An Approach for Managing Quality Attributes at Runtime using Feature Models

Luis Emiliano Sánchez, J Andrés Diaz-Pace, Alejandro Zunino, Sabine Moisan, Jean-Paul Rigault

An Approach for Managing Quality Attributes at Runtime using Feature Models

Luis Emiliano Sánchez*, J. Andrés Diaz-Pace*, Alejandro Zunino*, Sabine Moisan† and Jean-Paul Rigault†
*ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil, Buenos Aires, Argentina
Email: {emiliano.sanchez,andres.diazpace,alejandro.zunino}@isistan.unicen.edu.ar
†INRIA Sophia Antipolis Méditerrannée, Route des Lucioles 06902, Sophia Antipolis Cedex, France
Email: {sabine.moisan,jean-paul.rigault}@sophia.inria.fr

***Abstract*—Feature modeling has been widely used in domain engineering for the development and configuration of software products. A feature model represents the set of possible configurations to apply in a given context. Recently, this formalism was applied to the runtime (re-)configuration of systems with high variability and context changes, in which the selection of the best candidate configuration is seen as an optimization problem based on quality criteria. To this end, we propose an approach for the specification, measurement and optimization of runtime quality attributes based on feature models, and furthermore, we describe its integration into a component-based architecture for supporting dynamically adaptive systems. A novel aspect of our work is that feature models are annotated with quality-attribute metrics, and then an efficient and flexible algorithm is used to deal with the optimization problem. We report on some examples of adaptation and quality-attribute scenarios in the context of a video surveillance domain, in order to illustrate the pros and cons of our approach.**

***Index Terms*—Feature Models, Runtime Adaptation, Quality Attributes, Optimization, Component-Based Software Engineering, Dynamic Software Product Lines**

I. Introduction

Feature models [3] are a simple but powerful formalism for representing commonalities, varying aspects, and configuration rules of software products, which have been mostly used in the field of Software Product Lines (SPLs). In recent works, feature models have been applied for specifying and executing dynamically adaptive systems. They can be conceptualized as a *dynamic software product line* (DSPL) [8] in which variability and configurations rules are bound and checked at runtime. Similarly to what happens in traditional SPLs, feature models are a convenient formalism for representing a DSPL and enable automated reasoning about properties of interest of the dynamically adaptive system.

In [12] feature models were proposed for the representation and dynamic adaptation of component-based systems, such as a video surveillance (VS) processing chain. The domain of computer vision and video surveillance offers a challenging ground because of the high variability in both the surveillance tasks and the video analysis algorithms. From a functional perspective, the various VS tasks (e.g., counting, intrusion detection, tracking, scenario recognition) have different requirements, namely: observation conditions, objects of interest, and device configuration, among others; which might vary from one application to another. On the implementation side, selecting the (software) components themselves, assembling them, and tuning their parameters to comply with context might lead to different configuration variants. Moreover, the context is not fixed, but rather it evolves dynamically and requires runtime adaptation of the component assembly.

In a given execution context many configurations are valid but only one of them should be selected for system adaptation. The selection process must consider configuration rules, resource restrictions and stakeholders' preferences, especially with regard to non-functional properties or *quality attributes* of the system. Thus, the selection of the "best" system configuration implies to find the candidate that optimizes a given set of quality attributes quantified by means of *quality metrics*. This generally involves trade-offs between several aspects, such as: maximizing accuracy, achieving the best performance, or choosing the simplest setup or substitution for the current configuration, among others.

In previous work [16], we presented a heuristic search algorithm called CSA (Configuration Selection Algorithm) for solving the optimization problem resulting from the runtime configuration of a system based on feature models. This algorithm offers different variability points for leveraging between execution efficiency and optimality, and allows defining different objective functions for comparing configurations and optimizing multiple attributes simultaneously, while adhering to resource restrictions and feature model constraints. However, algorithms such as CSA necessarily require an infrastructure designed with capabilities for monitoring context changes and activating (at runtime) components that implement specific features.

In this article, we present the overall approach and component-based architecture in which the CSA is embedded. Our approach provides a runtime framework for the specification, measurement and optimization of quality-attribute properties expressed on top of feature models. We show how quality attributes can be specified by means of feature attributes and evaluated with quality metrics in the context of feature models. Furthermore, we discuss the selection process carried out by our optimization algorithm, highlighting several trade-off situations between quality attributes.

The rest of this paper is organized as follows. Section II gives background information about feature models, and their role in the representation and dynamic adaptation of

component-based systems. Section III describes our model-based approach for managing quality attributes on feature models, including details of the available metrics and the variants of the optimization algorithm. Section IV provides a representative example of runtime adaptation and quality-attribute scenarios using a video surveillance processing chain as case-study. Section V discusses related work. Finally, Section VI presents the conclusions and outlines future research.

II. Feature Models for Runtime Adaptation

A feature model is a compact representation of all possible products or configurations, for instance, of a software product line. These models are visually represented as features and relationships among them. Features correspond to selectable concepts of the system at any abstraction level: functional and non-functional (or quality-attribute) requirements, environment and context restrictions, runtime components, implementation modules, etc. A feature model is arranged in a hierarchy that forms a tree where features are connected by:

- *Tree constraints*: relationships between a parent feature and its child features (or sub-features). Tree constraints include *mandatory*, *optional*, *xor* (alternative) and *or* relationships between parents and sub-features.
- *Cross–tree constraints*: typically *inclusion* or *exclusion* statements of the form "if feature F is selected, then features A and B must also be selected (or deselected)".

The root of the tree represents the concept being described, generally the system itself, and the remaining nodes denote features and their sub-features.

The above description corresponds to the basic notations for feature model languages. Many language extensions have been proposed, including cardinality-based relationships, generic propositional formulas for cross-tree constraints, and *extended* or *attributed feature models* [9]. The latter is a type of model in which additional information is added as *feature attributes*. An attribute consists of a name, a domain, and a value. Attributes are often used to specify extra information, such as cost, response time, or memory required to support the feature, among others.

