Sanchez *et al. Journal of Software Engineering Research and Development* (2015) 3:10
DOI 10.1186/s40411-015-0022-1

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

An approach based on feature models and quality criteria for adapting component-based systems

L. Emiliano Sanchez[1*], J. Andres Diaz-Pace[1], Alejandro Zunino[1], Sabine Moisan[2] and Jean-Paul Rigault[2]

*Correspondence: emiliano.sanchez@isistan.unicen.edu.ar
[1] ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil, Argentina
Full list of author information is available at the end of the article

Abstract

Background: Feature modeling has been widely used in domain engineering for the development and configuration of software product lines. A feature model represents the set of possible products or configurations to apply in a given context. Recently, this formalism has been applied to the runtime (re-)configuration of systems with high variability and running in changing contexts. These systems must adapt by updating their component assembly configuration at runtime, while minimizing the impact of such changes on the quality of service. For this reason the selection of a good system configuration is seen as an optimization problem based on quality attribute criteria.

Methods: We propose an approach for system adaptation based on the specification, measurement and optimization of quality attribute properties on feature models. Furthermore, we describe its integration into a platform for supporting the self-adaptation of component-based systems. Feature models are annotated with quality attribute properties and metrics, and then an efficient algorithm is used to deal with the optimization problem.

Results and conclusions: Two performance properties –frame processing time and reconfiguration time– are estimated with our model against measurements obtained from the running system to show the accuracy of metrics on feature models for estimating quality attribute properties. The results show evidence that these metrics are reasonably accurate for measuring performance properties on a realistic component-based computer vision system.

Keywords: Feature models; Runtime adaptation; Quality attributes; Optimization; Component-based software engineering

1 Background

Feature models (Kang et al. 1990) are a simple but powerful formalism for representing commonalities, varying aspects, and configuration rules of software products, which have been mostly used in Software Product Lines (SPLs). In recent works, feature models have been applied for specifying and executing dynamically adaptive systems. These systems can be conceptualized as a *dynamic software product line (DSPL)* (Hallsteinsen et al. 2008) in which variability and configuration rules are bound and checked at runtime. As in traditional SPLs, feature models are a convenient formalism for representing a DSPL and enable automated reasoning about properties of its adaptive configurations.

In (Moisan et al. 2011), 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 configurations, among others; which might vary from one application to another. From an implementation perspective, selecting the (software) components themselves, assembling them, and tuning their parameters to comply with the context might lead to different configuration variants. Moreover, the context is not fixed but evolves dynamically and thus requires runtime adaptation of the component assembly in order to keep performing with a desirable quality of service.

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 process generally involves trade-offs between several goals, such as maximizing accuracy, achieving the best performance, or choosing the simplest setup (or substitution) for the current configuration, among others.

In a previous work (Sanchez et al. 2013), we presented a heuristic search algorithm called CSA (Configuration Selection Algorithm)[1] for solving the optimization problem resulting from selecting a valid configuration of a system based on feature models. This algorithm offers different strategies for leveraging execution efficiency and optimality, and allows us to define different objective functions for comparing configurations and optimizing multiple attributes simultaneously, while adhering to resource restrictions and feature model constraints. However, this algorithm requires an infrastructure with capabilities for: monitoring context changes, activating and assembling (at runtime) components that implement specific features, and gathering suitable metrics for system properties, so as to assess various configuration alternatives.

In this article, we present the overall approach and the component-based platform in which the CSA is embedded. Our approach provides a framework for the specification, measurement and optimization of quality attribute properties[2] expressed on top of feature models. We show how these properties can be specified by means of feature attributes and evaluated with *quality metrics* in the context of feature models. The global properties of the system are computed by means of aggregate functions over the features. Along this line, we discuss the selection process carried out by our optimization algorithm, highlighting some trade-off situations between quality attributes. A key aspect of model-based approaches for adaptive systems is the ability of the model to estimate a given system property, which is correlated with the actual property observed in the running system. For instance, if our approach computes a metrics for reconfiguration time as the sum of the individual times for each reconfiguration operation, e.g., add or remove a component from system assembly, we need to ensure that the aggregate metrics is a "good predictor" for the time that the system takes to reconfigure itself.

This work extends the original SBCARS article in several ways (Sanchez et al. 2014). Compared to earlier work, we provide additional information about our approach, its application, and implementing platform. We also extend its evaluation with new experiments that assess the accuracy of the proposed metrics. To do so, we rely on a concrete implementation of our platform for managing the adaptation of a computer vision processing chain based on OpenCV libraries (Opencv project 2015). This processing chain includes components for image segmentation, motion, face detection, etc. In particular, we are focused on two properties –*reconfiguration time* and *frame processing time*– which are common in computer vision systems, and then compare predicted against measured property values. Our experiments reported an accuracy of 87.6 % and 90.6 % for these two properties, respectively. These preliminary results suggest that it is possible to predict quality attribute properties with simple aggregate functions defined on feature models.

The rest of this article is organized as follows. Section 2 presents the proposed approach. It provides background about feature models, their role in the representation and adaptation of component-based systems, and describes how quality attributes are related to feature models, estimated by metrics, and then optimized with the CSA. It also covers a process for applying the approach. Section 3 presents the platform architecture for the approach along with a particular implementation and some adaptation scenarios. In Section 4, we perform an empirical evaluation of two of the proposed metrics for estimating reconfiguration time and frame processing time, and discuss lessons learned and limitations. Section 5 analyzes related work. Finally, Section 6 presents the conclusions and outlines future research directions.

2 Approach: feature models for runtime adaptation

An adaptive system is a system whose behavior can be changed during its execution according to the user´s needs or context changes. If the system can react to changes in the operating environment, the system is called self-adaptive (Oreizy et al. 1999). These systems are usually conceptualized using a reference model for autonomic control loops called MAPE-K (IBM 2003). In MAPE-K, there is a *managed element* (software or hardware resources) that is given autonomic behavior by coupling it with an *autonomic manager*. This manager is built in terms of four functions or steps (monitor, analyze, plan and execute) that define the autonomic control loop, an internal knowledge that represents the stakeholders´ goals, the managed element, and, optionally, its execution environment.

Our approach can be viewed as an instance of the MAPE-K loop, as follows: the managed element is a video surveillance processing chain implemented in terms of software components; a feature model formalism is used for the knowledge representation (about the system and its context); and an optimization process based on an heuristic search algorithm is used for the planning step. This algorithm employs quality attribute metrics for guiding the system adaptation.

The planning aspect takes into account the data from the monitor/analyze steps to produce a series of actions to be effected by the execute step on the managed element. In simple cases, *event-condition-action* (ECA) rules are used to directly produce adaptation plans from specific event combinations with the form "when *event* occurs and *condition* holds, then execute *action*". This approach might present some problems, like possible conflicts between rules, and limitations for representing restrictions on

system configurations. Other approach for planning the system adaptation is to define a transition graph model at design time, like the *reconfiguration transition system* (RTS) in (Oliveira and Barbosa 2014). Nodes in the graph represent system configurations and edges represent reconfiguration transitions, i.e., operations to adapt the system from a configuration to another. However, the number of valid configurations might grow exponentially for a large number of system variability points due to combination of features. Therefore constructing such a graph model can be infeasible in many domains.

Our approach presents an alternative to deal with the above concerns by means of feature models. It also addresses the following challenges:

- Representation of a set of valid configurations applicable at a given execution context, by means of feature model formalisms.
- Selection of the "best" candidate from this set, taking quality criteria into account, by means of the Configuration Selection Algorithm. This optimization algorithm is based on the Best-First Search schema (Pearl 1984), a well studied technique for combinatorial optimization problems such as configuration selection on feature models. This algorithm offers a set of proven characteristics with respect to correctness, completeness, efficiency, and optimality that meet the requirements for its application on runtime systems (Sanchez et al. 2013).

2.1 Feature models

According to (Kang et al. 1990), a feature model is a compact representation of all possible products or configurations, for instance, of an SPL. These models are visually represented as features and relationships among them. Features correspond to selectable concepts of the system at different abstraction level: functional and non-functional 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 feature of the tree represents the concept being described, generally the system itself, and the remaining nodes denote branches and sub-features that dis-aggregate the main concept into several elements and concerns.

Several extensions to this basic notation have been proposed (Schobbens et al. 2007), including propositional formulas for cross-tree constraints (Batory 2005), and *extended or attributed feature model* (Benavides et al. 2005). We use these two extensions for modeling system element dependencies and specifying quality attribute properties. Generic propositional formulas allow us to define constraints and dependencies among features, such as "features *A* and *B* imply not *C*". The second extension refers to a type of feature 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.

