# Metrics on Feature Models to Optimize Configuration Adaptation at Run Time

Luis Emiliano Sanchez
Universidad Nacional del Centro
de la Provincia de Buenos Aires
Tandil, Argentina
lsanchez@alumnos.exa.unicen.edu.ar

Sabine Moisan, Jean-Paul Rigault
INRIA Sophia Antipolis Méditerrannée
Route des Lucioles
06902 Sophia Antipolis Cedex, France
{sabine.moisan,jean-paul.rigault}@inria.fr

***Abstract*—Feature models are widely used to capture variability, commonalities and configuration rules of software systems. We apply this technique for modeling component-based systems that exhibits many variability factors at specification, implementation, and run time levels. This representation allows us to determine the set of valid configurations to apply in a given execution context, including at run time. A key challenge is to determine which configuration should be chosen taking into account especially non-functional aspects: quality of service, performance, reconfiguration time... We propose an algorithm for selecting the configuration that optimizes a given quality metrics. This algorithm is a variant of the Best-First Search algorithm, a heuristic technique suitable for feature model optimization. The algorithm is parameterized with different strategies and heuristics on feature models producing different optimality and efficiency characteristics. We discuss the algorithm, its strategies and heuristics, and we present experimental results showing that the algorithm meets the requirements for our real time systems.**

***Index Terms*—software metrics, feature model optimization, heuristic search in graphs, real time adaptation, search-based software engineering**

## I. Introduction

*Feature Modeling* is a Model-Driven Engineering method that has emerged as a simple technique for representing commonalities and varying aspects of software product lines and their configuration rules. Features correspond to selectable concepts of a system at a chosen abstraction level: functional and non-functional requirements, environment and context restrictions, run time components... Systems and their variability are represented by *Feature Models* that can be manipulated through *model transformations*. A feature model is a set of features organized along a tree-like structure that defines the relationships among features in a hierarchical manner, with logical relations (optional, mandatory, alternative...). Extra constraints such as implication or exclusion of features restrict the valid combination of features across different sub-trees (*cross-tree constraints*). Thus a feature model defines the set of valid *configurations* of a system.

In previous work [1] we proposed the use of feature models for the representation and dynamic adaptation of component-based systems such as video-surveillance processing chains. Following *model at run time* techniques, we are able to determine the set of valid configurations to apply in a given execution context. Of course only one configuration can be applied at a given time and the problem is to select the "best" one. Here, "best" is a trade-off between several runtime *quality attributes*. A quality attribute is a monotonic quantification of an aspect of the system quality. Examples are response time, accuracy, availability...

It is thus necessary to rank the possible configurations. Our approach is to define *quality metrics* for comparing configurations. The challenge comes down to the well studied problem of *Feature Model Optimization* [2], i.e. to find a valid configuration that minimizes a given *objective function*. This combinatorial optimization problem is known to be intractable in general because the number of valid configurations increases exponentially with respect to the number of optional features. It is also a *multi-objective* (or multi-criteria) optimization problem involving several aspects simultaneously.

The solution proposed here relies on the *Best-First Search algorithm* [4] [5] [6], a well studied technique for problems that are deterministic, observable, static, and completely known, as feature models. This algorithm offers a set of proven characteristics with respect to completeness, efficiency, and optimality that meet the requirements of real time systems.

The remainder of this paper is organized as follows: Section 2 presents related works. Section 3 briefly describes the application of feature models for the dynamic adaptation of component-based systems, introduces metrics for comparing configurations and presents some examples. Section 4 details the configuration selection algorithm with its associated strategies and heuristics. Section 5 experimentally compares several variants of the algorithm. Finally, section 6 concludes and outlines future work.

## II. Related Work

We organise the related work into 3 subsections. First, in section II-A we discuss the use of heuristic-graph search algorithms for solving combinatorial optimization problems. In section II-B, we present some tools and approaches that deal with the specification, measurement and optimization of non-functional requirements on feature models. Finally, in section II-C we describe the proposed algorithms, that solve the feature model optimization problem, and present the contributions of ours.

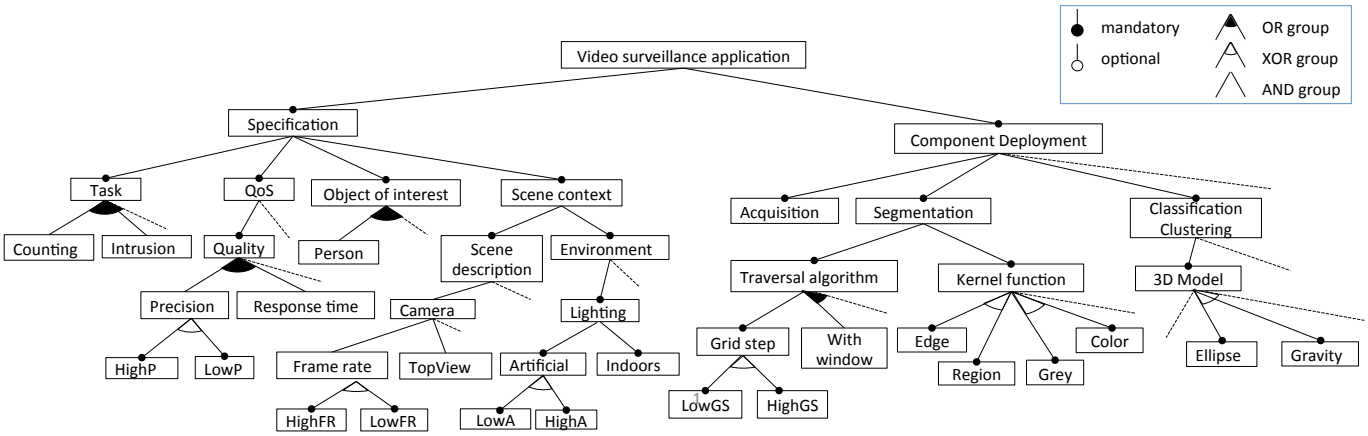


Fig. 1. Excerpts of a feature model for the specification and component view of a video surveillance system

### A. Heuristic-graph search algorithms

Combinatorial search problems in artificial intelligence can be divided into three classes: constraint satisfaction, combinatorial optimization and shortest path. Informally, in a constraint satisfaction problem (CSP) we must select values for each of a set of predefined variables such that a given set of constraints is not violated. A complete assignment that does not violate any constraints is called a solution. Combinatorial optimization problems (COPs) are like CSPs, however, there is an additional function that assigns a cost value to each solution and we wish to find the cheapest possible one. In a shortest path problem, we must select a sequence of operations that will reach a desired goal state from a given initial state and we wish to find the cheapest such sequence.

