Section 8.3.1), and we have used Clafer and Choco tools to generate all possible configurations from the variability models.

From the point of view of the domain-engineering process, Table 2 shows how our approach scales when generating all the variants specified by the variability model of each FQA. We study the size of the usage model, i.e., total number of features, and what type of feature (variable, operation, parameter, and framework) implies a higher number of valid configurations and takes more time to be generated. Note that, for integer variables, the number of configurations would increase exponentially if we considered the complete integer domain. However, this is avoided in our approach by splitting the range of integers into intervals in each experiment. Taking into account what the contextual variables are representing (e.g., message sizes), we can assume that 10 intervals are enough to cover the whole variable range (e.g., dividing the values in multiple of 10). Note that in most cases 5 intervals would be enough, considering the meaning of the contextual

[12] https://goo.gl/noZsYk
[13] http://150.214.108.91/code/energy-meter
[14] https://goo.gl/pND9RY
[15] http://mist.cs.wayne.edu/ptop.html

Table 2
Scalability of the domain engineering process.

FQA	#Variables[a]	#Operations	#Parameters[b]	#Frameworks	Total #Features	#Configurations	Time (s)
Contextual Help	1 (1,10)	1	7	1	10	160	0.52
Hashing	1 (1,10)	1	5 (1)	2	9	800	0.60
File Storage	2 (1,10)	3	4	1	10	1200	0.65
Logging	1 (1,10)	1	12	4	18	3300	1.04
Database	2 (1,10)	2	6	3	13	4800	1.08
Authentication	1 (1,10)	1	13 (1)	2	17	20,600	2.64
Encryption	1 (1,10)	2	13 (1)	2	18	24,000	3.21
Caching	4 (1,5)	2	5 (2)	2	13	187,500	22.97
Caching	4 (1,10)	2	5 (2)	2	13	12,000,000	1354.74 (~ 22 min.)
Test 1 (base case)	1 (1,10)	1	1	1	4	20	0.47
Test 2 (operations)	1 (1,10)	5	1	1	8	25,600	2.64
Test 3 (frameworks)	1 (1,10)	1	1	10	13	51,200	2.64
Test 4 (variables)	5 (1,10)	1	1	1	8	200,000	20.43
Test 5 (parameters)	1 (1,10)	1	20	1	23	256,000	31.90
Test 6 (worst case)	1 (1,10)	5	20	5	31	6,400,000	783.87 (~ 13 min.)
Test 7 (reduced intervals)	1 (1,5)	5	20	5	31	3,200,000	431.69 (~ 7 min.)

[a] Integer variables are divided in 10 intervals (1,10) or 5 intervals (1,5).
[b] In parentheses the number of integer parameters that have been divided in 10 and/or 5 intervals (same intervals as contextual variables).

variables in the FQAs.

First, we generate all possible configurations for each FQA characterized in Section 3 (rows 1–9 in Table 2). The values in these rows reflect real values taken from real frameworks that implement those FQAs. We can observe that, in general, the number of possible configurations of the FQAs is not very high. In most cases the configurations are generated in a few seconds. For example, the variability model of the Encryption FQA has 18 features that can generate up to 24,000 different configurations in 3.21 s. Only the Caching FQA has higher execution times when the values of the integer variables are divided into 10 intervals. In this case configurations are generated in minutes rather than seconds. This difference is due to the larger number of contextual variables and parameters that are integer (4 variables and 2 parameters). This increases the number of different configurations up to 12,000,000 and takes on average 22 min to generate all of them. However, despite the longer execution time it has to be remembered that this process is done only once by the domain expert. Moreover, the execution times can be reduced considerably (22.97 s) if the number of intervals is reduced from 10 to 5 (see row 8).