2.1.1 Model for a video surveillance system

Figure 1 depicts an example of feature model for a video surveillance processing chain, as described in (Moisan et al. 2011). This model was designed with a focus on the separation of requirements and implementation concerns, therefore the root is decomposed into two branches or sub-models: *VSspecification* and *VScomponent*. The first sub-model represents "what to do" and includes functionality (*Application, ObjectOfInterest* features), quality parameters (*QoS* feature) desired by users, and environmental and hardware conditions (*Context* feature). The second sub-model represents the system components and their parameters, that is "how (the software) should do it".

Cross-tree constraints are used to formalize extra-feature dependencies in the form of propositional formulas. In the *VScomponent* sub-model, these constraints define configuration rules for the correct assembly of software components. For instance, the dependency "component *A* requires the interface provided by component *B*" can be modeled by the constraint "*A* implies *B*". Furthermore, cross-tree constraints between both sub-models work as a bridge between application requirements and component assemblies that realize those requirements. For instance, the constraint "*ShadowRemoval* implies *Lighting.Low*" means that if *Lighting.Low* feature, which represents a low light context condition, is selected due to a light dimming event, the *ShadowRemoval* component is deselected, i.e. it cannot be part of the current system configuration.

A component is a unit of deployment that requires or provides services to other components through specific interfaces. A system configuration $\mathbb{C}$ of a video surveillance processing chain can be defined as a set of connected and running components, each one customizable with a set of parameters, and these components can be removed, added or replaced from the system assembly at runtime, as long as they meet interface requirements.

Figure 2(a) shows a configuration instance of the video surveillance processing chain. The purpose of this system is to analyze a sequence of images, i.e. a video, to detect interesting situations or events. Its global architecture follows a pipe-and-filter architectural pattern: it is a processing chain or pipeline of software components. These components can be seen as processing elements arranged in such a way the output of each element is the input of the next one. The pipeline starts with image acquisition, then segmentation of the acquired images to group image regions into "blobs", classification of possible objects, tracking these objects from one frame to the other, and lastly scenario recognition for, e.g., intrusion detection or other scenarios. The output results might be stored

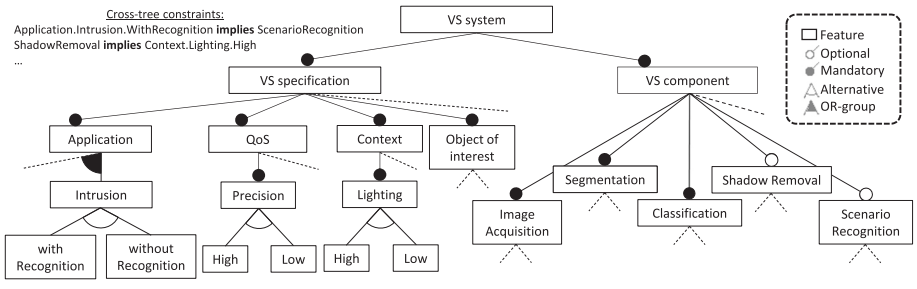
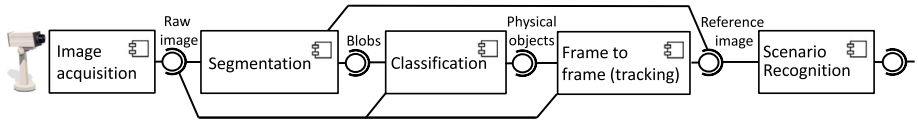
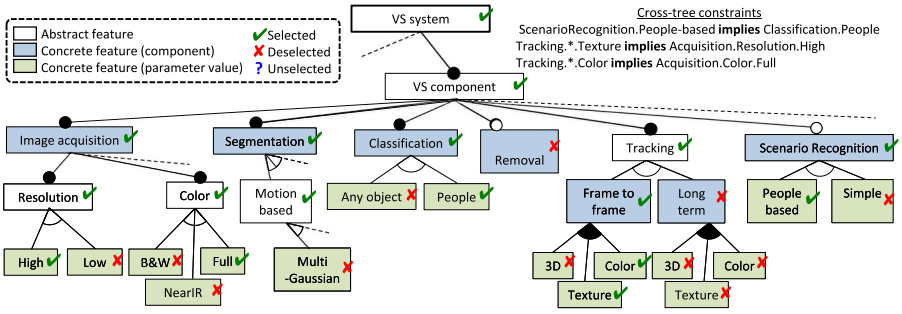


Fig. 1 Excerpt of feature model for a video surveillance system



(a) System configuration of a video surveillance processing chain



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

Fig. 2 Mapping system configuration to feature model configuration. **a** System configuration of a video surveillance processing chain. **b** Feature model full configuration of a video surveillance processing chain

for future processing, or might raise alerts to human observers. In the domain of computer vision and video surveillance systems, the processing chain can involve additional stages (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 (e.g., algorithms, strategies, input data, etc), each one corresponding to a different configuration parameter of the component.

2.2 Mapping model representation to runtime system

The mapping between the runtime system, i.e. the set of running components, and its model representation is achieved by means of *feature model configurations*. This and other key concepts for understanding the approach are presented below.

2.2.1 Full configurations

Formally, a system configuration $\mathbb{C}$ is represented by a *feature model full configuration* (simply called *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) and all constraints are satisfied. Figure 2(b) shows the full configuration that represents the system configuration in Fig. 2(a). Note that only the *VScomponent* branch is depicted due to the model size, and each running component is associated with a selected feature.

2.2.2 Partial configurations

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 of the acquisition and segmentation components of the processing chain. Users might 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.

The configuration in response to a context change is seldom a full configuration but rather a *partial configuration* of the feature model. 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 feature model constraints, resource restrictions, and the optimization of an objective function 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 performance and minimizes the required time for reconfiguring the system. This combinatorial optimization problem and the corresponding Configuration Selection Algorithm (CSA) are described in Section 2.3.4.

2.2.3 Concrete and abstract features

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. Concrete features have a one-to-one mapping to software elements. This mapping is necessary for identifying components and parameters to be tuned when reconfiguration is required. Examples of concrete features are *ImageAcquisition, Resolution.High*, etc. By contrast, the remaining features are called *abstract* and they usually correspond to high-level features used for organizing the whole diagram (e.g., *VSsystem*), grouping sets of components and parameter variants (e.g., *Tracking, Resolution*), or representing specification and context aspects (features in the *VSspecification* sub-model).

2.3 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 (Bass 2003). Quality attribute properties are typically quantified by quality metrics. Systems often fail to meet stakeholders needs regarding these attributes when they focus on some aspects without considering the impact on others. For instance, when system adaptation is required, selecting the configuration that minimizes the reconfiguration time might not be the best candidate regarding the overall *quality of service* (QoS). Furthermore, the overall QoS 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 (i.e., users) generally find it difficult to quantify their preferences in such conflicting situations.

The goal of our model-based approach for managing quality attributes is to quantitatively evaluate and trade-off multiple quality attributes to achieve 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 those metrics. The problem is formalized as the optimization of an objective function that aggregates the metrics and quantifies stakeholders' preferences for individual attributes.

In summary, 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 use of metrics for feature models to assess these quality attributes quantitatively at the system level. Finally, optimization deals with the maximization or minimization of conflicting attributes evaluated with these metrics, which are assigned to different weights in order to consider stakeholders' preferences, while still meeting configuration rules and resource restrictions.

2.3.1 Specification

Quality attribute properties must be specified at design time. 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 processing time, 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*/*properties*: 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 Fig. 1, these features correspond to the Quality of Service (*QoS*) branch.
2. *Quantitative attributes*/*properties*: 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 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*/*properties*: 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 to handle these attributes as quantitative properties.

2.3.2 Measurement

When analyzing quality attributes of component-based systems, a key aspect is how these attributes can be measured or predicted on the basis of properties of individual components, which are determined with a certain accuracy. According to (Crnkovic et al. 2005) some system quality attributes can be derived directly from the component attributes, called *directly composable properties*, like memory consumption; while others

might require a computation model that considers the system architecture together with the component attributes, called *architecture-related properties*, like performance properties. There are also system attributes that have no direct counterparts at the component level. They are rather the consequence of the system interactions with its environment, its architecture, plus attributes of various components. We focus on directly composable properties and architecture-related properties. That is, in our feature model, an attribute of a system configuration is a function of the same type of attribute of the components and features involved in that configuration.