CSPs and COPs are typically solved by searching over the bounded-depth tree of partial variable assignments. Unlike them, shortest path problems are typically solved with best-first search. [20] presents a search framework called best-leaf-first search (BLFS) that brings the idea of best-first search from shortest path problems to CSPs and COPs.

We address the same issue for solving a particular COP problem (feature model optimization) but using a different approach, simple and generic to be considered as reference for solving other problems. Besides, this could be an emerging optimisation technique in the domain of Search-Based Software Engineering, because it seems that these algorithms were not used on this research domain ( [7]).

### B. Non-Functional Requirements on Feature Models

Several papers ( [12] [13] [14] [15] [16] [9] [17]) deal with non-functional requirements and its variability in software product lines (SPL) modelled by Feature Models, and provide several techniques and tools for: (i) specifying these requirements as feature attributes, (ii) measuring them for partial and full configurations, (iii) dealing with trade-off, (iv) assisting the user in the configuration process for optimizing these requirements, etc.

For example [12] proposes some techniques for dealing with trade-off and measuring non-functional requirements (as security, availability, performance, time effort) in the configuration process, based on quality functions calculated over feature attributes. [16] also presents techniques to measure non-functional requirements and an optimization process based on CSP solvers. According to [16] and [17] non-functional requirements or properties can be categorized into the the following classes depending on how they are specified in feature models:

*1) Direct assigned properties:* this class contains properties that are represented as features because the direct selection by a stakeholder is reasonable. In our case of study, these features correspond to the Quality of Service (QoS) branch of the video surveillance feature model (figure 1), and the direct selection/deselection of them is required in two levels: on one side, stakeholders need to select/deselect them during the product configuration phase; on the other side, at run time, selections and deselections of these features are triggered by events that represent context changes.

*2) Qualitative properties:* this category includes properties that can only be described qualitatively using an ordinal scale (i.e., there is no metric from which we can retrieve quantifiable measures of the overall configuration). For example: usability, security, camera resolution, etc. In [9] and [17] it is suggested the use of a mapping function from qualifier tags onto real values. Stakeholders and domain experts are required to rank these qualifier tags according to their impact on the property.

*3) Quantitative properties:* this category contains properties that can be measured on a metric scale, as execution time, accuracy, switch time delay, etc. These properties can either measure a single feature or infer the value of a whole configuration with a defined metric that aggregate the values of the properties. Different aggregation functions are suggested as metrics, but some papers ( [18] [19]) agree in considering the following as the most representative: (i) Additive aggregation function: for measuring memory consumption, development cost, response time and switch delay (if the execution is sequential), etc; (ii) Product aggregation function: availability, accuracy, etc; (iii) Maximum aggregation function: latency, operational time, response time (if the execution is parallel); (iv) Minimum aggregation function: Throughput , Security, Usability (when qualifier tags for usability, security, etc are

mapped to real values, the general property of the system can be considered as the worst of its individual elements, i.e the minimum value of its features).

Our approach allows the specification of non-functional requirements through these three means.

### C. Algorithms for Feature Model Optimization

The algorithmic approaches to feature model optimization differ in the applied technique, the application domain, and the properties concerning time efficiency and optimality.

Feature model optimization can be addressed as a Constraint Satisfaction Problem in [8]. General purpose CSP solvers can cope with any mathematical expression as objective function. However, this approach has exponential complexity because it requires to explore and evaluate all configurations.

Planning algorithms can find an approximate solution. Besides, [9] deals with the multi-objective nature of the problem, by defining the objective function as a linear combination of different quality attributes, weighted along stakeholder preferences. In addition, it takes into account inequality constraints to deal, e.g., with bounded resource. However, this approach is limited to additive objective functions.

Other solutions are inspired by traditional AI algorithms. For instance, [11] defines a feature model transformation, called *Filtered Cartesian Flattening*, that refactors a feature model so that the optimization problem becomes a Multi-Dimensional Multi-Choice Knapsack Problem. Although this approach takes into account inequality constraints, the objective function is limited to a single quality attribute and therefore cannot deal with the optimization of several requirements simultaneously.

Configuration selection may also be based on genetic algorithms. In [10] the authors address the optimization problem as in the previously mentioned work [11]. Therefore, it cannot deal simultaneously with several criteria either.

Several aspects distinguish our work. First, we want to address optimization of feature models in *real time*, whereas most studies consider static analysis of feature models and configuration selection of software product lines at design time only. Secondly, our approach offers different strategies and heuristics providing either an optimal solution or approximate ones. Thus, it can be applied at design time, where optimality is desirable, as well as at run time where computational efficiency is mandatory. Lastly, although we do not yet consider inequality constraints, we address the optimization of the most usual forms of objective functions suggested as quality metrics as shown in table I: addition, product, maximum and minimum of attributes, and linear combinations of them to weight requirements thus dealing with different quality attributes simultaneously.

## III. Application and Metrics

The run time *system configuration* of a video-surveillance processing chain can be simply defined as a set of running components with their configuration parameters, that can be tuned, removed, added or replaced dynamically. We use feature modeling to represent the variability of both the specification and component views of such systems, as is shown in figure 1. We also include cross-tree constraints that formalize extra feature dependencies. In this feature model, it is important to distinguish between *concrete* and *abstract* features. The first ones are low-level features that reference concrete software elements, like deployable components or configuration parameters. By contrast, abstract features are used to represent specification aspects and to organize the whole system.

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 $S \cap D = \emptyset$, $S \cup D = F$ (set of all features). Context changes or user interactions imply to dynamically reconfigure the model (selecting or deselecting features). The reconfiguration seldom results in a full system configuration but rather in a *partial configuration* (i.e., a sub-model) of the feature model. A partial configuration represents the set of valid full configurations compatible with a given 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 that $S \cup D \cup U = F$.

The objective of the *configuration selection algorithm* 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. Selection and deselection of features are feature model transformations that can propagate other selections and deselections in order to satisfy logical relations and cross-tree constraints. Of course, invalid configurations are skipped.