A. Example: Video Surveillance System Model

Figure 1 depicts a simplified version of the feature model for a video surveillance processing chain, as described in [12]. The model is defined by the aggregation of two sub-models: $VSspecification$ and $VScomponent$. The first model represents "what to do" and includes: environmental and hardware conditions ($Context$ feature), functionality ($Application$, $ObjectOfInterest$ features) and quality parameters (QoS feature) desired by users. The second model represents the system components and their parameters, that is "how (the software) should do it". Note that cross-tree constraints are used to formalize extra-feature dependencies. For instance, in the $VScomponent$ sub-model, these constraints define configuration rules for the correct assembly of software components. Furthermore, cross-tree constraints between both sub-models allow to define *event-condition-action* (ECA) rules. Basically, system events are translated to the (de)selection of features on the specification side, conditions are defined by constraint logic, and actions are carried out by constraint propagation, which triggers the selection/deselection of features on the component side.

A system configuration $\mathbb{C}$ of a video-surveillance processing chain can be defined as a set of running components, each one customizable with a set of parameters, and these components can be removed, added, replaced, and tuned dynamically. In our approach, a component is a unit of independent deployment that requires or provides services to other system components through specific interfaces. A given component can be replaced by another one, either at design time or runtime, as long as the new component meets the requirements of the original component (expressed via its interfaces).

Figure 2a shows a configuration instance of the video surveillance processing chain. The purpose of this system is to analyze image sequences to detect interesting situations or events. Its global architecture is a five-stage pipeline of software components. The pipeline starts with image acquisition, then segmentation of the acquired images to group image regions into blobs, classification of possible objects, tracking of these objects from one frame to the other, and lastly scenario recognition for intrusion detection (or other real-time events). The output results might be stored for future processing, or mighty raise alerts to human observers. In the domain of computer vision and video surveillance systems, the pipeline can involve additional steps (e.g., clustering, shadow removal, and data fusion - in case of multiple cameras), which require the deployment of different software components. These components can have variants along different dimensions (e.g. algorithms, strategies, input data, etc), each one corresponding to a different parameter.

B. Mapping Runtime Space to Model Space

The mapping between the runtime system and its model representation is achieved by means of *feature model configurations*. Formally, a system configuration $\mathbb{C}$ is represented by a feature model *full configuration* defined as a 2-tuple $\langle S, D \rangle$ where S and D are sets of selected and deselected features respectively, such that $S \cap D = \emptyset$ and $S \cup D = F$ (set of all features). Figure 2b illustrates a full configuration example that fulfills cross-tree constraints. Each running component in Figure 2a is associated with a selected feature in the full configuration.

Features are classified into *concrete* and *abstract* ones depending on whether they represent software elements of the system, i.e. deployable components and their configuration parameters, or not. Concrete features reference software elements in a one-to-one mapping. Examples of concrete features are $ImageAcquisition$, $Resolution.High$, etc. By contrast, the remainder features are called *abstract* and they usually correspond to high-level features used for organizing the whole diagram (e.g. $VSsystem$), grouping sets of com-

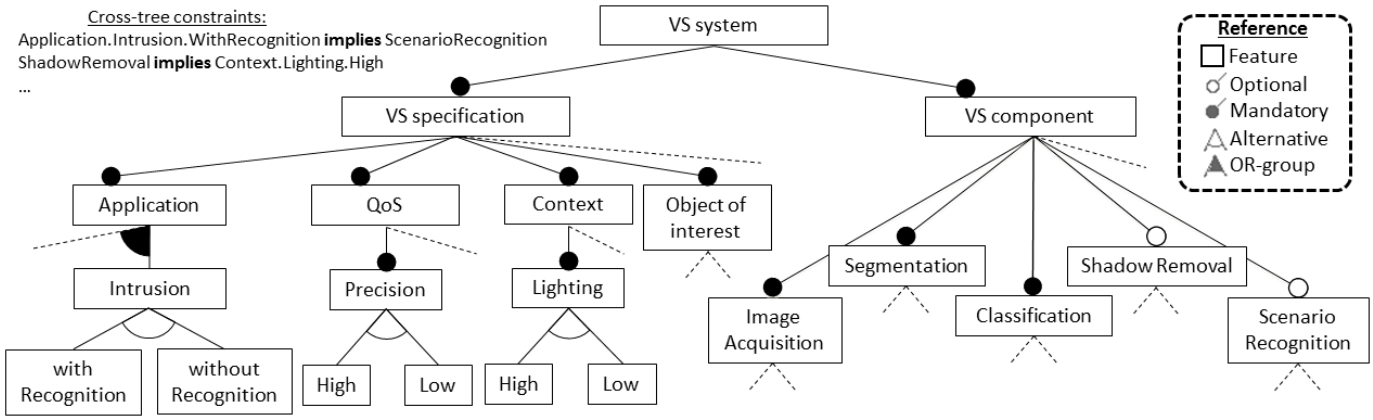
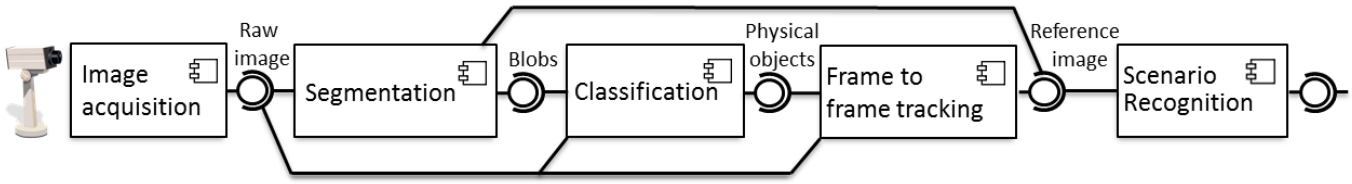
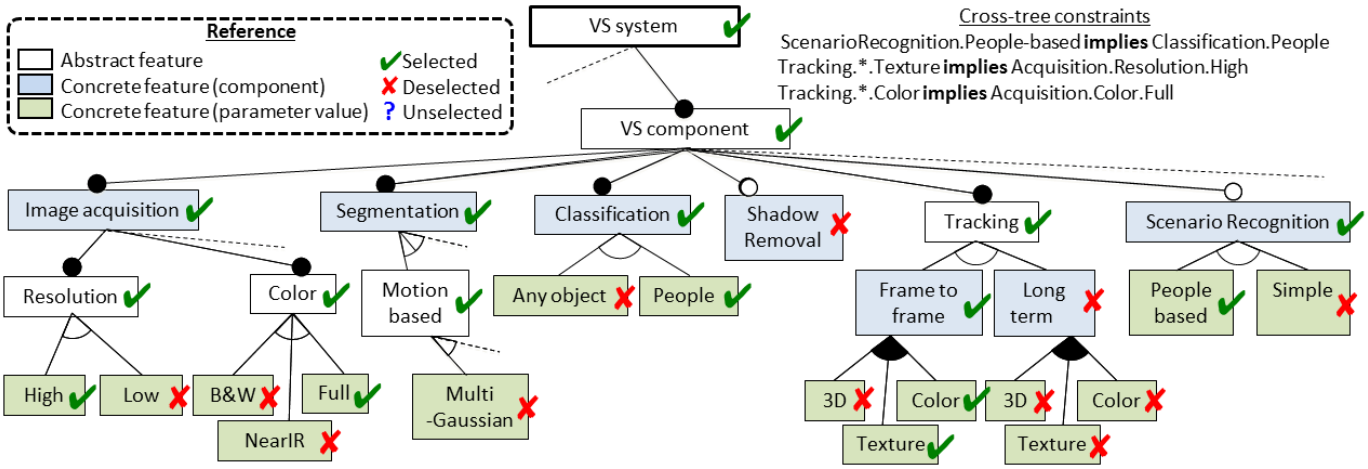