For each attribute, the overall value of a configuration is calculated by an *aggregate function* that considers 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 (Rosenberg et al. 2009). We have considered 4 functions, described in Table 1. Although these functions limit the set of metrics supported by the approach, they are appropriate for a wide range of scenarios and have mathematical properties suitable for optimization using feature models (Sanchez et al. 2013).

To compute the aggregate functions for each attribute a, all features of the feature model are enriched with two slots, a_S and a_D. 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 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, startup time, failure probability, accuracy, etc. Although our approach permits changing 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 attributes (e.g., memory consumption or response time), deselected features do not contribute at all ($a_D(f) = e$), but for others 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 *startup time* is taken into account, whereas when the feature is deselected its *shutdown time* is considered instead.

Table 1 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)

The ranking value of a given configuration is a combination of its aggregated values. An optimal system configuration $\mathbb{C}$ is defined as one that minimizes this value. In our case, the ranking value is given by the following weighted function:

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

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, which might have different forms according to the nature of the attributes (see Table 1), 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 that $\sum_{a \in A} |w_a| = 1$.

The linear combination of these metrics in such a single objective function as in Equation 1, allows us to deal simultaneously with several attributes. That is, we transform a multi-objective optimization problem into a mono-objective one by means of a scalarization technique known as *weighted sum method* (Marler and Arora 2010). The parameters of the scalarization are the weights of each term, and they provide a simple way for specifying stakeholders´ 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.

2.3.3 Examples of property specification and measurement

Figure 3 shows an example of how component property values are mapped to feature attribute slots. Three attributes are depicted: reconfiguration time (*rtime*), memory consumption (*mem*), and accuracy (*acc*). System reconfiguration time is computed with an additive metric function if the reconfiguration operations are applied sequentially. Thus, the total time is the sum of the *startup time* and the *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

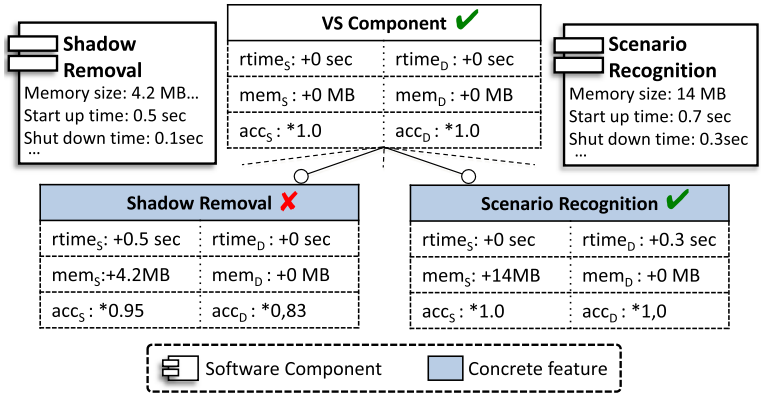


Fig. 3 Mapping example from component properties to feature slots $rtime_S$, $rtime_D$, mem_S, mem_D, acc_S, and acc_D

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 $rtime_S(SR) = StartupTime(SR)$.

Besides reconfiguration time, another interesting performance measure is the response time. Response time 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 components execution context. If several tasks are executed in the same thread, the overall required time is the addition of the required times per task; if each task is instead executed in parallel, the overall time is the maximum among the required times per task. In our video surveillance processing chain, we rename this performance measure as *frame processing time*, i.e., the time required to process a video frame. Then, the frame processing time is computed by a maximum function, if each component in the processing chain runs in parallel, or by an additive function if the execution is sequential. The accuracies of additive and maximum functions as global quality metrics for estimating reconfiguration and frame processing time are evaluated in Section 4.

Some properties might apply to some components but not to others. For instance, tracking or detection algorithms can be measured in terms of accuracy or sensitivity, but it is not the case for image acquisition. The same happens with security, usability, and other quality attributes. If an attribute does not apply to a component or parameter, its a_S value is set to the neutral element as an abstract feature.

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 and estimating the overall quality of system configuration candidates.

2.3.4 Optimization

The configuration selection or feature model optimization problem (Benavides et al. 2010) takes a partial configuration of an attributed feature model and an objective function as inputs and returns the full configuration fulfilling the criteria established by the function. Selecting the configuration that minimizes (or maximizes) the given function is an intractable combinatorial 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 bound periods of time, any optimization 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 (Sanchez et al. 2013) meets these requirements.

CSA is based on a Best-First Search schema (Pearl 1984) 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 individual 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 (or sub-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 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 1. For example, a memory consumption restriction has the form $M_a^+(\mathbb{C}) \leq \alpha$, where α is 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.

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. They have interesting properties about efficiency and optimality when they are equipped with *admissible heuristics* (Pearl 1984), i.e., a function that never overestimates the value of the best solution. Using relaxed models of the problem (Sanchez et al. 2013), we have designed admissible heuristics for the 4 aggregate functions presented in Table 1 and a linear combination of them (equation 1) in such a way that BF* can reach an optimal solution within reasonable time and GBFS can improve the optimality of the approximate solution.

For efficiency, the GBFS strategy appears as the ideal option for real-time systems that have to adapt in bound time, while BF* strategy is ideal for assisting design decisions, such as product generation of SPLs from feature models, since it guarantees the optimal solution using admissible heuristics, although it takes exponential time to compute. Details of the search strategies and heuristics are provided in (Sanchez et al. 2013), along with experimental results using randomly generated scenarios.

2.4 Process

In order to apply the approach, three activities must take place in a development process, namely: *modeling*, *mapping* and *attribute specification* (Fig. 4). The output of these activities is a configuration file with the feature model, a mapping of such features, and a quality attribute specification required by the self-adaptive platform.

The modeling activity consists of defining the feature model that represents the system assembly and other concerns (context conditions, user's requirements, etc). A system

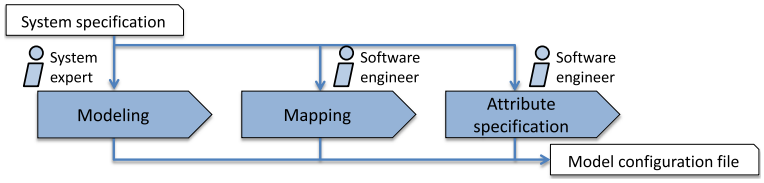


Fig. 4 Process overview

expert with knowledge of the domain application should conduct the design of the model. It must be checked at design time to guarantee consistency properties (Moisan et al. 2011). For example, for all possible execution contexts (partial configurations) at least one valid software configuration to deploy (full configuration) must exist. The modeling relies on FAMILIAR (FeAture Model scrIpt Language for manIpulation and Automatic Reasoning) (Acher et al. 2013), a scripting language and tool dedicated to the management of feature models. In particular, FAMILIAR allows us to capture the variability of the software system and its possible contextual changes, and also supports the verification of consistency properties.

The mapping and attribute specification are done by software engineers that are concerned with the system implementation. The mapping activity implies linking software elements (components and parameters) to features, and system events to feature selection/deselections (event rules). Currently, the approach is limited to one-to-one mappings between features and software elements, such as in feature-oriented software development (Apel and Kästner 2009) where features are mapped to implementation assets. The mapping must be specified manually in the configuration file, providing the name of the feature and its corresponding software element.

Finally, the attribute specification defines the attribute values and metrics for the optimization process. This specification is made manually in the same configuration file. Each metric is defined by an aggregate function over an attribute, and a specific weight, as explained in Section 2.3.2. An attribute value is defined in the configuration file providing its name, e.g., startup time, constant value, e.g., 0.5 seconds, and the associated software element, e.g., shadow removal component.

3 Platform architecture

In this section, we describe the platform that materializes the approach presented in the previous Section. The platform was designed to accomplish the following goals:

1. Provide basic functions for monitoring and re-configuring component-based system;
2. Articulate the running system with a runtime (feature-based) model, in order to represent and manipulate configurations and execution contexts;
3. Support the specification, measurement, and optimization of quality attribute properties on feature models, in order to plan and perform system adaptation minimizing the impact on quality of service.