### A. Metrics on feature models

An *optimal* system configuration $\mathbb{C}$ minimizes a given *objective function*. In our case we consider linear weighted functions $L(\mathbb{C}) = \sum_{a \in A} w_a \times M_a(\mathbb{C})$, where $A$ is the set of quality attributes of interest, $w_a$ is the weight of quality attribute $a$, and $M_a$ is the *quality metrics* associated with attribute $a$. $M_a$ may have different forms according to the nature of the attributes. There exist a wide range of quality attributes to evaluate run-time operations of a system. Some are general: response time, memory consumption, availability... Besides, video-surveillance systems exhibit specific attributes: accuracy and sensitivity of detection or tracking algorithms, reconfiguration time, relevance of object classification... Different quality metrics functions can be used to evaluate all these attributes. We identified four of them that are appropriate most of the time and that have mathematical properties suitable for optimization in feature models. Table I presents them. The linear combination of these metrics in an objective function such as $L(\mathbb{C})$ allows us to deal simultaneously with several runtime and non-functional attributes, different weights, and different quality metrics functions.

To compute the objective function, we enriched all features in the feature model with two slots ($a_S$ and $a_D$) for each different quality attribute $a$. These slots are initialized by default to the neutral element ($e$) of their specific metrics function:

| Quality metrics function | Quality Attributes |
|---|---|
| $M^+(\mathbb{C}) = \sum_{f\in S} a_S(f) + \sum_{f\in D} a_D(f)$ | memory consumption, reconfiguration time... |
| $M^\times(\mathbb{C}) = \prod_{f\in S} a_S(f) \times \prod_{f\in D} a_D(f)$ | accuracy, availability... |
| $M^M(\mathbb{C}) = \max(\max_{f\in S}(a_S(f)), \max_{f\in D}(a_D(f)))$ | latency, response time in parallel execution... |
| $M^m(\mathbb{C}) = \min(\min_{f\in S}(a_S(f)), \min_{f\in D}(a_D(f)))$ | security... |

TABLE I
DIFFERENT FORMS OF METRICS FOR A GIVEN QUALITY ATTRIBUTE $a$.

0 for addition, 1 for product, $\infty$ for minimum, and $-\infty$ for maximum. Concrete features may have predetermined values for some attributes that can changed at runtime. Note that a feature contributes differently to the metrics when it is selected ($a_S(f)$) as well as when it is deselected ($a_D(f)$). For most quality attributes, e.g. memory consumption or response time, deselected features do not contribute at all ($a_D(f) = e$) but for computing other attributes, e.g. reconfiguration time, some of them do: for instance, if a feature has just been selected, we may take into account the corresponding component setup time whereas when it is deselected we consider a shutdown time.

Owing to the tree structure of feature models and their configurations, the metrics in table I can be recursively computed for each quality attribute. Equation 1 shows the evaluation of a quality attribute $a$ in the particular case of an additive metric ($M^+$ in table I).

$$M_a^+(\mathbb{C}) = R_a^+(r)$$
$$R_a^+(f) = \begin{cases} a_S(f) + \sum_{f'\in F'} R_a^+(f') & \text{if } f \in S \\ a_D(f) + \sum_{f'\in F'} R_a^+(f') & \text{if } f \in D \end{cases} \quad (1)$$

The recursive evaluation starts from the root feature $r$ of the model, that belongs to $S$ by definition. $F'$ is the set of children of a feature $f$. Due to the existence of a neutral element and the associativity and commutativity of the four quality metrics, the metrics in table I and the recursive equations (1) are equivalent. Therefore, the linear weighted objective function can also be redefined as a linear combination of the recursive functions $R_a$ as shown in equation 5.

$$L(\mathbb{C}) = \sum_{a\in A} w_a \times M_a(\mathbb{C}) = \sum_{a\in A} w_a \times R_a(r). \quad (2)$$

### B. Examples

In this section some examples will be presented to show the specification of runtime attributes of component-based systems modelled by feature models, and its evaluation with different quality metrics.

Let's take the feature model in figure 2 as example. Here, components and configuration parameters of a video surveillance processing chain are mapped to features. Their runtime properties can be assigned to feature slots $a_S$ and $a_D$. There are two kinds of runtime properties depending on whether deselected features contribute or not to the evaluation. In our application (at least for now) only reconfiguration time takes into account deselected features. For the rest of attributes, like QoS, response time, accuracy, etc, deselected features do not contribute at all, therefore $a_D$ slots are set to default values (neutral element).

For example, we can compute the reconfiguration time with an additive metric, since it is the addition of the *startup time*, *shutdown time* and *parameter configuration time* of the added, removed and tuned components respectively. For instance, if the *Shadow Removal* component is currently running and a new reconfiguration is required, $a_S(ShadowRemoval) = 0seconds$ because the selection of this feature does not have any impact on the reconfiguration time since this component is already in execution. On the other hand, $a_D(ShadowRemoval) = ShutdownTime(ShadowRemovel)$, because the deselection of this feature implies the removal of the component. In the same way, if the *Shadow Removal* component is not running in the current system configuration and a new reconfiguration is required, $a_D(ShadowRemoval) = 0seconds$ and $a_S(ShadowRemoval) = StartupTime(ShadowRemoval)$.

Latency and response time can be measured with an additive or maximum metric depending on the execution context of components, i.e. if they are running in the same thread or in parallel. The system availability (or reliability) can be measured with a product metric because each component may have a probability of fail, and the overall system availability is the product of these probabilities. In the same way, due to the pipeline architecture of the video surveillance system, the accuracy of the required functionality (for example, face detection) can be computed as the product of the accuracy of the components that are involved in the implementation of this function, because the output of a component is the input of the next one. So, if one fails (due to a wrong image acquisition or segmentation, or giving a false negative or false positive) the rest of the chain fails. For instance, if the tracking function depends on both components (*Frame to Frame* and *Long Term*), the accuracy (and also the sensitivity) can be computed as the product of the accuracy of both components, because if one component returns a false negative, the other component works over an incorrect result.