In order to check whether other parts of the usage model affect the scalability of the domain engineering process, we carried out some additional tests where 'prepared' numbers were used to test different situations in which we are interested (rows 10–16). Thus, Test 1 is the base case, the simplest one, with 1 variable, 1 operation, 1 parameter and 1 framework. Test 1 is compared with tests 2–5 where each column takes a value estimated as a reasonable maximum (according to the real frameworks previously studied). With these tests it is possible to see that in effect the number of integer contextual variables (with 10 intervals) is the element that affects the number of configurations the most (see Test 4 in comparison with Test 1–3 and Test 5).

The time to carry out the experiments of the FQAs studied in Section 5 depends on several factors such as the computer's performance, the size of the specific parameters of each FQA (e.g., the message size), the number of executions done for each experiment, the measurement tools running along the experiments, and the number of configurations covered by each experiment. Assuming that each experiment covers only one configuration, Table 4 shows the average time to perform one experiment of the Logging, Encryption, and Caching FQAs, and the average time to perform all the required experiments to cover all the configurations of the FQAs. The average execution time of an experiment is similar for the three FQAs (around 6 s), while the time to perform all experiments covering all possible configurations depends a lot on the number of configurations. For instance, the Logging FQA needs around 6 h to perform all experiments that cover its 3300 possible configurations, while the Caching FQA needs around 15 days to perform the experiments covering 187,500 possible configurations. The important issue to highlight here is that all the configurations and experiments are done only once by the domain experts, and then the experiment results can be reused in each application that needs to incorporate a configuration of the FQAs.

Additionally, from the point of view of the application engineering process the number of generated configurations is smaller than the one considered in Table 2. Table 3 shows how our approach scales when the application architect creates partial configurations of the FQAs, through their selections in the variability models. Firstly, we present the values corresponding to the partial configurations in Figs. 12–14 (top of Table 3). Secondly, for different FQAs we calculate the times for other partial configurations that were created considering two different scenarios identified as the most usual ones: (1) the application engineer needs help to decide the best framework to be used and the best configuration of that framework, but they know the context in which the application will be used, i.e. they know the values of the contextual variables and of the operations, and (2) the application engineer knows the framework to be used but it does not know the contextual information or how to configure it to obtain the greenest solution (rows labeled with (*f*) next to the number of selected features). As shown in Table 3 the execution times are always less than 2 s, except for the Caching FQA where, as seen previously, the number of integer variables increases the number of configurations considerably. Even in this case, however, the execution time is less than 1 min.

Finally, finding the optimum configuration is a search problem that has a quadratic $O(n^2)$ complexity.

8.4.2. Benefits of our proposal

To evaluate the benefits to energy efficiency and performance that a software developer can obtain when different configurations are considered, we have chosen a subset of all configurations for an FQA and have compared them in order to check if it makes sense to use our process to find an optimum configuration. Tables 5 and 6 show a set of configurations of the logging and encryption FQAs, respectively, to be compared. We divide the variable context parameter (message size in MB) into two intervals since observing the tendencies in Figs. 9 and 10 there are no significant changes inside those intervals. For each framework, we present two configurations, with their energy consumption (in Joules) and their computational time (in milliseconds). For each interval of the variable context, we highlight, in green, the most energy efficient configuration, while the worst configuration is shown in red. We also show, in bold, the configuration/s with the best performance, while the worst configuration in performance is shown in strikethrough text.

Table 3
Scalability of the application engineering process.

FQA conf.	#Features	#Selections[a]	#Config.	Time (s)
Fig. 12 (Encrypt1)	18	2	1200	0.79
Fig. 12 (Encrypt2)	18	2	1200	0.79
Fig. 13 (Logging1)	18	4	45	0.56
Fig. 13 (Logging2)	18	4	30	0.53
Fig. 14 (Caching1)	13	6	300	0.56
Test1 (Cont. Help)	10	2	16	0.51
Test2 (Cont. Help)	10	1 (f)	160	0.55
Test3 (Hashing)	9	2	80	0.52
Test4 (Hashing)	9	1 (f)	400	0.57
Test5 (File Storage)	10	3	4	0.50
Test6 (File Storage)	10	1 (f)	1200	0.66
Test7 (Logging)	18	2	330	0.68
Test8 (Logging)	18	1 (f)	900	0.69
Test9 (Database)	13	3	24	0.52
Test10 (Database)	13	1 (f)	1600	0.8
Test11 (Authentic.)	17	2	2060	1.09
Test12 (Authentic.)	17	1 (f)	10,300	1.72
Test13 (Encryption)	18	2	1200	0.79
Test14 (Encryption)	18	1 (f)	12,000	1.98
Test15 (Caching)	13	5	15,480	3.53
Test16 (Caching)	13	1 (f)	7,200,000	889.96