The platform was implemented in C++. Its architecture defines three main collaborating modules, as shown in Fig. 5, which provide a simple interface that makes the design loosely coupled:

- *Component Manager* (CM): this part 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 reconfiguration operations for adapting the current configuration to the new execution context. The CM is responsible for applying these operations, hence changing the system configuration. These operations can be of the following

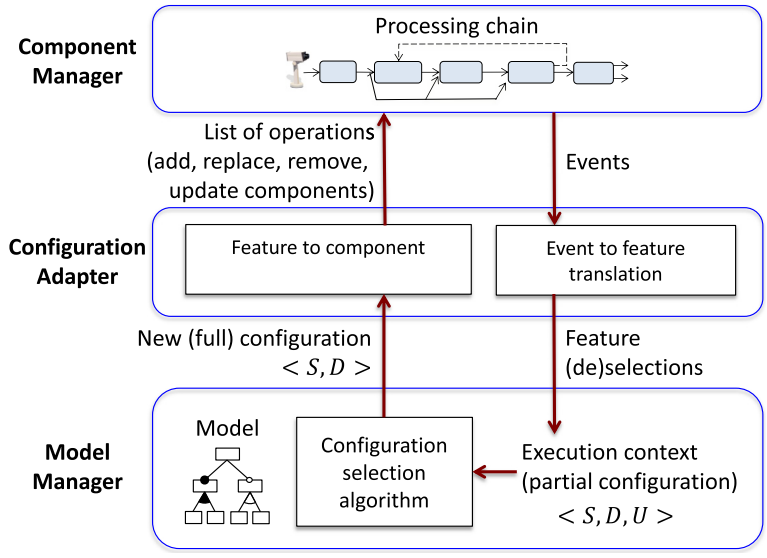


Fig. 5 Platform architecture

types: (i) *add* a component to system assembly, *(ii) remove* a component, *(iii) replace* one component with another, or *(iv) update* the parameter configuration of a component.

- *Configuration Adapter* (CA): it is a mediator 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 full configuration 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 feature model representation of the running system. Besides features and their constraints, this model includes feature attributes, resource restrictions, the 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 an 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).

The platform lifecycle and control loop is described below. For illustrative purposes, an adaptation example is then presented in Section 3.2. Details of the CM module and a particular implementation for managing the self-adaptation of a computer vision processing chain are provided in Section 3.3. This implementation is used for evaluating the accuracy of metric functions in Section 4.

3.1 Lifecycle and control loop

The mapping to the MAPE-K model is interesting to identify some architectural aspects. The implemented video processing chain corresponds to the managed element in the

MAPE-K model while the platform corresponds to the autonomic manager. The different steps that conform the autonomic manager loop are easily identified in its modules, namely: the monitor function is carried out by the CM module that captures events and software exceptions from the running system; the analyze function is done at the CA module with rules for relating runtime events and selection/deselection of features; the plan function is performed by both the CSA algorithm and the CA module that translate the new full configuration into reconfiguration operations; and finally, the execute function is done by the CM module. Feature models and quality metrics are used for the knowledge representation of the system, context and stakeholders' goals.

The general platform lifecycle is shown in Fig. 6. Essentially, it comprises 3 steps, namely: initialization, control loop, and finalization. During initialization, the modules (CM, CA and MM) are created, and two threads are started for executing the main control loop: one thread corresponds to the Component Manager thread that executes and monitors the managed system as a set of running components, while the other thread is the Configuration Adapter thread that handles events that may imply possible reconfigurations. When an event is triggered by a component, the CM thread dispatches it to the CA by sending a message. The event is queued to be later processed by the CA thread, so that the CM is not blocked and might continue with normal operation while next configuration is computed by the CA in background.

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

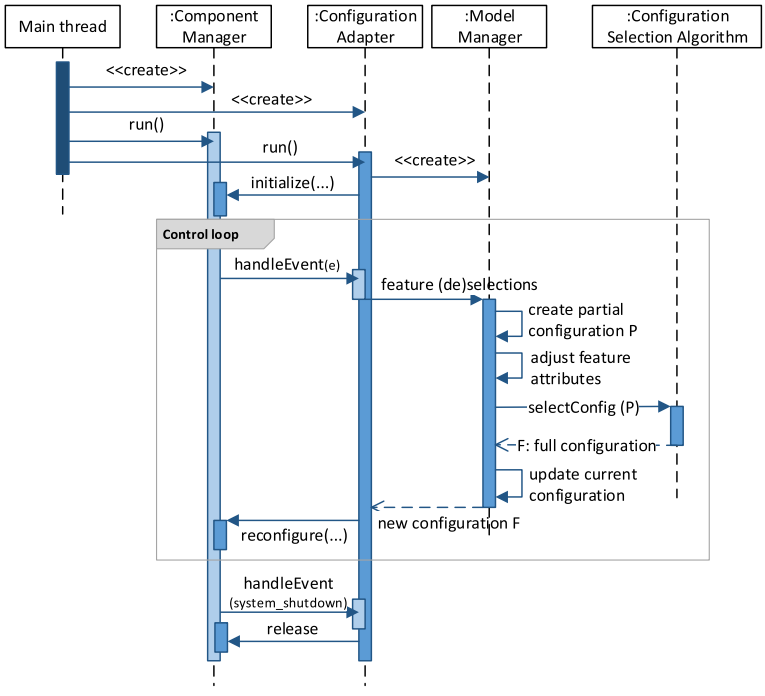


Fig. 6 Sequence diagram showing the platform lifecycle and control loop

might 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 reconfiguration operations (e.g., addition, removal, replacement, and parameter tuning of software components). The CA instructs the CM to reconfigure the managed system by providing these operations in a message, and stays idle while there are no events to handle. Finally, the platform is finalized or released when a particular *system shutdown* event is triggered, generally due to a user request.

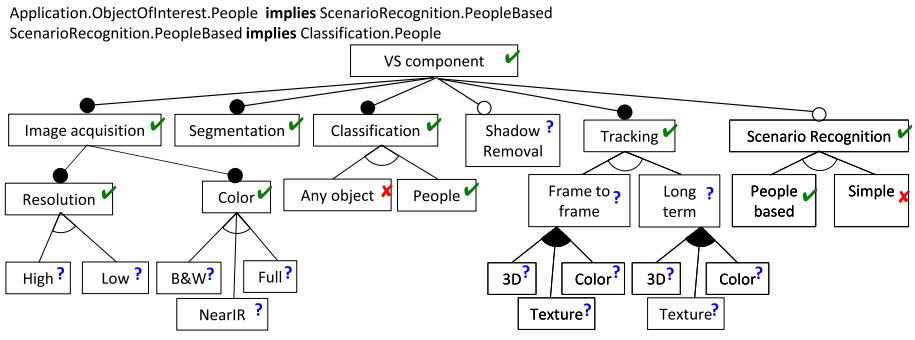
3.2 Adaptation example of a video surveillance system

To illustrate both the approach and platform, we present a simple scenario of runtime adaptation of the Video Surveillance system described in Section 2.1.1. 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 is expressed in seconds and must be minimized; and accuracy for intrusion detection (*acc*), which is a ratio between *0* and 1 and must be maximized. Then, the objective function 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.

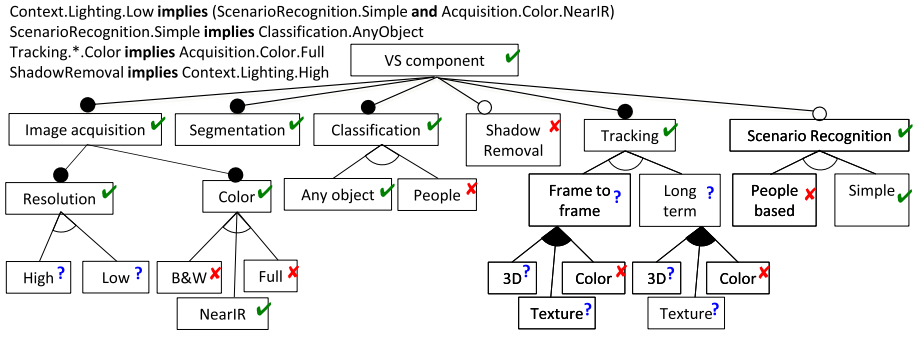
Currently we do not have a complete implementation of this system, but rather a simplified version that is used in Section 4 to empirically evaluate quality metrics. For this reason, properties weights w_{rtime} and w_{acc}, and feature attributes in Fig. 2(b) were configured manually. For these values, $M^{\times}_{acc}(\mathbb{C})$ 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 Fig. 7(a). 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 Fig. 8(a), 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), 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 Fig. 8(a), 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



(a) Initial partial configuration for intrusion detection with people recognition