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 not the acquisition component and its parameters. The same happens with security, usability and other quality attributes. For example, let's suppose that our video surveillance system is running

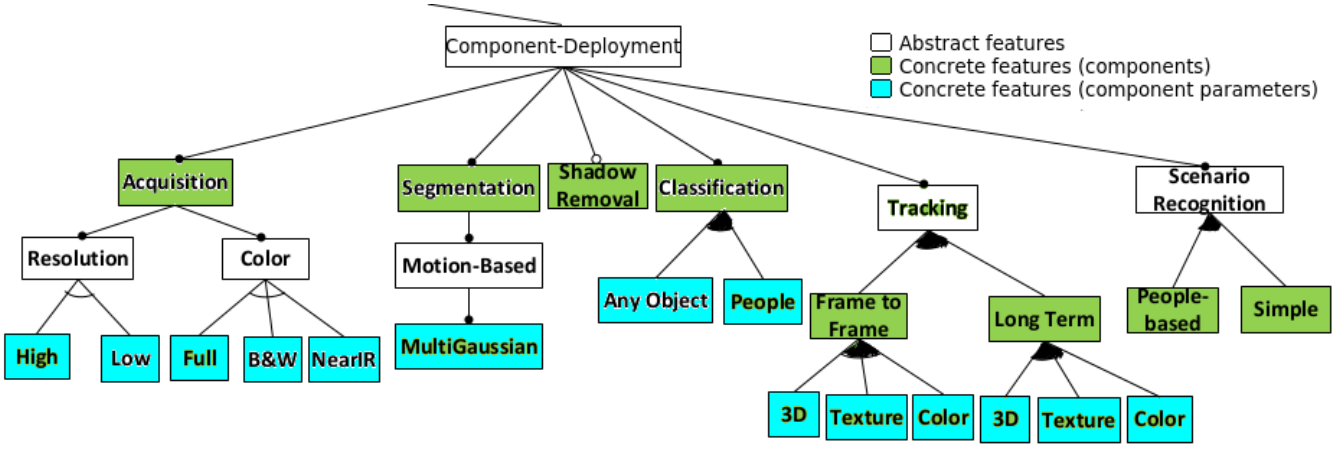


Fig. 2. Excerpts of a feature model for the component view of a video surveillance processing chain

on an online environment, connected to the Internet, to be acceded remotely. We can measure the security of the system as the minimum value of the security components (firewalls, authentication module, etc) because the overall vulnerability of the system depends on the most vulnerable component. The security degree can be specified as a qualitative attribute, using qualifier tags like *log, medium or high security*, and then mapped to numeric values. For those other components and parameters that a particular attribute does not apply, we set the $a_S$ value with the neutral element, like abstract features.

With these simply examples, we show the potential of possibilities of the suggested metrics for measuring different runtime and non-functional aspects. Lastly, the linear weighed function is fundamental for grouping all these attributes with different metrics and weights to evaluate the overall quality of the candidate system configurations. The weights are important not only for prioritizing attributes, besides whether they are positive or negative, the attributes will be minimized or maximized respectively.

## IV. Configuration Selection Algorithm

The configuration selection algorithm is based on the Best-First Search algorithm. This algorithm performs a systematic search over an abstract structure called *state-space graph*. In our problem, this structure is a directed graph where nodes are valid states of the problem (partial and full valid configurations) and edges are feature model transformations (selection and deselection of unselected features).

From a given initial partial configuration, the configuration selection algorithm (Algorithm 1) generates new nodes by selecting and deselecting features. It uses an heuristic function to estimate the quality of these nodes in order to drive the search. The algorithm succeeds when it reaches a full configuration (goal node). Due to the directed graph structure of the state-space graph, the algorithm needs to remember the visited nodes in order to avoid redundant paths. As this compromises the efficiency of the algorithm, we build the state-space graph as a binary tree. To do that, for each partial configuration (nodes of the state-space graph), the algorithm chooses one unselected feature and generates at most two successors, one selecting this feature and the other deselecting it. A simple example of a binary tree structured state-space graph is depicted in figure 3.

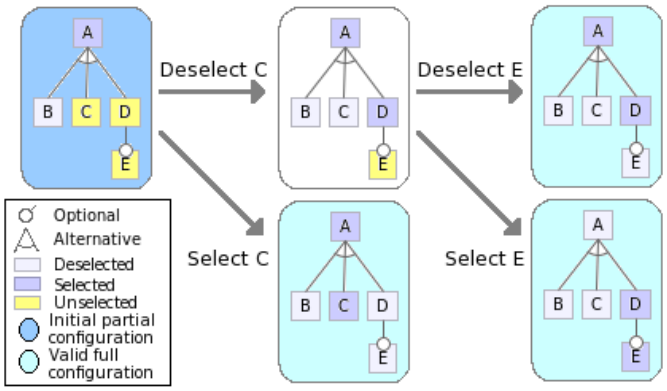


Fig. 3. Example state-space graph

The algorithm, implemented in C++, has three variation points. First, the *OPEN set* is a container where the algorithm adds and removes the generated nodes together with their heuristic value, that is used to prioritize the partial solutions. It can be implemented with different structures, like a stack or priority queue, corresponding to different *search strategies* (discussed in section IV-A).

Second, *the heuristics* $h$ (lines 11 and 15 of Algorithm 1) constitutes another variation point. Each node of the state-graph represents generally a partial configuration, that is a *set* of full configurations. The later set increases exponentially with the number of unselected features. Thus exactly determining the value of the optimal full configuration with respect to an objective function is difficult. Hence the algorithms uses an *heuristic function* to get a quick estimate of the objective function. Several heuristics are possible and discussed in section IV-B.

Last, *feature selection* (line 8) chooses the next unselected feature candidate to be selected/deselected. Following existing work in Constraint Satisfaction Problems, we intend to choose features in order to avoid decisions that can lead to invalid

**Algorithm 1** Configuration Selection Algorithm

**Input:** $initial$ // partial configuration
**Output:** $goal$ // full configuration
1: add $initial$ into $OPEN$ set
2: **while** $OPEN$ is not empty **do**
3: remove some configuration $conf$ from $OPEN$
4: **if** $conf$ is full configuration **then**
5: $goal \leftarrow conf$
6: exit successfully
7: **else**
8: $f \leftarrow$ next unselected feature in $conf$ // feature selection
9: $nS \leftarrow conf$ where $f$ is selected
10: **if** $nS$ is valid **then**
11: evaluate $nS$ with $h$; add <$nS$,$h(nS)$> into $OPEN$
12: **end if**
13: $nD \leftarrow conf$ where $f$ is deselected
14: **if** $nD$ is valid **then**
15: evaluate $nD$ with $h$; add <$nD$,$h(nD)$> into $OPEN$
16: **end if**
17: **end if**
18: **end while**

nodes. For instance, *structural metrics* on feature models (measuring size, complexity and other structural properties) could be used for this purpose. However, in our current implementation we merely choose the unselected features in an arbitrary order; indeed, we prefer to focus on the interactions between the search strategies and their companion heuristics.