[a] (f) indicates that the selected feature corresponds with a specific framework, otherwise the selections correspond with the variables and operations.

It can be observed that, for the logging FQA, in general, the greenest configurations are those using the Simple Implementation framework, as discussed in Section 5. However, for large messages (10 MB), the Simple Implementation framework demonstrates the worst performance when messages are sent to the console.

Let us consider the following scenario, a developer has to choose between a configuration of the logging FQA to log the message to file. Normally, he would simply decide to use the most popular framework (e.g., LogBack) that in this case corresponds to the worst configuration in both energy efficiency and performance. However, our proposal will choose the Simple Implementation framework, obtaining an improvement of 63% in energy consumption (a decrease in energy consumption from 5.31 J to 1.99 J), and 55% in performance (from 237 ms to 106 ms), for messages of between 100 B and 100KB; while for messages of between 100 KB and 10 MB the improvement is around 25% in energy consumption (a decrease in energy consumption from 10.34 J to 6.77 J), and 29% in performance (from 462 ms to 328 ms).

In the case of the encryption FQA, the most energy efficient configurations are the worst in performance, as highlighted in Table 6; while the best configurations in performance are those with less energy efficiency. So, applying our proposal to select the best configuration in energy efficiency (i.e., the AES algorithm provided by the javax.crypto library) saves between 9% and 12% in energy, in comparison with the worst configuration (i.e., the algorithm DESede provided by the Bouncy Castle framework). However, those optimum configurations in energy are the worst in performance, so we would lose between 8% and 12% in performance with respect to the best configurations in performance, when choosing the most energy efficient configuration.

A similar analysis can be applied to the caching FQAs, as discussed in Section 5. However, in the case of the caching FQA, as observed in tendencies, in Fig. 11, the most energy efficient configurations are also the best ranked in performance. So, the selection made by our proposal is straightforward in this case.

One threat to internal validity arises through the use of the clonable features. Thanks to its clonable features, our proposal allows multiple instances of an FQA to be generated for the same application. However, there are frameworks, such as the logging frameworks used in this paper, that under the SLF4J API do not allow different frameworks to be used in the same application. This particular case can be easily solved in our proposal by including a new constraint that forces the selection of the same logging framework for all instances. In addition, we can also analyze the logging configurations selected for those instances of the usage models and perform a trade-off between them to select the best configuration in common, i.e., we can take into account the energy efficiency and performance of all instances of the logging usage models to select the best framework.

Table 4
Scalability of the experimentation.

FQA	#Config.	Time	#Config.	Time
Logging	1	6760 ms (6.76 s)	3300	6.2 h
Encryption	1	5085 ms (5.09 s)	24,000	33.9 h
Caching	1	6825 ms (6.83 s)	187,500	355.5 h (~ 15 days)

9. Related work

In this section, we comment on the related work on characterization, modeling, experimentation, and configuration of the QAs and their relationships with the associated FQAs. We principally consider the relationships of the FQAs with energy efficiency and performance.

Despite the fact that the performance QA has been widely studied in the literature [19–22] as one of the main QAs to take into account when developing an application, energy efficiency has recently become an important QA to also be considered [2]. Jagroep et al. [2] propose energy efficiency as a QA, focusing on usage resources such as software utilization, energy usage and workload. Thus, there are several examples of work which analyzes the energy efficiency of different aspects of the applications [23–25]. However, little work has been done about the relationships between the energy efficiency and other QAs, and even less on the relationships between the functionalities (i.e., FQAs) required to satisfy traditional QAs (e.g., security, usability) and the energy efficiency of that functionalities and their different configurations [26].