Figure 1. Feature model for a video surveillance system



(a) System configuration of a video surveillance processing chain



(b) Feature model full configuration of a video surveillance processing chain

Figure 2. Mapping system configuration to feature model configuration

ponent and parameter variants (e.g. $Tracking$, $Resolution$), and representing specification and context aspects (features in $VSspecification$ sub-model).

Context changes or user interactions are events that can trigger dynamic reconfigurations of the model (selecting and deselecting features). For instance, lighting changes can have an impact on the parametrization (i.e. configuration) of the acquisition and segmentation components of the processing chain. Users may require to recognize different events or perform a different task, tuning or even replacing the scenario recognition component for another task-dependent component. As another example, energy supply conditions can imply a system reconfiguration.

These events seldom result in a full system configuration but in a *partial configuration* of the feature model instead. A partial configuration is a partial assignment of feature values that represents the set of valid full configurations compatible with an execution context. It is defined as a 3-tuple $\langle S, D, U \rangle$ where U is the set of *unselected* (i.e., unassigned) features, such that S, D, and U are pairwise disjoint and $S \cup D \cup U = F$. A key challenge is to derive an "optimal" full configuration from a given partial configuration. This process consists in selecting or deselecting unselected features until U becomes empty considering the satisfaction of logical

constraints and resource restrictions, and the optimization of multiple objectives based on quality-attribute properties. The optimal decision is usually made in the presence of trade-offs between two or more conflicting objectives. For example, selecting a new configuration that maximizes system performance while at the same time minimizes the required time for reconfiguring it. This combinatorial optimization problem and the corresponding Configuration Selection Algorithm (CSA) are described in Section III-C.

C. Software Architecture

In previous sections we introduced feature models for describing full and partial configurations of systems, and how these configurations can be adapted to a different execution contexts. In this section we describe the component-based architecture that enables such a dynamic adaptation. This architecture was designed to accomplish the following goals: (i) support the specification, measurement, and optimization of quality-attribute properties on feature models; (ii) articulate the running system with a runtime (feature-based) model of that system; and (iii) provide hooks for configuration selection algorithms (such as CSA [16]).

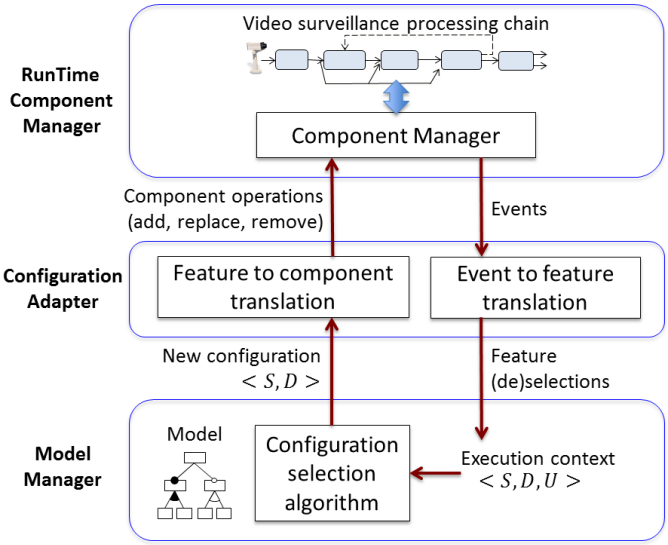


Figure 3. Runtime Adaptation Architecture

The architecture defines three main collaborating components, as shown in Figure 3:

- *Runtime Component Manager* (CM): this framework deals with the low-level aspects of software components and configuration changes. It captures basic events about context changes (e.g., lighting changes) and user interactions (e.g. preference for high resolution), and then forwards those events to the Configuration Adapter, which returns a set of component operations for adapting the current configuration to the new execution context. The CM is responsible for applying these operations, that is to tune, add, remove, or replace software components, hence changing the system configuration.
- *Configuration Adapter* (CA): it is a mediator module between the CM and the Model Manager. It receives events from the CM and interprets them as feature actions (selection and deselection of features) for the Model Manager. In return, it obtains a new feature model full configuration that is compatible with the new execution context. Since the CA manages the mapping between features and software elements, it is responsible for instructing the CM to reconfigure the system.
- *Model Manager* (MM): it holds a representation of the running system (e.g., a model of the video surveillance system). Besides features and their constraints, this model includes feature attributes, resource restrictions, an objective function to be optimized, and the full configuration that represents the current system configuration. A key part of the MM is the *Configuration Selection Algorithm* (CSA), which is in charge of selecting a new full configuration from a given partial configuration. This algorithm enforces configuration validity and resource restrictions, and makes use of the objective function to guide the selection (e.g., minimizing the number of component changes in the processing chain, maximizing the detection accuracy, or any linear combination of them).

A typical execution scenario for an event-driven reconfiguration is shown in Figure 4. The main loop corresponds to the video surveillance processing chain. Its execution environment is provided and controlled by the CM. When an event occurs, the CM handles it by sending an asynchronous message to the CA. Note that the system continues with its normal operation, while next configuration is computed in background by the MM .