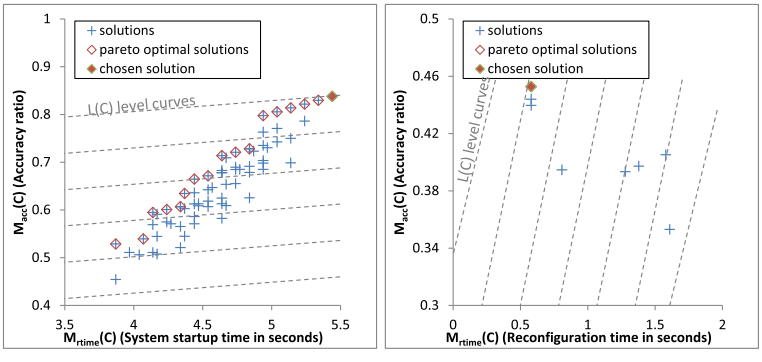


(b) New partial configuration for intrusion detection with light dimming

Fig. 7 Partial configuration examples. **a** Initial partial configuration for intrusion detection with people recognition. **b** New partial configuration for intrusion detection with light dimming

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 point in 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



(a) Derived from initial partial configuration (b) Derived from new partial configuration

Fig. 8 Solution sets. **a** Derived from initial partial configuration. **b** Derived from new partial configuration

stage). 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 Fig. 7(b). 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. Fig. 8(b) depicts the set of solutions (8 possible full configurations) derived from the partial configuration in Fig. 7(b). 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.

3.3 Component manager implementation

The Component Manager is an abstract class defined as a thread with three pure virtual (abstract) methods that are invoked by the Configuration Adapter to initialize, reconfigure and release the managed system. Its interface is depicted in the gray box in Fig. 9.

Due to the one-to-one mappings between features and system components and parameters, the CA uses a basic set of four reconfiguration operations to instruct the CM, namely: *add* (feature selected), *remove* (feature deselected), *replace* (one feature selected and other deselected in an alternative relationship), and *update component* (feature representing a component parameter is selected or deselected). However these operations do not have an specific semantics. For instance, adding a component can mean to deploy

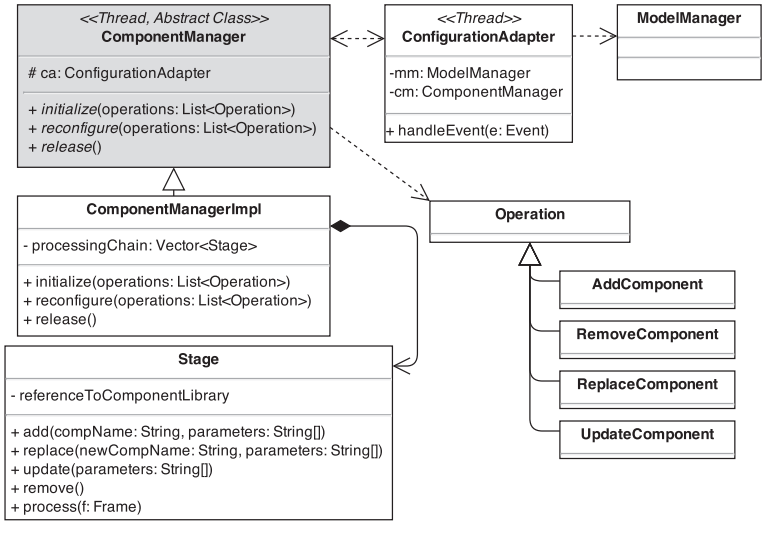


Fig. 9 Simplified class diagram of the platform

it on a particular slot of the system that requires an specific interface, or it can mean to connect a new component with an already deployed one, among others. The Component Manager and operation classes must be extended in order to apply the approach on a particular system and provide semantics to these operations.

For the sake of experimentation, we implemented a simple version of the CM that sets up the system as a linear arrangement of components to perform different computer vision tasks over a sequence of images. As shown in class diagram in Fig. 9, each running component is associated with a slot or *stage* in the processing chain. Thus components are added, removed and replaced from specific stages in the processing chain.

For implementing the CM, we evaluated some C++ component frameworks based on the OSGi component model (McAffer et al. 2010), such as the Service Oriented Framework (SOF)[3] and the CTK Plugin Framework (CTK)[4], but finally decided to use an ad-hoc solution loosely based on the OSGi model. Components were implemented as shared libraries (DLL files in the Microsoft Windows operating system) that can be loaded or unloaded dynamically upon request. These components are based on the computer vision components defined in (Rocha et al. 2011) that use the OpenCV libraries (Opencv project 2015). OpenCV is a collection of efficient algorithms and data types for implementing image processing and computer vision systems.

Each component manages its own internal state and resources, and performs a specific task within the processing chain, like image acquisition, segmentation, motion or face detection, etc. All components implement a common interface defined by specific C++ declarations, so that a given component can be replaced by another one with the same interface, if necessary. This interface consists of 4 functions: (i) *component initialization*, called when the component is added to a system assembly; (ii) *component release*, called when it is going to be removed or replaced by other; (iii) *component update*, called when component parameters are modified; and (iv) *execution* of the main task of the component. The latter basically involves performing some processing on an input image (video frame).

4 Evaluation: methods, results and discussion

In (Sanchez et al. 2013), we evaluated the optimization aspect of the approach by conducting some experiments in which we analyzed scalability, efficiency, and optimality of CSA using automatically generated scenarios. In this article, we complement this evaluation with concrete measures and analyze the accuracy of the additive and maximum metric functions for estimating two properties of interest in a video processing chain: *frame processing time* and *reconfiguration time*. Specifically, we performed four experiments with the additive and maximum metrics: two for frame processing time, and other two for reconfiguration time. The goal was to compare predicted against measured properties of the running system. To do so, we stated the following research questions:

Q1. What is the average accuracy of the additive metric for predicting frame processing time in sequential execution?
Q2. What is the average accuracy of the maximum metric for predicting frame processing time in parallel execution?
Q3. What is the average accuracy of the additive metrics for predicting reconfiguration time?

4.1 Application system and quality attribute Properties

For experimental purposes, we used the Component Manager implementation presented in Section 3.3 that controls a pipeline of software components for processing a sequence of video frames. The corresponding feature model is depicted in Fig. 10. This model only includes deployment and component aspects of the system, showing the software elements that conforms it: platform executable, OpenCV libraries, the stages that conforms the pipeline, and the components that can be assembled in each stage.

The CM implements two different execution modes for processing the pipeline of video frames: sequential and parallel execution. In the former, each video frame is processed sequentially in a single thread. It starts in the *Image Adquisition* stage, which captures images from a physical camera or a video file. Then the image is preprocessed in the *Filter* stage before performing some task of interest to the user in the *Task Dependent* stage. Finally, the resulting image is visualized in the *Visualization* stage. For parallel execution, the producer-consumer concurrency pattern is applied: each stage is performed by an independent physical thread and is connected to each other with a FIFO queue for storing the frames that are required by next stage. In this way, the required time for processing a single frame is bounded to the maximum response time of the involved components.

As it can be appreciated from the model, the second and third stages are optional, and the components *EdgeDetection* and *MotionDetection* are incompatible with the image output format of *ImageSmoothing*, what is expressed by cross-tree constraints. This implies a set of 14 valid configurations. Inputs and outputs of these components are OpenCV structures that represent images or video frames. As an example, output images of some of these components are shown in Fig. 11.

The configuration of components can be tuned by the *update* operation. For instance, the *CameraReader* source can be set to either a physical camera or a video file. As another example, the *FaceDetection* component, based on a Haar feature-based cascade classifier (Viola and Jones 2001), can be configured with different classifiers in order to detect frontal or profile faces, upper, lower part or full bodies, among others. For simplicity, in the model and therefore in the experiments, the variability of components is limited to their default configuration.