First, regarding the characterization of the QAs. There are some approaches that focus on identifying the functionalities (FQAs) and parameters related to the QAs of an application [3–5,27]. However, none of them relate the energy efficiency and performance with the FQA functionalities. On the one hand, some papers identify FQAs and the necessity of managing their variability. For instance, in [3], the authors analyze around 500 non-functional requirements from different specifications of industrial applications and identify that most of the so called non-functional requirements are not really non-functional because they describe functional behavior of the application. In [5], the authors identify functionalities related to the usability FQA that affect the software architecture. However, they do not analyze the relationships with different QAs such as energy efficiency and performance. On the other hand, those papers that do analyze the relationships between different QAs do not take into account the recurrent functionalities offered by the FQAs nor the energy efficiency as a QA. In [28] the authors use a multi criteria decision making method to analyze the preferences and interactions of QAs based on a fuzzy measure. Their approach is to define whether two QAs interplay in a complementary way or in a redundancy way. Also, in [29], the authors consider the relationships between QAs, but they only evaluate whether or not they affect other QAs positively or negatively. However, none of these approaches [28,29] provide any kind of characterization with the purpose of quantifying the QAs or experimental results to evaluate the impact in energy efficiency and performance. Moreover, there is no formal process to model and generate the best configurations of the QAs, as we propose.

Energy efficiency (or sustainability) analysis is part of our approach, and although the objective of this paper is not to provide a full set of experimental results about energy efficiency such as the example shown in Section 5, the energy information of different recurrent functionalities is important for analysis in our approach. Thus, we can consider

Table 5
Comparative of configurations for the logging FQA.

Variable Context (Message Size in MB)	Framework	Configuration	Energy Consumption (J)	Computational Time (ms)
[0.0001..0.1]	Log4J	(console, txt, false)	2.70	131
		(file, txt, false)	2.81	140
	LogBack	(console, txt, false)	3.44	166
		~~(file, txt, false)~~	~~5.31~~	~~237~~
	java.util.logging	(console, txt, false)	2.60	121
		(file, txt, false)	2.64	134
	Simple Implementation	(console, txt, false)	1.95	108
		(file, txt, false)	**1.99**	**106**
(0.1..10]	Log4J	(console, txt, false)	7.94	617
		(file, txt, false)	8.00	374
	LogBack	(console, txt, false)	8.44	423
		(file, txt, false)	10.34	462
	java.util.logging	(console, txt, false)	7.65	606
		(file, txt, false)	7.79	368
	Simple Implementation	~~(console, txt, false)~~	~~6.78~~	~~641~~
		(file, txt, false)	**6.77**	**328**

Table 6
Comparative of configurations for the encryption FQA.

Variable Context (Message size in MB)	Framework	Configuration	Energy Consumption (J)	Computational Time (ms)
[0.0001..0.1]	javax.crypto	~~(AES, 16, CBC, PKCS5)~~	~~4.18~~	~~170~~
		(DESede, 24, CBC, PKCS5)	4.48	165
		(Blowfish, 16, CBC, PKCS5)	**4.53**	**157**
	Bouncy Castle	**(AES, 16, CBC, PKCS5)**	**4.54**	**157**
		~~(DESede, 24, CBC, PKCS5)~~	~~4.60~~	~~160~~
		(Blowfish, 16, CBC, PKCS5)	**4.54**	**157**
(0.1..10]	javax.crypto	~~(AES, 16, CBC, PKCS5)~~	~~10.18~~	~~478~~
		(DESede, 24, CBC, PKCS5)	10.63	471
		(Blowfish, 16, CBC, PKCS5)	11.45	437
	Bouncy Castle	(AES, 16, CBC, PKCS5)	11.56	446
		(DESede, 24, CBC, PKCS5)	**11.62**	**423**
		(Blowfish, 16, CBC, PKCS5)	11.49	438

several papers that provide the energy experimental information as a repository [8,30–32] in order to use the information gathered in our approach and generate the configuration based on that energy data. The main problem is that none of these repositories provide experimental results on energy efficiency of FQAs, but rather on other functionalities, like, for example, for the Internet of Things (IoT) components [32], or Java collection classes [33].