Based on predefined event rules, the CA informs the MM about a subset of selected/deselected features for the new execution context. For example, a light dimming event implies the selection of feature $Context.Lighting.Low$. The MM creates a partial configuration based on those features, and also adjusts feature attributes for the selection step. As we will explain later, some feature attributes have predefined values while others change depending on the current system configuration. Next, the selection step is performed by the CSA that takes as inputs the partial configuration as well as extra information about feature constraints, attributes, resource restrictions, and the objective function to minimize. When a new full configuration is computed, the CA compares the new and current configurations in order to identify re-configuration operations (e.g., addition, removal, replacement, and parameter tuning of software components). Finally these operations are executed by the CM, and the CA returns to an idle state.

III. Quality Attributes on Feature Models

A quality attribute is a key aspect (or property) of the system that is used by its stakeholders to judge its operation, rather than specific (functional) behaviors [2]. Quality-attribute properties of a system are typically quantified by quality metrics, particularly those with runtime characteristics. Systems often fail to meet stakeholders needs regarding these attributes

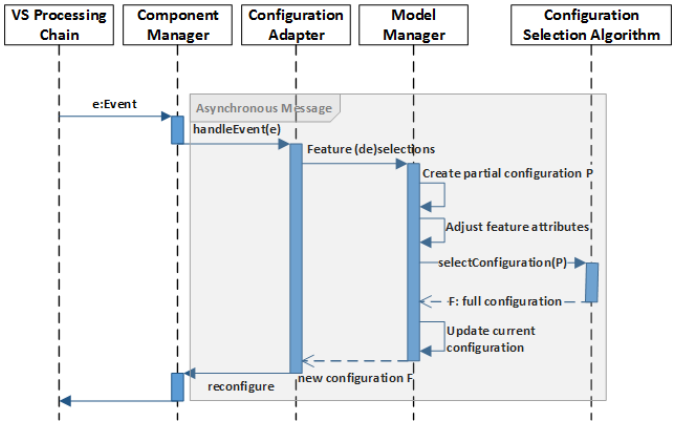


Figure 4. Reconfiguration due to a system event

when they focus on some aspects without considering the impact on others. For instance, when system adaptation is required, it might not be possible to select a configuration that minimizes the reconfiguration time while at the same time maximizes the overall quality of service (QoS) since for this we probably need to setup and tune additional components that impact negatively on the reconfiguration time. Furthermore, the overall QoS degree is defined based on a combination of conflicting runtime properties (e.g., response time, accuracy, availability, security). For example, replicating communication and computation to achieve availability, or including a shadow removal component to achieve high accuracy for event recognition, might conflict with performance requirements (e.g., low response time) or resource restrictions (e.g., maximum memory consumption). Stakeholders generally find it difficult to quantify their preferences in such conflict situations.

The goal of our model-based approach for managing quality attributes is to quantitatively evaluate and trade-off multiple quality attributes to arrive at a better overall system configuration. We do not look for a single metric but rather for a quantification of individual attributes and for trade-offs among these different metrics, formalizing the problem as the optimization of an objective function that aggregates these metrics and quantify stakeholders' preferences for individual attributes.

In our approach, the management of runtime quality attributes involves three steps, namely: (i) specification, (ii) measurement, and (iii) optimization. Specification deals with the representation and assignment of quality-attribute properties of individual system elements to features. Measurement implies the design of metrics for feature model to assess these quality attributes at the system level. At last, optimization deals with the maximization (and/or minimization) of conflicting attributes, which are assigned to different weights in order to consider stakeholders' preferences, while still meeting configuration rules and resource restrictions.

A. Specification

Quality-attribute properties must be specified at design time for system architects. A wide range of properties exists to evaluate the runtime operations of a system. Some attributes are common to most adaptive systems, like reconfiguration time, response time, memory consumption, availability, among others. Besides, video-surveillance systems exhibit specific attributes, namely: accuracy and sensitivity of detection or tracking algorithms, relevance of object classification, frame rate, among others.

These properties can be categorized into the following classes, depending on how they are specified on the feature models:

1) *Direct assigned attributes*: this class contains attributes that are representable as features, because they can be directly selected by stakeholders during the product configuration phase at development time. At runtime, the selection and deselection of these features can be triggered by events coming from context changes. In our model of Figure 1, these features correspond to the Quality of Service (QoS) branch.
2) *Quantitative attributes*: this category contains feature attributes that can be measured on a metric scale, such as: response time, reconfiguration time, accuracy, among others. These attributes define properties of individual features, but one can infer a measure for the overall configuration using some metric function able to aggregate the values of individual elements. For example, the system memory consumption can be calculated as a sum of the required memory for each running component.
3) *Qualitative attributes*: this category includes attributes of features that can only be described qualitatively using an ordinal scale, i.e. a set of qualifier tags like *low*, *medium*, and *high* for usability, security, or camera resolution, among others. In this case, there is no metrics for deriving quantifiable measures of the overall configuration. However, a mapping function from qualifier tags onto real values can be used in order to handle these attributes as quantitative properties.

B. Measurement

For each attribute, the overall value of a configuration is calculated by an *aggregate function* that consider the value of the selected features (for some particular attributes, deselected features are also considered). An aggregate function is a function that performs a computation on a set of values to return a single value. Different aggregate functions are suggested as *quality metrics,* according to the nature of the attribute [15]. We have identified 4 functions (described in Table I) that have mathematical properties suitable for optimizing using feature models.

To compute the aggregate functions, all features of the feature model are enriched with two slots, a_S and a_D, for each attribute a. These slots represent the contribution of the feature to the aggregated value when its state is selected or deselected). These slots are initialized by default to the

Table I
AGGREGATE FUNCTIONS FOR A GIVEN QUALITY ATTRIBUTE a AND CONFIGURATION $\mathbb{C} = \langle S, D \rangle$

Function	Formulation	Quality Attribute Examples
Addition	$M_a^{+}(\mathbb{C}) = \sum_{f \in S} a_S(f) + \sum_{f \in D} a_D(f)$	required memory, reconfiguration and response time (sequential execution),
Product	$M_a^{\times}(\mathbb{C}) = \prod_{f \in S} a_S(f) \times \prod_{f \in D} a_D(f)$	accuracy, availability
Maximum	$M_a^{M}(\mathbb{C}) = \max(\max_{f \in S}(a_S(f)), \max_{f \in D}(a_D(f)))$	reconfiguration and response time (parallel execution)
Minimum	$M_a^{m}(\mathbb{C}) = \min(\min_{f \in S}(a_S(f)), \min_{f \in D}(a_D(f)))$	security, usability (using a metric scale for qualifier tags)

neutral element (e) of their specific function: 0 for addition,1 for product,∞ for minimum, and$-\infty$ for maximum. Neutral elements do not affect aggregate values, so they are used as "null" values for feature slots where attributes do not apply. Concrete features might have predetermined values for some attributes that correspond to inherent properties of software elements (components and parameters), such as required memory, start up time, failure probability, accuracy, etc. Although our approach permits to change them dynamically, these property values are generally considered constant across the system execution. These values are usually predetermined by system experts, for instance, by measuring the performance of each component in isolation.