### A. Search Strategies

Search strategies decide which node to visit and expand next, defining different graph traversals. These strategies are classified in uninformed and informed. *Uninformed search strategies* (also called blind search) are those which have no additional information about states beyond that provided in the problem definition. Well known uninformed strategies are *Depth-First Search* (DFS) and *Breath-First Search* (BFS). Instead, informed search or *heuristic search strategies* order and visit nodes according to which node is "more promising" following heuristic criteria.

In our approach, different search strategies can be performed according to the data structure (or container implementation) used as OPEN set. For example, DFS is achieved by a stack, where nodes are pulled in *last-in first-out* order, whereas BFS by means of a queue, *first-in first out* order. However, these strategies are not suitable for searching optimal or sub-optimal solutions. In its place, we choose two well-known informed strategies that rely on heuristic functions as choice criteria and that present interesting properties in terms of efficiency and optimality. They are analyzed empirically in section V.

*1) Best-First search Star (BF*):* This strategy is a generalization of the A* algorithm and is described in [4] under the name BF*. To implement it, we use a priority queue as OPEN set, where nodes are pulled, hence visited, in a priority order given by the heuristic functions described in section IV-B. We use node-cost functions instead of path-cost ones as in original A*.

This strategy favors optimality over efficiency. If the heuristic function is *exact*, i.e. if it computes the value of the best full configuration, the corresponding search reaches the optimal solution in the minimum number of visited nodes. As the heuristics becomes less exact, this strategy backtracks more often. However, it still ensures the optimality of the solution if the heuristics is *admissible*, i.e. if it never overestimates the value of the best solution, yet with exponential time and memory complexity in the worst case.

*2) Greedy Best-First Search (GBFS):* In this strategy the next visited node is the best successor of the current node. Specifically, it performs a depth-first search taking into account the heuristic value of nodes to decide which branches expand of the state-space graph. It is implemented by means of a data structure that we call *priority stack*: nodes are added into a priority queue, but when a removal is required, the priority queue is emptied into a stack and the node is removed from the stack.

In contrast to BF*, it favors efficiency over optimality because it only backtracks when an invalid node is reached, as backtracking algorithms. If the heuristic is exact, it behaves as BF* finding the optimal solution. This strategy has exponential time cost in general but polynomial cost on feature models without cross-tree constraints (as confirmed by our experiments, see V). Regarding memory complexity, it is linear in general with respect to the number of unselected features.

### B. Heuristics

Heuristics evaluate a partial configuration giving a quick estimate of the quality (value of the linear weighted function used as objective function) of the best element in the set of full configurations that it represents. The more exact the heuristics is, the more optimal and efficient the search strategies are. GBFS and BF* require the exact best value to drive the search to the optimal solution in optimal time. Unfortunately, due to cross-tree constraints, exact functions evaluable in polynomial time can seldom be defined. Yet, we propose exact heuristics for *relaxed models* of the problem that can be used as admissible heuristics for the general problem, in such a way that BF* can reach an optimal solution within reasonable time and GBFS can improve the optimality of the approximate solution.

A *relaxed model* [4] is a model with fewer restrictions. The state-space graph of a relaxed model is a super-graph of the original solution space since removing restrictions introduces new nodes and edges in the graph. As the relaxed model just extends the state-space graph, any optimal solution of the original problem is also a solution (not necessarily optimal) in the relaxed version. Hence, an exact heuristics for a relaxed model is admissible for the original model.

We defined two types of relaxed models $A$ and $B$ such that, for a given feature model $FM$ (with any arrangement of AND, OR, XOR groups and cross-tree constraints), its relaxed models $FM_A$ and $FM_B$ are *generalizations* of $FM$, i.e., $[[FM]] \subseteq [[FM_B]] \subseteq [[FM_A]]$ where $[[FM]]$ denotes the set of valid full configurations of $FM$. For each relaxed model, we defined an *exact* heuristic function (resp. $H_A$ and $H_B$) that is thus *admissible* for general feature models.

A feature model of type B is any that has no cross-tree constraints. A feature model of type A is also of type B (without cross-tree constraints) and has only AND groups, i.e. optional and mandatory features. A simple example with a feature model and its relaxed versions is depicted in figure 4. Heuristic functions evaluate partial configurations considering these relaxations.

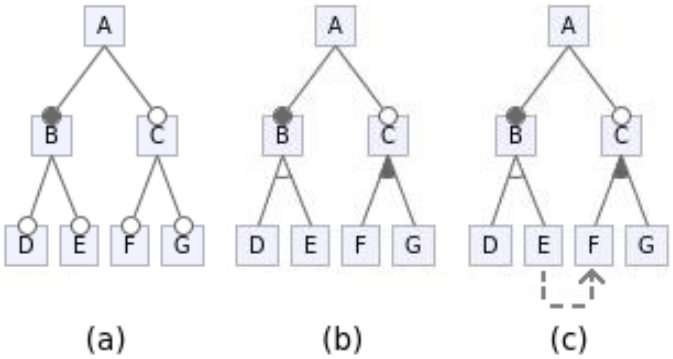


Fig. 4. Example relaxed models: (a) type A, (b) type B, (c) original model

*1) Heuristic A:* this heuristic function ignores cross-tree constraints and considers features in XOR and OR groups as optional (AND groups). To evaluate partial configurations $H_A$ is defined recursively as in equation (1) with an extra term for unselected features. For the latter, we choose the best (minimum) value between selecting and deselecting the feature. That is why the function is defined in terms of two recursive functions, $HA_S$ and $HA_D$, for computing the heuristic value of a branch considering its root feature selected and deselected respectively. Equation 3 describes the computation of $H_A$ to minimize a quality attribute $a$ with an additive associated metrics.

$$\begin{aligned} HA^+(r) =& HA_S^+(r) \\ HA_S^+(f) =& a_S(f) + \sum_{f' \in S'} HA_S^+(f') + \sum_{f' \in D'} HA_D^+(f') \\ &+ \sum_{f' \in U'_{mand}} HA_S^+(f') \\ &+ \sum_{f' \in U'_{others}} \min(HA_S^+(f'), HA_D^+(f')) \\ HA_D^+(f) =& a_D(f) + \sum_{f' \in F'} HA_D^+(f') \end{aligned} \tag{3}$$

Some considerations: (i) the heuristic evaluation starts from the root feature $r$ that is selected in partial configurations by definition; (ii) $U'_{mand}$ represents the set of unselected subfeatures of $f$ that are mandatory, hence they are computed as selected; (iii) $U'_{others}$ represents the set of unselected subfeatures of $f$ that are not mandatory, i.e. optional or alternatives (XOR and OR features), hence both functions ($HA_S$ and $HA_D$) are computed and the minimum value is aggregated; (iv) in $HA_D$ all subfeatures of $f$ are treated as deselected.