As in our approach, the relevance of reasoning about energy efficiency and performance at the architectural level is to be able to compare the energy consumption and performance of the different architectural configurations of the same applications (architectural patterns, design variations, frameworks, etc.) [19–21,34]. Some approaches focus on the definition of architectural tactics [2] and design patterns [35] driven by the energy. Other approaches define new architectural description languages (ADLs) that include profiles and analysis of energy consumption and performance [34]. Whatever the case, the experimentation consists of estimating the energy consumption of the application code to analyze the effects of applying a specific architectural pattern or design to the application [2,25,35]. It does not consider the different configurations of the recurrent functionality, their parameters and implementations, which can be reused in many different applications, like the FQAs we consider in this paper.

With respect to the variability modeling and configuration of the FQAs, several approaches try to model and customize them to applications' requirements. Two important approaches are presented in [27] and [4]. In [27], the authors present CORE (Concern-Oriented REuse), where each kind of software characteristic, from base application functionality, including FQAs to non-functional properties, are modeled in reusable units called *concerns*. In [4], a process to model a family of FQAs separately from the base application's functionality is defined, following a Software Product Line (SPL) approach. However, information concerning how FQAs affect energy efficiency and performance is not considered when modeling and customizing FQAs.

Finally, another interesting approach is presented in [36]. The authors approximate the influence of each feature in the feature model on a non-functional property, before generating the configurations. However, they predict the effects of the features instead of giving real measurements (estimations) as we do. Additionally, they model the applications' variability, so they need to build a variability model for each different application, while we focus on specific recurrent functionality (FQAs). Their variability model is always the same because the FQAs can be reused in several applications. Furthermore, they do not consider energy efficiency of applications in their work, which is also a novel and well-known non-functional property nowadays.

10. Conclusions and future work

The process presented in this paper will help software developers to build more energy-efficient software, taking into account other quality attributes such as performance. Our approach allows a richer analysis of the energy consumption and performance of different alternatives for functional quality attributes. Configurations generated, using our approach achieve improvements of up to 63% in energy efficiency and up to 55% in performance.

For future work, we plan to complete our approach in order to consider other non-functional properties such as memory consumption, and levels of security and usability [37], of the different configurations together with the energy efficiency and performance attributes. We also

plan to evaluate the energy consumption and performance of the FQAs deployed in real applications [22,38] to demonstrate the applicability and benefits of our approach.

Acknowledgements

This work is supported by the projects Magic P12-TIC1814 and HADAS TIN2015-64841-R (co-financed by FEDER funds).