Note that a feature f contributes differently to the aggregate value when it is selected ($a_S(f)$) or deselected ($a_D(f)$). For most quality-attribute properties (e.g. memory consumption or response time), deselected features do not contribute at all ($a_D(f) = e$), but for other attributes some of them do. For instance, for minimizing the reconfiguration time, if a feature representing a software component is selected for the next execution context, the corresponding component *start up time* is taken into account, whereas when the feature is deselected its *shut down time* is considered instead.

The ranking of a given configuration is a combination of its aggregated values. Along this line, we define an optimal system configuration $\mathbb{C}$ as the one that minimizes the following weighted function:

$$L(\mathbb{C}) = \sum_{a \in A} w_a \times \frac{M_a(\mathbb{C}) - \mu_a}{\sigma_a} \qquad (1)$$

where A is the set of quality attribute properties of interest, w_a is the weight of each quality attribute a, M_a is the aggregate function associated with attribute a, that might have different forms according to the nature of the attributes (see Table I), and μ_a and σ_a are the average value and the standard deviation of M_a for all valid configurations. The expression $(M_a(\mathbb{C}) - \mu_a)/\sigma_a$ is required to *normalize* M_a, since each attribute has different measuring units (e.g., milliseconds for response time, megabytes for memory consumption, etc) and orders of magnitude. The computation of μ_a and $\sigma_a$is done automatically at design time, while w_a must be set manually considering $\sum_{a \in A} |w_a| = 1$.

The linear combination of these metrics in a single objective function, such as Equation (1), allows us to deal simultaneously with several runtime attributes. That is, we transform a multi-objective optimization problem into a mono-objective problem by means of a linear scalarization technique known as *weighted sum method* [10]. The parameters of the scalarization are the weights of each term, and they provide a simple way for specifying stakeholder preferences for attributes. By convention, the optimization problem is stated in terms of minimization, but each individual term can be maximized or minimized if the associated weight is negative or positive respectively.

An example of an extended feature model that maps property values from components to features slots is shown in Figure 5. Three attributes are depicted: reconfiguration time ($rtime$), memory consumption ($memory$), and accuracy (acc). We compute system reconfiguration time with an additive metric since the Component Manager applies configuration operations sequentially. Thus, the total time is the sum of the *start up time* and *shutdown time* of added and removed components respectively. For instance, if $ShadowRemoval$ (SR) component is currently running and a new reconfiguration is required, $rtime_S(SR) = 0\,sec$ since selecting this feature does not have any impact on the reconfiguration time because the component is already in execution. In turn, $rtime_D(SR) = ShutdownTime(SR)$ because deselecting this feature implies the removal of the component. In the same way, if the component is not running in the current configuration and a new reconfiguration is required, $rtime_D(SR) = 0\,sec$ and $rtime_S(SR) = StartupTime(SR)$.

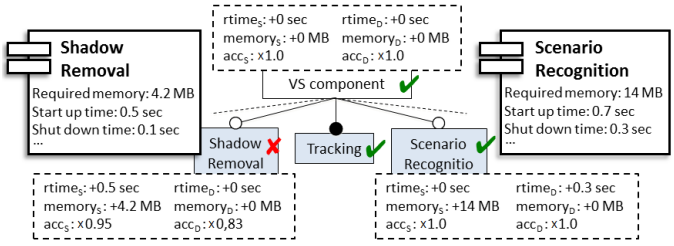


Figure 5. Extended feature model example

Besides reconfiguration time, another interesting performance measure is response time. It is defined as the time required for the system to process a request or task. It can be measured with an additive or maximum function depending on the execution context of components: if several tasks are executed in the same thread, the overall required time is the addition of the required times per task; otherwise if each task is executed in parallel, this time is the maximum among the required times per task. In our video surveillance pipeline example, an equivalent performance measure is the frame

period, i.e. the time required to process an image frame. It is computed by a maximum function since each step in the pipeline performs in parallel.

Regarding availability, we define it as the time ratio when the system is in a functioning condition or, mathematically, as the probability to operate satisfactorily (expressed as 1 minus the failure probability). Then the system availability can be measured with a product function because each component may have an operational probability, and the overall value is the product of these probabilities. In the same way, due to the pipeline architecture of the video surveillance system, the accuracy of a required task (e.g., intrusion detection) can be computed as the product of the accuracy of the involved components in each step. So, if one fails (due to noise during image acquisition, wrong segmentation, or false negative or false positive during scenario recognition) the rest of the chain fails.

Some properties may be applicable for some components but not for others. For instance, tracking or detection algorithms can be measured in terms of sensitivity, but it is not the case for image acquisition. The same happens with security, usability, and other quality attributes. For example, let us assume that our video surveillance system is running on an online environment, connected to the Internet, to be acceded remotely. We can measure the overall security of the system as the minimum value of the involved components (firewall, authentication module, etc) because the overall system vulnerability depends on the most vulnerable element. The security degree can be specified as a qualitative attribute, using qualifier tags like *low*, *medium*, and *high security*, and then mapped to numeric values. If an attribute does not apply to a component or a parameter we set its a_S value to the neutral element as we do for abstract features.

With the above examples we showed a variety of runtime properties and metrics for measuring them. The linear weighed function is fundamental for grouping these measures with different weights to evaluate the overall quality of system configuration candidates.

C. Optimization

The optimization problem takes a partial configuration of an extended feature model and a so-called objective function as inputs and returns the full configuration fulfilling the criteria established by the function. Selecting the configuration that minimize (or maximize) the given function is an intractable combinatorial optimization problem since the set of valid configurations increases exponentially with respect to the number of optional features.