By analogy, this heuristic function can be also defined for the product, maximum and minimum metrics, because the four metrics are *commutative monoids*: an algebraic structure with a single associative and commutative binary operation and an identity or neutral element.

*2) Heuristic B:* this heuristic function only ignores cross-tree constraints. $H_B$ is similar to $H_A$ but unselected features in XOR and OR groups are treated differently. Equation 4 describes the computation of $H_B$ to minimize a quality attribute $a$ with an additive metrics.

$$\begin{aligned} HB^+(r) =& HB_S^+(r) \\ HB_S^+(f) =& a_S(f) + \sum_{f' \in S'} HB_S^+(f') + \sum_{f' \in D'} HB_D^+(f') \\ &+ \sum_{f' \in U'_{mand}} HB_S^+(f') \\ &+ \sum_{f' \in U'_{opt}} \min(HB_S^+(f'), HB_D^+(f')) \\ &+ \sum_{f' \in U'_{or}} \min(HB_S^+(f'), HB_D^+(f')) + \alpha \\ &+ \sum_{f' \in U'_{xor}} HB_D^+(f') + \alpha \\ HB_D^+(f) =& a_D(f) + \sum_{f' \in F'} HB_D^+(f') \end{aligned} \tag{4}$$

OR-unselected features ($U'_{or}$) are similar to optional-unselected features ($U'_{opt}$), but at least one of them must be selected. In case that $\forall f' \in U'_{or}, HB_D^+(f') < HB_S^+(f')$, this implies that no features are considered selected in the evaluation, therefore the value $\alpha = HB_S^+(f'') - HB_D^+(f'')$ is added where $f''$ is a feature from $U'_{or}$ which value $HB_S^+(f'') - HB_D^+(f'')$ is the minimum. Note that $\alpha$ adds $HB_S^+(f'')$ and subtract $HB_D^+(f'')$ because, to consider the selection of $f''$, its deselected value must be subtracted because in the previous term it was considered deselected.

For XOR-unselected features ($U'_{xor}$) only one of them must be selected. Therefore all are computed as deselected ($HB_D^+(f')$) and $\alpha$ is added to consider the selection of the feature with minimum $HB_S$ value. In brief, $\alpha$ represents the selection of at least one feature.

This algorithm for computing the exact value of B relaxed models can be also applied for the product metric, because it has an inverse operator too (division). Hence, for the product metric $\alpha = HB_S^+(f'')/HB_D^+(f'')$. For the maximum and minimum metrics, another algorithm is used for computing the exact value, but due to its complexity it is not worth to explain.

These heuristics evaluate partial configurations from leaves to root with polynomial time complexity ($O(n^2)$, where $n$ is the number of unselected features). These heuristics are equivalent to equation (1) to evaluate full configurations, and thereby they are also equivalent to quality metrics in table I. Equation 5 applies, replacing $R$ with $H_A$ or $H_B$. Therefore, a general heuristics to estimate the value of a linear

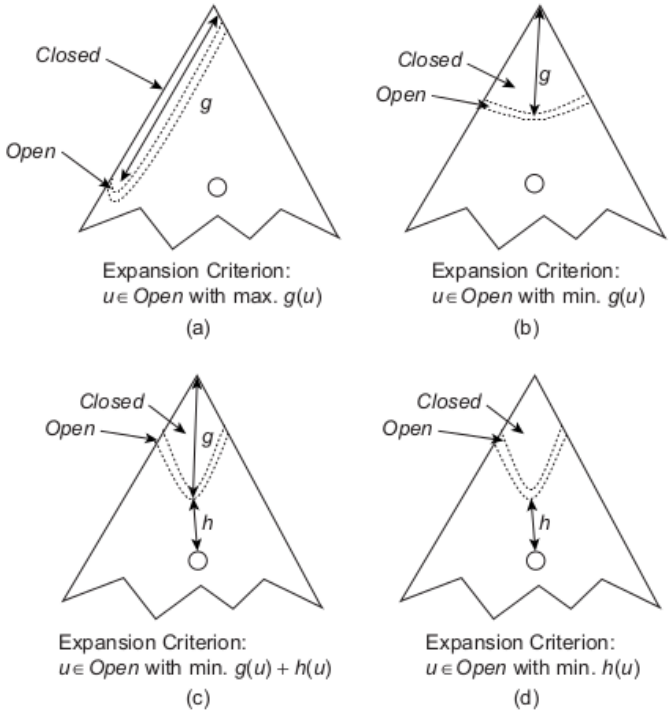


Fig. 5. Different search strategies: (a) DFS, (b) BFS, (c) A*, and (d) greedy best-first search.

weighted objective function can be defined as the same linear combination of heuristic functions.

$H_A$ and $H_B$ are exact for the relaxed models $A$ and $B$ when the linear weighted function is reduced to one attribute (only one metrics). For the general case, these functions are not exact but only admissible: the estimated value is always less or equal than the real value of the best solution. The admissibility of these heuristics is an important property that ensures that solutions of BF* strategy are optimal for general linear weighted functions and general feature models.

The admissibility is a property that also holds with any linear or non-linear combination of metrics and heuristics. Therefore we could define more complex expressions for comparing configurations and BF* could also get the optimal solution. For example, the linear weighted function suggested in [9] could be used:

$$L(\mathbb{C}) = \sum_{a \in A} w_a \times \frac{M_a(\mathbb{C}) - \mu_a}{\sigma_a} \quad (5)$$

where $\mu_a$ and $\sigma_a$ are the average value and the standard deviation of each quality metric. The objective here is to normalize the order of magnitude and unified the measuring units (response time in miliseconds, memory consumption in megabits, etc) of different attributes.

### C. Comparison with Best-First Search for shortest path problems

Best-First Search is an instance of the general graph-search algorithm in which a node $n$ is selected for expansion based on an evaluation function $f(n)$. The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. This is done by storing the visited nodes in a priority queue ordered by $f(n)$. A* is a variant of Best-First Search for solving shortest path problems where $f(n)$ is defined as a path-cost function $g(n)+h(n)$, where $g(n)$ is the path distance from the initial node to a node $n$ defined as the addition of the edge weights, and $h(n)$ is an estimated cost of the cheapest path from $n$ to a goal node. Greedy Best-First Search is another variant of Best-First Search, also for shortest path problems, where $f(n) = h(n)$. Therefore it favors efficiency over optimality. A graphical representation of the main graph-search strategies are depicted in figure 5. This figure is quoted from [6].