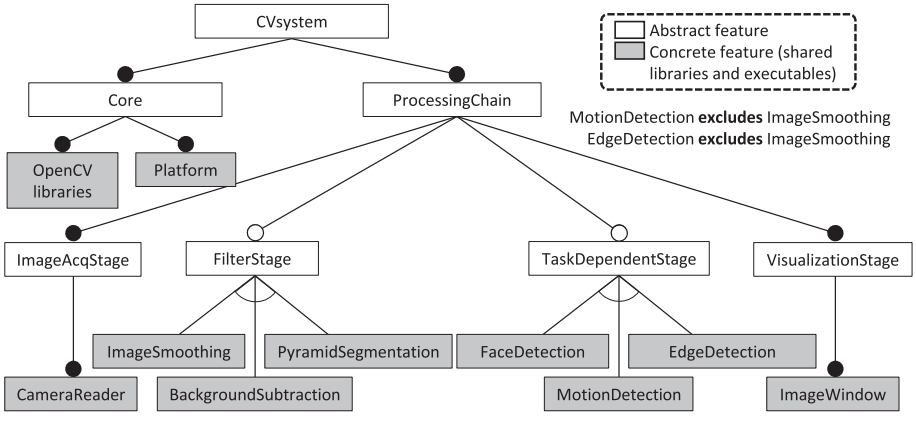


Fig. 10 Feature model of the vision system used as case study

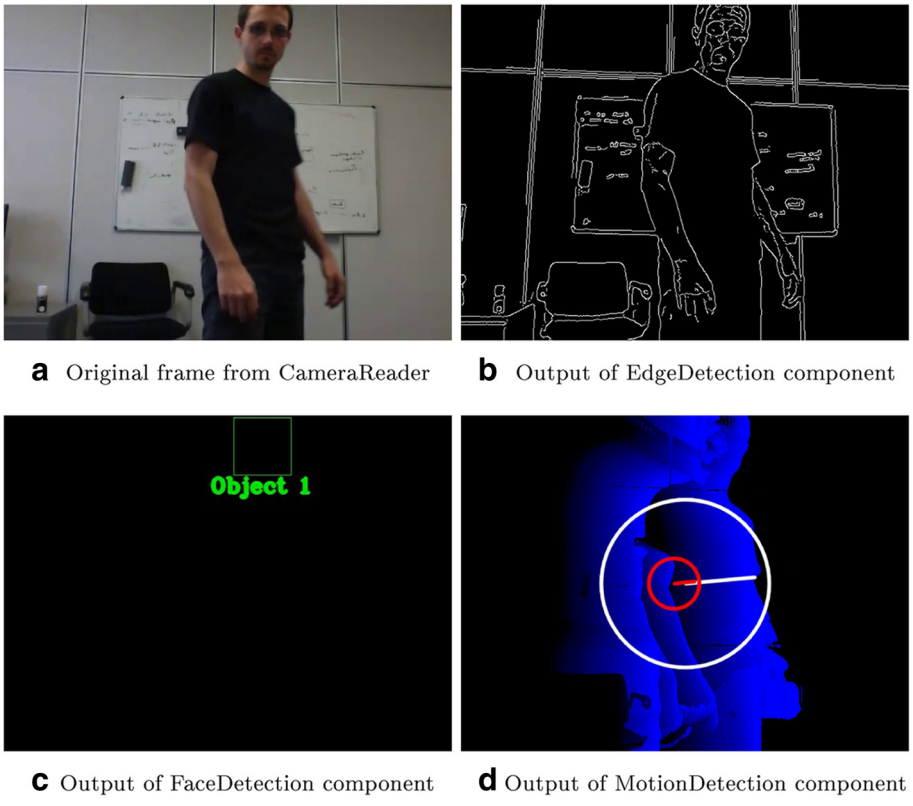


Fig. 11 Output examples of some vision components. **a** Original frame from CameraReader. **b** Output of EdgeDetection component. **c** Output of FaceDetection component. **d** Output of MotionDetection component

We chose *reconfiguration time (rtime)* and *frame processing time (fptime)* for a number of reasons. First, both are directly composable properties (as explained in Section 4): reconfiguration time is estimated as the sum of the *startup* and *shutdown time* of added and removed components respectively; while, due to the linear arrangement of components, frame processing time can be computed as the sum or maximum *response time* of all components involved in the processing chain, depending on whether it performs a sequential or parallel execution respectively. Second, performance measures in C++ programs are very accurate (in comparison to other languages, like Java), since they are compiled directly to machine code that is directly executed by the central processing unit. Besides, the allocation and de-allocation of resources, like memory and library modules, can be done manually (no automated garbage collection).

4.2 Experiment setup

The experiments were divided into four steps, namely: (i) measuring the properties of each component in isolation; (ii) using these property values and the additive/maximum metric function for estimating frame processing time of each configuration, and reconfiguration time of each transition between configurations; (iii) measuring the actual frame processing time and reconfiguration time of the same instances; and (iv) comparing estimated and actual property values to evaluate the accuracy of the metric functions and

thus answering questions Q1, Q2 and Q3. For reconfiguration time we performed two experiments with different models –one being an extension of the other– in order to observe how the metric accuracy is affected when the model is extended with additional features. All measurements were performed on the same PC with processor AMD FX-6300 CPU 3.5 GHz, 16 GB RAM, and running Windows 7 OS.

For the first step, we performed a set of repetitive measurements on every components so as to compute the following values: *startup time*, i.e., time required by the Component Manager to load the component; *shutdown time*, i.e., time required to unload the component; and *response time*, i.e., time required by the component to process a video frame. These values are summarized in Table 2. For startup and shutdown time, the components were loaded and unloaded 100 times, and then we computed their average values to validate their stability. These values involve the time required by the Component Manager for loading and unloading the library modules, allocating and de-allocating the components to the processing chain, and calling the *init* and *release* procedures of the component library. Recall that each component is responsible for managing its own set of variables and resources through the *init* and *release* procedures. Furthermore, the response time was calculated as the average of the time required by each component to process a set of approximately 1350 frames that composes a video sample of 48 seconds (28 frames per second), each frame with a resolution of 640x480 pixels and a color depth of 24 bits. It is a long video sequence from an office with a person coming in and out of scene.

In the following steps we evaluate the accuracy of the metrics for computing the quality attribute properties. For frame processing time, we considered the measurement of the 14 configurations. The estimated frame processing time value (step ii) is computed with the additive (sequential execution) and maximum (parallel execution) metrics over the feature model in Fig. 10 considering response time values in Table 2. Thus, the metric functions are defined as follows:

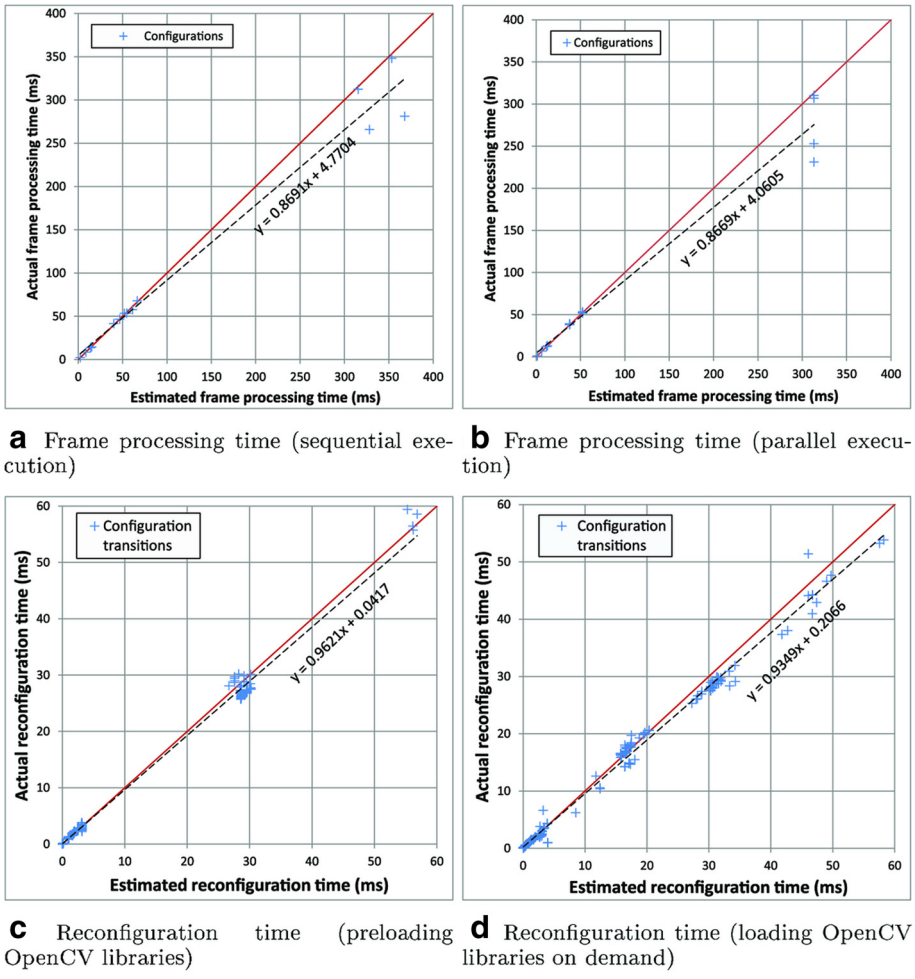
$$M^{+}_{fptime}(\mathbb{C}) = \sum_{f \in S} fptime_S\left(f\right) + \sum_{f \in D} fptime_D\left(f\right) \tag{2}$$

$$M^{M}_{fptime}(\mathbb{C}) = max\left(max_{f \in S} fptime_S\left(f\right), max_{f \in D} fptime_D\left(f\right)\right) \tag{3}$$

where $fptime_S\left(f\right)$ and $fptime_D\left(f\right)$ are 0, except for concrete features where $fptime_S\left(f\right) = responseTime\left(f\right)$, i.e., only selected concrete features contribute to the frame processing time value. The actual value (step iii) of each configuration was computed as the average time for processing each frame in the video sample. Finally, in Fig. 12(a) and (b) we present