In real-time systems that have to adapt themselves in bounded periods of time, any configuration selection algorithm must meet correctness, completeness, and efficiency requirements, preferably with a high degree of optimality. Algorithm correctness is fundamental since it is impracticable to deploy an invalid configuration, i.e. a configuration that does not fulfill feature constraints and resource restrictions. Completeness and time efficiency are required under time constraints. Finally, although an optimal solution is not mandatory, it is desirable to compute good-enough solutions. The proposed Configuration Selection Algorithm (CSA) [16] accomplishes these requirements.

CSA is based on a Best-First Search schema [14] that performs a systematic search over an abstract structure called *state-space graph*. In our case, this structure is a binary tree where nodes are valid states of the problem (partial and full configurations) and edges represent selection/deselection of features. From a given initial partial configuration, that represents the root of the tree-like state-space graph, the algorithm generates new nodes by selecting and deselecting features. It uses an heuristic function to estimate the objective function value of these nodes in order to drive the search towards the optimal solution, and a container ($OPEN$ set) for storing and ordering the visited nodes. The algorithm succeeds when it reaches a full configuration (goal node).

The algorithm is enriched with constraint propagation techniques (over feature constraints) that reduce the search space considerably and discard invalid configurations. In addition, the algorithm was extended to validate resource restrictions (global constraints). Resource restrictions are represented as inequality constraints using an aggregate function from Table I. For example, a memory consumption restriction has the form $M_a^+(\mathbb{C}) \leq \alpha$, α being the memory limit. If one of these restrictions is violated, the configuration is considered invalid and discarded. Inequality constraints with aggregate functions can be used to enforce other resource restrictions, like CPU load, bandwidth use, or maximum number of running components (if the Component Manager includes this limitation).

The $OPEN$ set of visited nodes defines different *search strategies* depending on its implementation structure (e.g., a stack, queue, priority queue). Some well-known search strategies includes *Depth-First Search* (DFS), *Breadth-First Search* (BFS), *Best-First Search Star* (*BF**), and *Greedy Best-First Search* (*GBFS*). The last two are *informed search strategies* that require the heuristic function to guide the search to the optimal (or sub-optimal) solution. BF* is implemented by means of a priority stack where nodes are ordered according to their heuristic value, while GBFS is implemented with a stack where each pair of successors are added in the order given by the heuristic function, performing a backtracking search.

For efficiency, the GBFS strategy appears as the ideal option for real-time systems that have to adapt in bound time. This strategy ensures polynomial time complexity for feature models without cross-tree constraints and guarantees over 90% optimality. Regarding optimality, BF* strategy is ideal for assisting design decisions, such as product configuration and generation of software product lines from feature models since it guarantees the optimal solution using *admissible heuristics* [14], although it takes exponential time to compute.

Details of the search strategies and heuristics are provided in [16], along with some properties and experimental results using randomly generated scenarios regarding efficiency and optimality of the algorithm variants.

IV. Case-Study: Adapting a Video Surveillance System

To illustrate our approach, we present a simple scenario of runtime adaptation. In this example, the users' goal is to execute the VS system for detecting intrusion with people recognition under various illumination conditions. For simplicity, we consider only two optimization criteria: reconfiguration time ($rtime$), which must be minimized; and accuracy for intrusion detection (acc), which must be maximized. Then, the objective function (to be minimized) is defined by $L(\mathbb{C}) = w_{rtime} \times (M^{+}_{rtime}(\mathbb{C}) - \mu_{rtime})/\sigma_{rtime} + w_{acc} \times (M^{\times}_{acc}(\mathbb{C}) - \mu_{acc})/\sigma_{acc}$, where $w_{rtime} > 0$ for minimization and $w_{acc} < 0$ for maximization. Note that $M^{\times}_{acc}(\mathbb{C})$ and $M^{+}_{rtime}(\mathbb{C})$ are normalized using $\mu_{rtime} = 1.42\,sec$, $\sigma_{rtime} = 0.65\,sec$, $\mu_{acc} = 0.6$ and $\sigma_{acc} = 0.11$.

For system startup, let us assume that the scene is under normal light conditions. Since the system is not yet in operation, the initial full configuration stored by the Model Manager (MM) has an empty set of selected features and the reconfiguration time is equivalent to the system startup time. According to the users' goal, the Configuration Adapter (CA) sends to the MM the features $Application.Intrusion.WithRecognition$ and $Application.ObjectOfInterest.People$ to be selected. The initial, partial configuration computed by the MM is partially depicted in Figure 6. Remember that a full system configuration should be derived from the partial configuration. The selected features lead, via constraint propagation, to the selection of the features $ScenarioRecognition.PeopleBased$ and $Classification.People$, in order to achieve the goals. The rest of the system settings still remain undefined, providing a set of 72 possible full configurations for the given execution context. This set of configurations are shown in a two-dimensional space in Figure 7, with accuracy ($M^{\times}_{acc}(\mathbb{C})$) and startup time ($M^{+}_{rtime}(\mathbb{C})$) as their coordinates, and level curves that indicate the direction in which $L(\mathbb{C})$ decrease.

The CSA is in charge of choosing one of the available (full) configurations. Let us suppose that we want to select the most accurate configuration, no matters its required startup time (as the system is still offline anyway), so we set $w_{rtime} \approx 0$ for the objective function. This is reflected in level curves that are nearly horizontal. As it can be seen in Figure 7, the solution returned by the algorithm is the one that maximizes the intrusion detection accuracy and, consequently, the startup time since the most accurate components and parameters require more time to be in operation. This solution includes the parameters $Resolution.High$ and $Color.Full$ for $ImageAcquisition$, and the components $ShadowRemoval$ and $Tracking.LongTerm$ with its three parameters for considering 3D information, image texture and color, in order to improve intrusion detection performance.

Let us then assume that, at some time, ambient light is drastically dimmed, so the system has to adapt to this lighting reduction. The corresponding "light dimming" event is triggered by the processing chain during the image analysis (segmentation step). This event propagates from the Component Manager to the CA. Event rules in CA consider $Application.Intrusion.WithRecognition$ and $Context.Lighting.Low$ as selected features to achieve intrusion detection with the current lighting condition. Taking into account cross-tree constraints, the MM infers the (new) partial configuration depicted in Figure 8. Note that this is a more restrictive scenario. The $ScenarioRecognition$ component is not longer able to precisely recognize people, probably leading to more false positives during detection. However, this is the best that the VS system can do with poor lighting conditions.