These informed strategies work over graph search problems where a weighted directed graph is given (edges are directed and have a cost). In our problem, the edges, that represent selection and deselection of features, are directed but do not have any cost. That is why we define heuristics as node-cost functions. We may see an analogy between our heuristic functions and $f(n) = g(n) + h(n)$: selected and deselected features contribute to the heuristic value with a concrete and certain value, like $g(n)$, while unselected features contribute with an uncertain value that must be estimated, like $h(n)$. Another remarkable difference is the use of the *priority stack* structure presented in section IV-A2 to emulate the behaviour of the Greedy Best-First Search strategy.

## V. Evaluation

We have run experiments and compared completeness, optimality and efficiency of the previous algorithms and heuristics. Regarding completeness, Best-First Search is a *systematic* algorithm [4]: it visits all the nodes in the worst case, but only once, which ensures completeness and termination.

We have already discussed in the previous section time and memory complexity of search strategies and optimality and efficiency of BF* when heuristics are admissible or exact. In this section we empirically measure these properties with a series of experiments using randomly generated feature models. Of course these models are validated, i.e. we verify that they have at least one valid full configuration.

Both search strategies (GBFS and BF*) combined with both heuristics ($H_A$ and $H_B$) are tested and compared. Three main feature model parameters are considered for analyzing the scalability of these variants: number of features ($|F|$), number of cross-tree constraints, and number of terms (i.e., number of quality metrics $|A|$) of the linear objective function. We focused on optimality of solutions and execution time of algorithms, the most representative results for comparison purposes. Each experiment is performed with at least 100 feature models as samples, and the average value is computed in order to reduce the fluctuations due to random generation. All measurements were performed on Linux Fedora 14 with Intel Xeon CPU 2.4 GHz and 16 GB RAM.

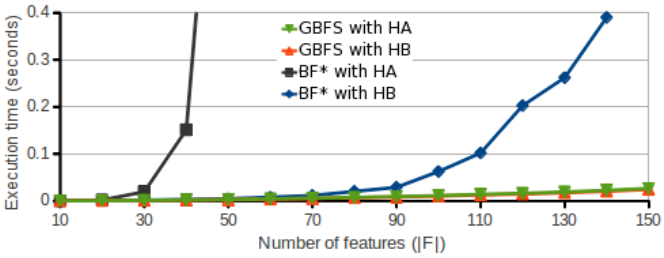


Fig. 6. Time efficiency comparison. BF* execution time rises exponentially and GBFS polynomially (green and red GBFS curves overlap).

### A. Scalability

The first experiment (figure 6) evaluates the scalability of the four variants of the algorithms with respect to the total number of features. The feature models are generated as balanced trees with a branching factor of 5 children per feature. Feature relationships are selected randomly among *mandatory* or *optional*, XOR (alternative) and OR groups. The quality attribute values are set randomly with a uniform distribution over (0,1). No cross-tree constraints are included. The objective function is a linear weighted combination of four different quality metrics: $L(\mathbb{C}) = w_1 \times M^{+}_{a_1}(\mathbb{C}) + w_2 \times M^{\times}_{a_2}(\mathbb{C}) + w_3 \times M^{M}_{a_3}(\mathbb{C}) + w_4 \times M^{m}_{a_4}(\mathbb{C})$.

Since there are no cross-tree constraints, the GBFS strategy never backtracks and both GBFS versions are fully scalable with respect to the number of features, as shown on figure 6. For BF*, both versions obtain the optimal solution but one can see that performance strongly depends on the heuristics used. Therefore heuristics B is obviously better because of its outstanding improvement in performance.

### B. Efficiency of heuristics in BF*

The efficiency of heuristics is also analyzed with respect to others parameters. Figure 7 shows two experiments, where the number of cross-tree constraints and the number of terms in the objective function vary.

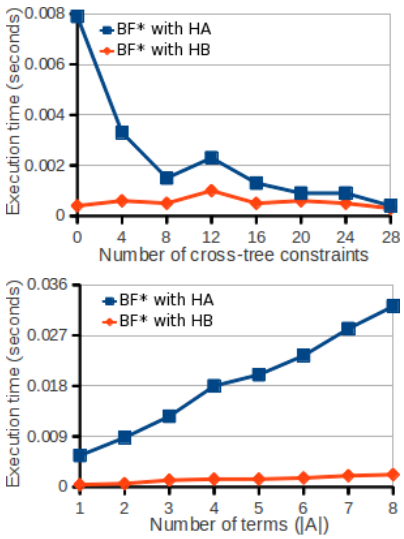


Fig. 7. Time efficiency comparison of heuristics using BF* strategy

In the first experiment (up), we consider a single additive function to minimize (i.e., a single quality attribute). Feature models are generated randomly with 30 features, because with this value the execution time of both heuristic functions is almost equal, as seen on figure 6. Simple cross-tree constraints are added, like "imply" and "exclude", that are 2-SAT clauses in conjunctive normal form (e.g $f_1 \rightarrow f_2 \equiv \neg f_1 \vee f_2$). The curves show how the time efficiency improves as cross-tree constraints are added, because incorporating restrictions reduces exponentially the number of valid configurations. The general case with random $k$-SAT cross-tree constraint clauses is not considered in these experiments but the algorithm can cope with it.

In the second experiment (down), feature models without cross-tree constraints are evaluated. For each instance of the experiment, a new metrics that evaluates the contribution of a different attribute is added to the linear objective function. The curve shows how the execution time increases almost linearly with the number of metrics, because the computation of linear weighted functions has linear complexity respect to the number of terms.

The linear combination of heuristic functions becomes less exact (but still admissible) with respect to the linear combination of quality metrics as new terms are added, leading to worse performance of BF*. This fact implies an exponential curve that can not be appreciated on this experiment with few terms and features.

### C. Optimality of heuristics in GBFS

The execution time of both heuristics is similar. Therefore they do not have a significant impact on the efficiency of the GBFS strategy, as shown in figure 6. However, it is interesting to compare the impact of the heuristics on the optimality of solutions, the strategy being constant. Figure 8 shows two experiments using the GBFS strategy, where the number of cross-tree constraints and the number of evaluated metrics vary. The optimality of a solution is measured by the following ratio: $|worst - solution| \,/\, |worst - best|$, where the value of best and worst solutions are calculated with BF*.