References

[1] L. Chen, M.A. Babar, B. Nuseibeh, Characterizing architecturally significant requirements, IEEE Softw. 30 (2) (2013) 38–45, http://dx.doi.org/10.1109/MS.2012.174.
[2] E. Jagroep, J.M. van der Werf, S. Brinkkemper, L. Blom, R. van Vliet, Extending software architecture views with an energy consumption perspective, Computing (2016) 1–21, http://dx.doi.org/10.1007/s00607-016-0502-0.
[3] J. Eckhardt, A. Vogelsang, D.M. Fernández, Are "non-functional" requirements really non-functional?: An investigation of non-functional requirements in practice, 38th International Conference on Software Engineering (ICSE), (2016), pp. 832–842, http://dx.doi.org/10.1145/2884781.2884788.
[4] J.M. Horcas, M. Pinto, L. Fuentes, An automatic process for weaving functional quality attributes using a software product line approach, J. Syst. Softw. 112 (2016) 78–95, http://dx.doi.org/10.1016/j.jss.2015.11.005.
[5] F.D. Rodríguez, S.T. Acuña, N. Juristo, Reusable solutions for implementing usability functionalities, Int. J. Software Eng. Knowl. Eng. 25 (04) (2015) 727–755, http://dx.doi.org/10.1142/S0218194015500084.
[6] K. Pohl, G. Böckle, F.J.v.d. Linden, Software product line engineering: foundations, principles and techniques, (2005).
[7] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, (2012).
[8] D.J. Munoz, M. Pinto, L. Fuentes, HADAS and web services: Eco-efficiency assistant and repository use case evaluation, International Conference in Energy and Sustainability in Small Developing Economies (ES2DE), (2017), pp. 1–6, http://dx.doi.org/10.1109/ES2DE.2017.8015334.
[9] K. Czarnecki, S. Helsen, U.W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Software Process 10 (1) (2005) 7–29, http://dx.doi.org/10.1002/spip.213.
[10] D. Benavides, S. Segura, A. Ruiz-Cortés, Automated analysis of feature models 20 years later: a literature review, Inf. Syst. 35 (6) (2010) 615–636, http://dx.doi.org/10.1016/j.is.2010.01.001.
[11] G.G. Pascual, R.E. Lopez-Herrejon, M. Pinto, L. Fuentes, A. Egyed, Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications, J. Syst. Softw. 103 (2015) 392–411, http://dx.doi.org/10.1016/j.jss.2014.12.041.
[12] E. Tsang, Foundations of Constraint Satisfaction: the Classic Text, BoD–Books on Demand, 2014.
[13] M.W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, K. Deb, On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach, Empir. Softw. Eng. 21 (6) (2016) 2503–2545, http://dx.doi.org/10.1007/s10664-015-9414-4.
[14] J.I. Panach, N.J. Juzgado, F. Valverde, O. Pastor, A framework to identify primitives that represent usability within model-driven development methods, Inf. Softw. Technol. 58 (2015) 338–354, http://dx.doi.org/10.1016/j.infsof.2014.07.002.
[15] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G.K. Olsen, A. Svendsen, Adding standardized variability to domain specific languages, International Software Product Line Conference, SPLC, (2008), pp. 139–148.
[16] I. Reinhartz-Berger, K. Figl, Ø. Haugen, Comprehending feature models expressed in CVL, Model-Driven Engineering Languages and Systems, Springer, 2014, pp. 501–517.
[17] J.M. Horcas, M. Pinto, L. Fuentes, Extending the common variability language (CVL) engine: a practical tool, International Systems and Software Product Line Conference (SPLC), (2017), pp. 32–37, http://dx.doi.org/10.1145/3109729.3109749.
[18] E. Rouillé, B. Combemale, O. Barais, D. Touzet, J.M. Jézéquel, Leveraging CVL to manage variability in software process lines, Asia-Pacific Software Engineering Conference, 1 (2012), pp. 148–157, http://dx.doi.org/10.1109/APSEC.2012.82.
[19] S. Becker, H. Koziolek, R. Reussner, The palladio component model for model-driven performance prediction, J. Syst. Softw. 82 (1) (2009) 3–22, http://dx.doi.org/10.1016/j.jss.2008.03.066. Special Issue: Software Performance - Modeling and Analysis.
[20] A. Martens, H. Koziolek, L. Prechelt, R. Reussner, From monolithic to component-based performance evaluation of software architectures, Emp. Softw. Eng. 16 (5) (2011) 587–622, http://dx.doi.org/10.1007/s10664-010-9142-8.