Since the system is executing, we consider reconfiguration time as a priority over accuracy for the next adaptations. Along this line, we set $|w_{rtime}|$ greater than $|w_{acc}|$ in the objective function. Figure 9 depicts the set of solutions (8 possible full configurations) derived from the partial configuration in Figure 8. Here, CSA returns the configuration that minimizes added, removed and tuned components in order to reduce reconfiguration time. The chosen solution keeps the $Tracking.LongTerm$ component, together with $3D$ and $Texture$ parameters, but removes $ShadowRemoval$ since it is not compatible with the new lighting context and tunes $ImageAcquisition$, $Classification$ and $ScenarioRecognition$ parameters.

For this small-size example with 26 features, a few cross-tree constraints and 2 optimization criteria, both CSA variants (BF* and GBFS) behave similarly: they get the optimal solution in both scenarios with an execution time around 0.5-0.7 ms. For more details about the algorithm performance in larger instances of the problem, please refer to [16].

V. Related Work

We can organize the related work into three categories, namely: (i) use of models and QoS criteria at runtime for adaptive systems; (ii) management and variability of non-functional properties on feature models; (iii) and algorithms for feature model optimization.

In the first category, several approaches have proposed the use of models and QoS criteria at runtime for specifying and executing adaptive systems, such as architecture-based self-adaptation [6][5] and DSPL [13]. In [6], Garlan et al. propose an approach that uses the software architecture of a system as a model for dynamic adaptation. This approach provides a framework that supports mechanisms for self-adaptation and allows adaptation expertise to be specified and reasoned about. Similarly, in [5], Silva et al. address the dynamic selection of architecture configurations based on QoS criteria but they describe the system architecture with *Arquitecture Description Languages* (ADL) instead of architectural models like [6]. In [13], Morin et al. present an approach for managing DSPL at runtime, which combines model-driven and aspect-oriented techniques. Besides feature models for DSLPs, this approach uses additional models for representing system context, architecture and reasoning (i.e., configuration selection), while we instead appeal to a unified model for representing context, architecture and reasoning.

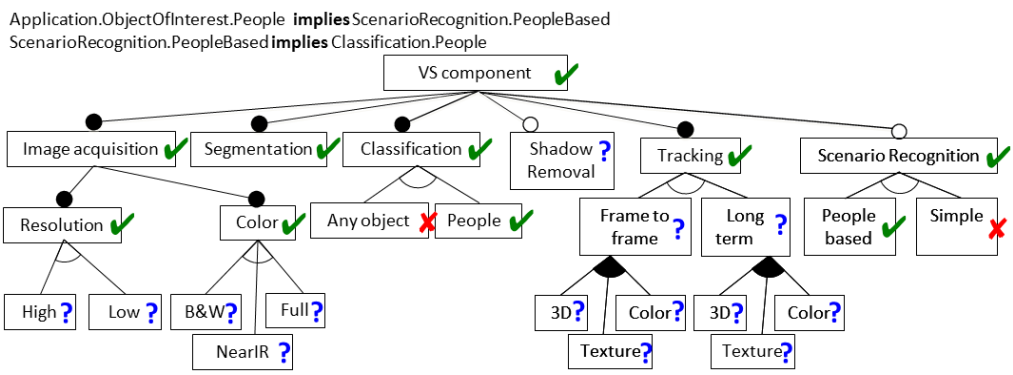


Figure 6. Initial partial configuration for intrusion detection with people recognition

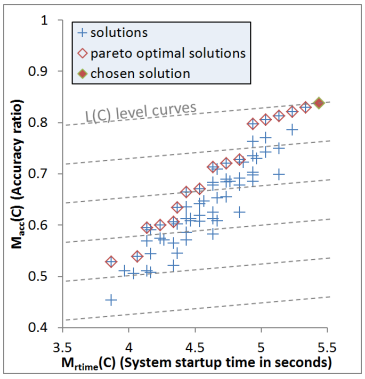


Figure 7. Solution set derived from initial partial configuration

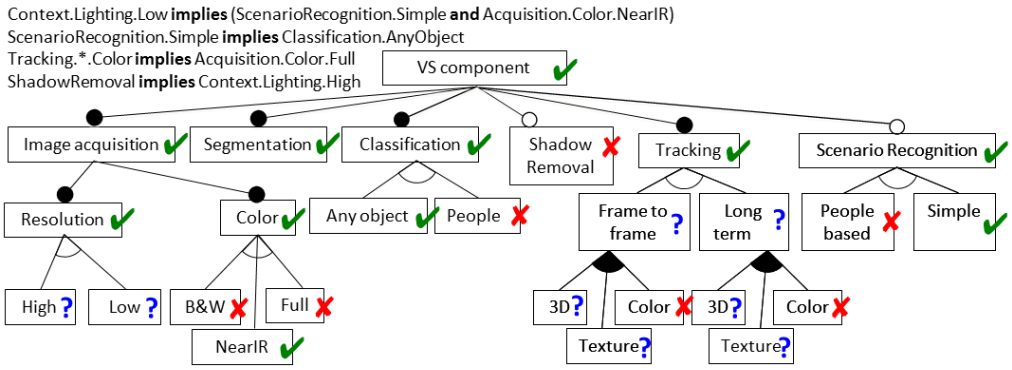


Figure 8. New partial configuration for intrusion detection with light dimming

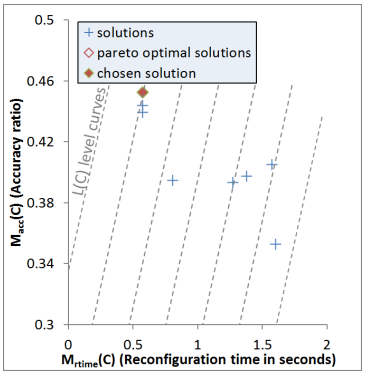


Figure 9. Solution set derived from new partial configuration

Besides system and context representation, these works differ mainly in the reasoning process for configuration selection: [6] and [5] rely on *utility theory* to decide among multiple potential adaptation alternatives considering business objectives and priorities with regard to runtime attributes; whereas the reasoning framework in [13] is based on a goal-based model but no optimization strategy is explicitly described. Unlike them, we formalize and resolve this issue as a feature model optimization problem.