Table 2 Performance properties of individual components

Component	Startup time (ms)	Shutdown time (ms)	Response time (ms)
CameraReader.dll	1.805 ± 0.069	0.047 ± 0.007	1.648 ± 0.328
ImageSmoothing.dll	0.819 ± 0.060	0.037 ± 0.004	12.972 ± 0.509
BackgroundSubtraction.dll	0.813 ± 0.041	0.037 ± 0.004	37.887 ± 3.758
PyramidSegmentation.dll	1.397 ± 0.107	0.055 ± 0.006	52.478 ± 1.902
FaceDetection.dll	27.833 ± 0.538	0.056 ± 0.004	313.260 ± 36.708
MotionDetection.dll	0.848 ± 0.052	0.038 ± 0.004	11.853 ± 0.770
EdgeDetection.dll	0.802 ± 0.036	0.034 ± 0.004	6.543 ± 0.354
ImageWindow.dll	4.692 ± 0.524	1.310 ± 0.195	0.536 ± 0.102
OpenCV libraries	19.543 ± 0.615	0.373 ± 0.019	NA



a Frame processing time (sequential execution) **b** Frame processing time (parallel execution)

c Reconfiguration time (preloading OpenCV libraries) **d** Reconfiguration time (loading OpenCV libraries on demand)

Fig. 12 Measured and estimated properties. Dotted black lines are regression curves and solid red lines represent the ideal trend. **a** Frame processing time (sequential execution). **b** Frame processing time (parallel execution). **c** Reconfiguration time (preloading OpenCV libraries). **d** Reconfiguration time (loading OpenCV libraries on demand)

two scatter plots to compare actual and estimated values of the frame processing time of these configurations in both execution modes (step iv).

For reconfiguration time, we have a total of 210 transitions among a set of 15 valid states (14 system configurations plus 1 for system shutdown). The estimated reconfiguration time value is also computed with an additive metric over the feature model in Fig. 10 enriched with startup and shutdown time values in Table 2. Thus, the metric function is defined as follows:

$$M^{+}_{rtime}(\mathbb{C}) = \sum_{f \in S} rtime_S(f) + \sum_{f \in D} rtime_D(f) \tag{4}$$

where $rtime_S(f)$ and $rtime_D(f)$ are 0, except for concrete features where $fperiod_S(f) = startupTime(f)$, if f was deselected in a previous configuration, and $fperiod_D(f) = shutdownTime(f)$, if f was selected in a previous configuration. As third step, each transition was performed and measured 30 times to get a statistical value of the actual reconfiguration time.

We performed two experiments predicting reconfiguration time with two different feature models: the one depicted in Fig. 10 and a refinement of it that consists of replacing the feature *OpenCVlibraries* with the sub-model presented in Fig. 13. This new model breaks down the OpenCV library component into a set of 10 libraries with dependencies among them. For instance, *opencv_video* requires functionality provided by *opencv_imgproc* which in turn depends on *opencv_core*. Components in the processing chain require different subsets of these libraries. For instance, *CameraReader* and *ImageWindow* require *opencv_highgui*, while MotionDetection requires *opencv_video*. These dependencies are expressed in terms of cross-tree constraints in the model.

With these two models, we defined two different ways for managing OpenCV libraries. In the former, OpenCV libraries are considered as a single component, therefore all libraries are loaded and unloaded together when the system is initialized and released respectively, although not all libraries are necessarily used. We called it the *preloaded* model. In Fig. 12(c) we present a scatter plot to displays actual and estimated values of the reconfiguration time of the 210 possible configuration transitions in this model.

In the refined model, OpenCV libraries are managed separately so that they are loaded just when a component requires it. Thus initializing and releasing the system imply smaller times than the *preloaded* model. We called it the *on demand* model. In Fig. 12(d) we present a scatter plot that display actual and estimated values of reconfiguration time considering this model. For estimating reconfiguration time, the startup and shutdown time of individual OpenCV libraries were measured, but they are not included in Table 2 for the sake of brevity.

4.3 Results and discussion

In Fig. 12 we clearly observe a linear correlation between measured and predicted values of frame processing time and reconfiguration time properties. Regression lines (dotted black curves) slightly deviate from the ideal trend (solid red curves) that represents the perfect match of estimated and actual property values. Besides linear correlation, we analyzed the *Accuracy* of the metrics. The accuracy is a ratio between 0 and 1 that is computed as $1 - FaultRate$, being the fault rate the average relative difference between estimated and actual property values:

$$FaultRate = \frac{1}{n}\sum_{i=1}^{n}\frac{|actual_i - estimated_i|}{actual_i} \tag{5}$$

In Table 3, we summarize the results of the four experiments.

For our research question Q1, the additive function is quite accurate (94.1 %) for predicting frame processing time on sequential execution. Regarding Q2, the maximum

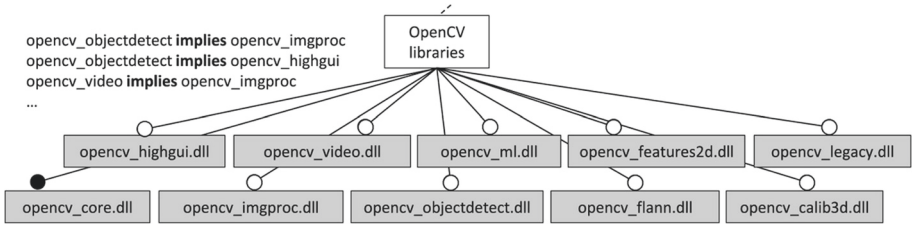


Fig. 13 Refinement of feature OpenCVlibraries

Table 3 Results of experiments

Experiment	Frame proc. time (sequential execution)	Frame proc. time (parallel execution)	Reconf. time (preloaded)	Reconf. time (on demand)
Slope regression	0.869	0.866	0.962	0.935
Y-intercept regression	4.770 ms	4.060 ms	0.042 ms	0.207 ms
Pearson coefficient	+ 0.976	+ 0.973	+ 0.995	+ 0.993
Fault rate	0.059	0.094	0.095	0.124
Average accuracy	0.941	0.906	0.905	0.876

metric is adequate for estimating frame processing time on parallel execution with an accuracy of 90.6 %. Finally, regarding Q3, the results show that the additive metric predicts reconfiguration time with an average accuracy of 90.5 % and 87.6 % on the original and refined model respectively.

We observe that the additive metrics is more accurate for frame processing time than for reconfiguration time, and for the latter, it is better in the simplest model. We believe this is due to the complexity of the underlying experiment. For the former, we only evaluated 14 scenarios (valid configurations) involving the response time of 8 components. Besides, processing video frames requires more CPU (and GPU) operations, whose execution time measurement is more accurate, than input/output operations. For reconfiguration time, we evaluated 210 scenarios (transitions among valid configurations) and loading shared libraries required additional input/output operations that affected the accuracy of measurements. Particularly, the experiment with the refined model is more complex since it manages the 10 OpenCV libraries independently, when compared to the simplified model that loads and unloads all libraries at once.

From the linear regression analysis, we observe that the Pearson correlation coefficient is very close to 1. For both frame processing and reconfiguration time, the estimated values are slightly higher than the measured ones on average. At first sight, the opposite should be expected due to the overheads introduced by the Component Manager for controlling component execution and reconfiguring the processing chain. However, these overheads are negligible with respect to component response, startup and shutdown time. The CameraReader component creates one image object per frame, while ImageWindow destroys them, so, for the sequential processing of video frames, the overhead introduced by the CM only involves invoking each component in order with a reference to the image frame objects. For parallel processing, we expected an overhead due to the synchronization of threads on the FIFO queues. On the other hand, reconfiguration requires to iterate and parse a list of operations instructed by the Configuration Adapter module. These overhead times are not considered in the experiment model, since they are orders of magnitude lower than performance properties values in Table 2.