[21] C. Trubiani, I. Meedeniya, V. Cortellessa, A. Aleti, L. Grunske, Model-based performance analysis of software architectures under uncertainty, International ACM Sigsoft Conference on Quality of Software Architectures (QoSA), (2013), pp. 69–78, http://dx.doi.org/10.1145/2465478.2465487.
[22] Y. Zhuang, The performance cost of software obfuscation for android applications, Comput. Secur. (2017), http://dx.doi.org/10.1016/j.cose.2017.10.004.
[23] K. Grosskop, J. Visser, Energy efficiency optimization of application software, Adv. Comput. 88 (2013) 199–241, http://dx.doi.org/10.1016/B978-0-12-407725-6.00005-8.
[24] G. Kalaitzoglou, M. Bruntink, J. Visser, A practical model for evaluating the energy efficiency of software applications. ICT4S, (2014).
[25] C. Sahin, M. Wan, P. Tornquist, R. McKenna, Z. Pearson, W.G.J. Halfond, J. Clause, How does code obfuscation impact energy usage? J. Softw. 28 (7) (2016) 565–588, http://dx.doi.org/10.1002/smr.1762.
[26] C. Tomazzoli, M. Cristani, E. Karafili, F. Olivieri, Non-monotonic reasoning rules for energy efficiency, J. Ambient. Intell. Smart Environ. 9 (3) (2017) 345–360, http://dx.doi.org/10.3233/AIS-170434.
[27] M. Schöttle, O. Alam, J. Kienzle, G. Mussbacher, On the modularization provided by concern-oriented reuse, Modularity, (2016), pp. 184–189.
[28] A.M. Alashqar, A.A. Elfetouh, H.M. El-Bakry, Analyzing preferences and interactions of software quality attributes using choquet integral approach, International Conference on Informatics and Systems (INFOS), (2016), pp. 298–303, http://dx.doi.org/10.1145/2908446.2908447.
[29] F. Pinciroli, Improving software applications quality by considering the contribution relationship among quality attributes, Procedia Comput. Sci. 83 (2016) 970–975, http://dx.doi.org/10.1016/j.procs.2016.04.194.
[30] K. Djemame, D. Armstrong, R. Kavanagh, A. Juan Ferrer, D. Garcia Perez, D. Antona, J.-C. Deprez, C. Ponsard, D. Ortiz, M. Macías Lloret, et al., Energy efficiency embedded service lifecycle: towards an energy efficient cloud computing architecture, International Conference on ICT for Sustainability, (2014), pp. 1–6.
[31] A. Hindle, A. Wilson, K. Rasmussen, E.J. Barlow, J.C. Campbell, S. Romansky, Greenminer: a hardware based mining software repositories software energy consumption framework, Working Conference on Mining Software Repositories, (2014), pp. 12–21, http://dx.doi.org/10.1145/2597073.2597097.
[32] D. Kim, J.-Y. Choi, J.-E. Hong, Evaluating energy efficiency of internet of things software architecture based on reusable software components, Int. J. Distrib. Sens. Netw. 13 (1) (2017), http://dx.doi.org/10.1177/1550147716682738.
[33] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle, Energy profiles of java collections classes, International Conference on Software Engineering (ICSE), (2016), pp. 225–236, http://dx.doi.org/10.1145/2884781.2884869.
[34] C. Stier, A. Koziolek, H. Groenda, R.H. Reussner, Model-based energy efficiency analysis of software architectures, European Conference on Software Architecture (ECSA), (2015), pp. 221–238.
[35] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, E.Y. Nakagawa, Investigating the effect of design patterns on energy consumption, J. Softw. 29 (2) (2017) e1851, http://dx.doi.org/10.1002/smr.1851.
[36] N. Siegmund, M. Rosenmüller, C. Kästner, P.G. Giarrusso, S. Apel, S.S. Kolesnikov, Scalable prediction of non-functional properties in software product lines: footprint and memory consumption, Inf. Softw. Technol. 55 (3) (2013) 491–507, http://dx.doi.org/10.1016/j.infsof.2012.07.020. Special Issue on Software Reuse and Product Lines.
[37] R.M. Savola, Quality of security metrics and measurements, Comput. Secur. 37 (Supplement C) (2013) 78–90, http://dx.doi.org/10.1016/j.cose.2013.05.002.
[38] I. Ayala, L. Mandow, M. Amor, L. Fuentes, A mobile and interactive multiobjective urban tourist route planning system, J. Ambient Intell. Smart Environ. 9 (1) (2017) 129–144, http://dx.doi.org/10.3233/AIS-160413.