In the second category about management and variability of non-functional properties, different works have addressed this issue by means of software product lines conceptualized as feature models [1][18][19]. These works provide several techniques and tools for specifying properties as feature attributes, measuring them for partial and full configurations, dealing with trade-offs, and assisting the user in the product configuration process. For instance, Bartholdt et al. [1] propose a tool for dealing with trade-offs and measuring non-functional properties in the configuration process of SPLs, based on aggregate functions computed over feature attributes. According to [19], non-functional properties can be specified following any of the three classes mentioned in Section III-A. Siegmund et al. [18] propose a similar categorization to select an appropriate measurement technique for these properties, and provide an optimization process based on CSP solvers. The main difference between these works and our proposal is that they consider non-functional properties on feature models for design decisions, while we address this management for runtime adaptation.

Regarding the third category, algorithms for solving the feature optimization problem can be roughly classified into two categories: exact and approximate algorithms. The former includes traditional CSP solvers [4] and planning techniques [20] that perform a systematic search over the solution space of partial/full configurations, ensuring optimality. Since a systematic search implies an exponential time complexity, approximate algorithms are preferred for large problem instances when an approximate solution is acceptable. For instance, Guo et al. [7] propose a genetic algorithm for optimized configuration selection. In [17], the authors have compared several evolutionary algorithms for solving the multi-objective optimization problem on feature models and computing the *Pareto front* of solutions. Lastly, in [21], the authors refactor a feature model so that the optimization problem becomes a *Multi-Dimensional Multi-Choice Knapsack problem* and both exact and approximate methods can be applied.

Our algorithm (CSA) provides two main variants: one for computing exact solutions, and another one for approximate solutions with a more efficient use of computational time and memory. Thus, although we use the latter at runtime where

computational efficiency is mandatory, the former can be applied at design time where optimality is desirable. The use of admissible heuristics on feature models ensures exact solutions and improves algorithm performance, but limits quality metrics to the variants depicted in Table I, while other approaches, like CSP solvers and genetic algorithms, are more expressive in that regard.

VI. Conclusions and Future Work

The primary contribution of this article is an approach for improving the dynamic adaptation of component-based systems that addresses quality attributes at runtime by means of feature models. It provides a simple but still expressive framework to specify, quantitatively evaluate and trade-off multiple quality attributes to arrive at a better system configuration.

The approach is integrated into a component-based architecture that provides mechanisms for event handling and self-adaptation. It can be tailored to different classes of component-based systems, but it requires an extra engineering effort at design time for mapping system components, context and events to feature models, as well as for specifying attribute values, metrics and weights for the optimization step. Along this line, we expect to extend existing tool support for system architects.

The optimization step is supported by a heuristic search algorithm that ensures correctness and completeness while addresses time efficiency and scalability for large scale instances of the problem. Even optimality can be achieved at expense of a lower performance. However, a perceived limitation is that the objective function is restricted to a linear combination of 4 basic aggregate functions. Although these functions are appropriate most of the time, an interesting improvement would be to provide support for general objective functions. In addition, we want to compare our current algorithm against other alternatives for feature model optimization.

On the application side, we plan to bring the approach to the domain of service-oriented computing for assisting the composition and integration of computer vision applications. A challenge here is how to deal with concerns related to mobile and distributed computing. In a distributed context, quality attributes like security, availability, scalability and battery consumption can be more difficult to manage than in centralized systems. Furthermore, if we think about a set of distributed (and cooperative) applications, they will be expected to negotiate wisely their individual preferences for quality attributes in order to consider the global perspective of quality of service. A promising alternative in this direction is to extend the proposed approach and algorithm using Distributed Constraint Optimization techniques (DCOP) [11].

References

[1] J. Bartholdt, M. Medak, and R. Oberhauser. Integrating quality modeling with feature modeling in software product lines. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 365–370, Sept 2009.

[2] Len Bass. *Software architecture in practice*. Addison-Wesley, 2 edition, April 2003.

[3] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[4] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In Oscar Pastor and João Falcão e Cunha, editors, *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2005.

[5] D. Cunha da Silva, AB. Lopes, F.AP. Pinto, and J.C. Leite. Selecting architecture configurations in self- adaptive systems using qos criteria. In *Software Components Architectures and Reuse (SBCARS), 2012 Sixth Brazilian Symposium on*, pages 71–80, Sept 2012.

[6] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. In Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko, editors, *Autonomic Computing and Networking*, pages 31–55. Springer US, 2009.

[7] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 84(12):2208 – 2221, 2011.

[8] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.

[9] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, January 1998.

[10] R.Timothy Marler and JasbirS. Arora. The weighted sum method for multi-objective optimization: new insights. *Structural and Multidisciplinary Optimization*, 41(6):853–862, 2010.

[11] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *ARTIFICIAL INTELLIGENCE*, 161:149–180, 2006.

[12] Sabine Moisan, Jean-Paul Rigault, Mathieu Acher, Philippe Collet, and Philippe Lahire. Run time adaptation of video-surveillance systems: A software modeling approach. In JamesL. Crowley, BruceA. Draper, and Monique Thonnat, editors, *Computer Vision Systems*, volume 6962 of *Lecture Notes in Computer Science*, pages 203–212. Springer Berlin Heidelberg, 2011.

[13] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, Oct 2009.

[14] J. Pearl. Heuristics: Intelligent search strategies for computer problem solving. Reading, Mass. Addison-Wesley, 1984.

[15] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar. An end-to-end approach for qos-aware service composition. In *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, pages 151–160, Sept 2009.

[16] L.E. Sanchez, S. Moisan, and J.-P. Rigault. Metrics on feature models to optimize configuration adaptation at run time. In *Combining Modelling and Search-Based Software Engineering (CMSBSE), 2013 1st International Workshop on*, pages 39–44, May 2013.

[17] A.S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 492–501, May 2013.

[18] N. Siegmund, M. Rosenmuller, M. Kuhlemann, C. Kastner, and G. Saake. Measuring non-functional properties in software product line for product derivation. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 187–194, Dec 2008.

[19] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012.

[20] Samaneh Soltani, Mohsen Asadi, Dragan Gašević, Marek Hatala, and Ebrahim Bagheri. Automated planning for feature model configuration based on functional and non-functional requirements. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 56–65, New York, NY, USA, 2012. ACM.

[21] Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268 – 1284, 2009. SI: Architectural Decisions and Rationale.