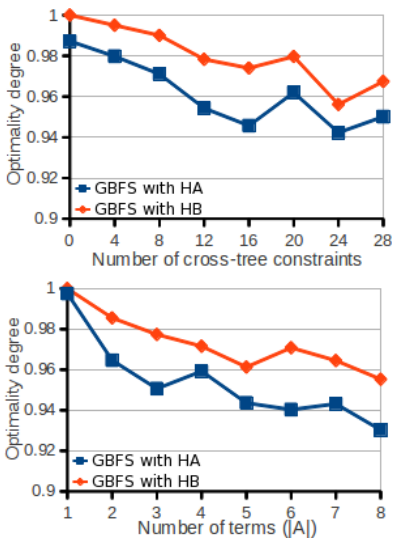


Fig. 8. Optimality comparison of heuristics using GBFS strategy

Both experiments were conducted in the same conditions as before (30 features per sample, branching factor of 5, one single additive metrics to minimize for the first experiment, and no cross-tree constraints for the second). They show how optimality decreases as the density of constraints increases and as the linear objective function incorporates more terms, because, as explained before, under these conditions the heuristics becomes less exact. However, both heuristics yield configurations that

are more than $90\%$ optimal. Note that GBFS with $H_B$ obtains the optimal solution (optimality degree = 1) on feature models without cross-tree constraints and using a one term linear weighed function. In other experiment, we have shown that the number of features has no impact on optimality.

## VI. Conclusion and future work

From our experiments, the GBFS strategy with heuristics $H_B$ appears as the ideal option for real time systems that have to adapt in bounded time. This strategy ensures polynomial time complexity for relaxed feature models and guarantees over $90\%$ optimality, which is good enough for our purpose.

On the other hand, BF* strategy with heuristics $H_B$ is ideal for assisting design decisions, such as product configuration and generation of software product lines from feature models. This search strategy takes a significant time to compute a solution. It is not appropriate in real time unless the feature models have less that about 130-140 features for which the execution time is about $0.4$ seconds. But for decisions during design, extra time is acceptable to obtain the optimal configuration.

This work only considers the most important variants of the Best-First Search algorithm. However, we can extend the algorithm incorporating new variants in both dimensions: search strategies and heuristic functions.

Regarding search strategies, hybrid solutions could be considered [4]. Only the implementation of the $OPEN$ set (see algorithm 1) needs to be changed. For example, it could start as a priority queue and continue as a stack, combining BF* and GBFS strategies, and thereby getting different efficiency-optimality trade-offs. The A* algorithm is a popular version of the Best-First Search algorithm. Although it uses a path cost function instead of a node cost function as we do, we could integrate some of the improvements proposed by its many variants into our approach.

Regarding heuristics, we have not yet considered non-admissible ones, i.e. heuristics that overestimate the value of the best solution. They could allow us to relax the optimality requirement: BF* with non-admissible heuristics does not ensure optimality but its performance improves significantly.

The proposed approach still has a third variation point that was not explored yet. As mentioned in section IV, using CSP heuristics for feature selection could improve the performance of the algorithm in feature models with many cross-tree constraints, especially with $k$-SAT clauses (with $k > 2$) since in this case finding an optimal solution is harder [3].

Finally, the approach could be extended to incorporate inequality restrictions, e.g., maximum memory consumption, maximum number of running components... as part of the definition of the optimization problem. The general algorithm scheme doesn't need to be changed. Only the design of new specific heuristics should be considered.

## References


[1] S. Moisan, J. Rigault, M. Acher, P. Collet, P. Lahire. *Run Time Adaptation of Video-Surveillance Systems: A Software Modeling Approach*. Proc. ICVS 2011, LNCS 6962, , pp 203-212, Springer.

[2] D. Benavides, S. Segura, A. Ruiz Cortes. *Automated Analysis of Feature Models 20 Years Later: A Literature Review*. Journal Information Systems, Volume 35, Issue 6, pp 615-636. September, 2010.

[3] M. Mendonca, A. Wasowski, and K. Czarnecki. *SAT–based analysis of feature models is easy*. In Proceedings of the Software Product Line Conference, 2009.

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

[5] Artificial Intelligence: a modern approach.

[6] Heuristic Search: Theory and Application

[7] Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications. Mark Harman, S. Afshin Mansouri and Yuanyuan Zhang. April 9, 2009

[8] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. *Automated reasoning on feature models*. Advanced Information Systems Engineering: 17th Int'l conf, pp. 491–503., 2005.

[9] S. Soltani, M. Asadi, D. Gašević, M. Hatala, E. Bagheri. *Automated planning for feature model configuration based on functional and non-functional requirements*. In Proc. 16th International Software Product Line Conference. Vol. 1, pp. 56-65, 2012.

[10] J. Guo, J. White, G. Wang, J. Li, and Y. Wang. *A genetic algorithm for optimized feature selection with resource constraints in software product lines*. Journal of Systems and Software, vol. 84, no. 12, pp. 2208–2221, Dec. 2011.

[11] J. White, B. Dougherty, and D. C. Schmidt. *Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening*. J. Systems & Software, vol. 82, p. 1268–1284, 2009.

[12] Joerg Bartholdt, Roy Oberhauser, Andreas Rytina, Marcel Medak. "Integrating Quality Modeling in Software Product Lines". International Journal on Advances in Software, vol 3 no 1 and 2, year 2010.

[13] H. Zhang, S. Jarzabek, and B. Yang. "Quality Prediction and Assessment for Product Lines". LECTURE NOTES IN COMPUTER SCIENCE, pages 681–695, 2003.

[14] L. Etxeberria and G. Sagardui. "Variability Driven Quality Evaluation in Software Product Lines". In Software Product Line Conference, 2008. SPLC'08. 12th International, pages 243–252, 2008.

[15] S. Jarzabek, B. Yang, and S. Yoeun. "Addressing quality attributes in domain analysis for product lines". In Software, IEE Proceedings-, volume 153, pages 61–73, 2006.

[16] "Measuring Non-functional Properties in Software Product Lines for Product Derivation"

[17] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake, "SPL Conqueror: Toward optimization of non-functional properties in software product lines," Software Quality Journal, 2011.

[18] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar, "An End-to-End Approach for QoS-Aware Service Composition," Proc. IEEE EDOC Conf., 2009, pp. 151-160.

[19] T. Yu and K.-J. Lin, "Service selection algorithms for Web services with end-to-end QoS constraints," Information Sys. and e-Business Management, vol. 3, no. 2, pp. 103-126, 2005.

[20] Best-First Search for Bounded-Depth Trees