The reason for these deviations are feature interactions (Siegmund et al. 2013), i.e., the selection of one feature influences non-functional properties of other features. The impact of these interactions is better observed when estimating frame processing time. The response time of components in the task dependent stage varies depending on the image output of the component in the filter stage. These components apply some image preprocessing filters that change image characteristics and so reducing the response time of the following components. For instance, the configurations using FaceDetection with and without ImageSmoothing have an average processing time of 252.78 ms and 306.81

ms respectively in parallel execution, what shows that smoothing reduces face detection response time considerably.

4.4 Threats to validity

The experiments above had some drawbacks. The application system is composed of a total of 19 components, namely the platform executable, 8 component libraries, and 10 OpenCV libraries. This leads to 14 possible configurations. Although our experiments represent a realistic vision system, we might imagine a case study with hundred of components and parameters in order to provide a deeper statistical analysis of the accuracy of metric functions on feature models, and how it is affected when the model scale in terms of number of features and configurations.

With the additive and maximum metric functions we demonstrate predictability of performance properties such as reconfiguration time and frame processing time on sequential and parallel execution contexts. However, we leave aside the remaining set of metrics for other non-functional properties.

These metric functions do not contemplate the interaction between features. We rely on aggregate functions where features contribute to configuration property values depending only on their own state (selected or deselected). The problem of prediction on feature models when the influence of features on configuration properties depends on the states of other features has been studied in (Siegmund et al. 2013). In this regard, there is a clear need to extend the metrics set to consider general objective functions in order to compute complex properties in other software architectures where feature interaction influences are more complex.

Finally, the properties values depicted in Table 2 are hardware-dependent performance properties because they rely heavily on the hardware characteristics of the execution environment. Besides, the response time of each component depends on the characteristic of the input video, such as frame resolution, color depth, scene elements, etc. Therefore, values in Table 2 are not suitable for execution environments in general. However, it is important to emphasize that the goal of our approach is not to provide accurate predictions of non-functional properties, but rather to provide a "good enough" mechanism for comparing configurations based on quality criteria. Our experiments were performed with the purpose of showing how a simple metric function can be used for estimating performance properties of a system configuration represented by a feature model.

5 Related work

Related work is separated 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 predictability of quality attributes on component-based software systems and software product lines. Related work on algorithms for feature model optimization was previously discussed in (Sanchez et al. 2013).

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 (da Silva et al. 2012; Garlan et al. 2009) and DSPL (Morin et al. 2009). In (Garlan et al. 2009), 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 (da Silva et al. 2012), 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 (Garlan et al. 2009). In (Morin et al. 2009), 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.

Besides system and context representation, these works differ mainly in the reasoning process for configuration selection: (Garlan et al. 2009) and (da Silva et al. 2012) 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 (Morin et al. 2009) 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 (Bartholdt et al. 2009; Siegmund et al. 2008, 2012). These works propose 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. 2009 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 (Siegmund et al. 2012), non-functional properties can be specified following any of the three classes mentioned in Section 2.3.1. Siegmund et al. 2008 propose a similar categorization to select an appropriate measurement technique for these properties, and provide an optimization process based on Constraint Satisfaction Problem (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 runtime adaptation where configuration selection algorithms must meet correctness, completeness, and efficiency requirements.

Finally, regarding the third category, predictability of quality attributes was already addressed for both component-based systems and software product lines. In (Crnkovic et al. 2005), Crnkovic et al. analyze the relation between the quality attributes of components and those of their compositions. They provide a classification of properties according to the principles applied in deriving system properties from the properties of the involved components. This classification is illustrated with several examples of runtime properties, some of which are directly derived from component properties with some kind of aggregate function. Siegmund et al. address predictability of properties on software product lines (Siegmund et al. 2013). They highlight the fact that a prediction cannot be exact if the interaction among features is not taken into account, because the selection (or deselection) of one feature may influence non-functional properties of other features. Thus, they propose an approach for predicting a configuration property from a small set of generated and measured configurations. By comparing measurements,

they approximate the influence of each feature on the property in question instead of measuring feature properties in isolation.

The metric functions and properties that we evaluate on feature models conform to the classification proposed in (Crnkovic et al. 2005). However we do not consider more expressive functions to deal with the feature interaction concern addressed in (Siegmund et al. 2013) that affects predictability on feature models. This is an important concern that might cause unpredictable anomalies when estimating non-functional properties in software product lines.

6 Conclusions

This article presents an approach to improve the dynamic adaptation of component-based systems by dealing with runtime quality attributes on feature models. The main contribution is the provision of a simple but still expressive framework to specify, quantitatively evaluate and balance multiple quality attributes to reach a better system configuration. This is especially important for applications that exhibit high variability at the specification and implementation levels, leading to a large number of configurations.

Our approach works integrated into a component-based platform that provides mechanisms for event handling and self-adaptation. We used this platform to implement a computer vision system as a linear arrangement of components following the pipe-and-filter architectural style. However, we consider that it can be tailored to different component assemblies by providing a more sophisticated Component Manager implementation. In this respect, we plan to further analyze approach limitations and extend its applicability to other architectural styles. This would probably require to analyze more complex mappings among features and software elements. In addition, we should note that the approach requires an extra engineering effort at development time for designing the model, mapping it to system elements, and specifying attribute values, metrics and weights for the optimization step. Along this line, we expect to further develop the tooling support for system developers.

The configuration selection is supported by a heuristic search algorithm that ensures correctness and completeness while addressing time efficiency and scalability for large scale instances of the problem. With a different algorithm strategy, even optimality can be achieved at expense of a lower performance. The use of admissible heuristics on feature models guarantees optimal solutions, but a limitation is that the objective function is restricted to a linear combination of the basic aggregate functions (as depicted in Table 1). An interesting improvement is then to provide support for general objective functions to cover more complex properties. Since several algorithms have been proposed for dealing with the feature model optimization problem, a comparative evaluation of our current algorithm against other alternatives is needed in order to compare their efficiency and optimality.

The aggregate functions that we have considered are appropriate for several properties that are directly derived from properties of individual components. For instance, we showed evidence that the additive and maximum functions are reasonably accurate for measuring performance properties on a realistic component-based computer vision system. Anyway, more evaluations remain to be done to study the effectiveness of these functions and the influence of feature interactions on larger models. Furthermore, other metrics and runtime properties need to be assessed.

On the application side, we plan to adapt our approach to the domain of service-oriented computing for assisting the development, 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 such as security, availability, scalability and battery consumption can be more difficult to manage than in centralized systems.

7 Consent

Written informed consent was obtained from the person involved in the video sample used for experimental purposes.

Endnotes

[1]https://github.com/EmilianoSanchez/FeatureModelOptimization

[2]For simplicity, terms such as quality attribute, quality attribute property, and non-functional property are treated as synonymous.

[3]http://sof.tiddlyspot.com/

[4]http://www.commontk.org/index.php/Documentation/Plugin_Framework

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
All authors contributed equally to different parts of the approach design, platform implementation and experiment execution. The manuscript was prepared by LES. Corrections and reviews were made by SM, JPR, JADP and AZ. All authors read and approved the final manuscript.

Acknowledgements
This work was partially supported by ANPCyT (Argentina) through PICT Project 2013 No. 0464, and also by CONICET (Argentina) through PIP Project No. 112-201101-00078.

Author details
[1]ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil, Argentina. [2]INRIA Sophia Antipolis Méditerrannée, Route des Lucioles 06902, Sophia Antipolis, France.

Received: 5 December 2014 Accepted: 27 May 2015
Published online: 23 June 2015

References

Acher M, Collet P, Lahire P, France RB (2013) Familiar: A domain-specific language for large scale management of feature models. Sci Comput Programm 78(6):657–681

Apel S, Kästner C (2009) An overview of feature-oriented software development. J Object Technol (JOT) 8(5):49–84

Bartholdt J, Medak M, Oberhauser R (2009) Integrating quality modeling with feature modeling in software product lines. In: Boness K, Fernandes JM, Hall JG, Machado RJ, Oberhauser R (eds). Fourth International Conference on Software Engineering Advances, 2009. ICSEA '09. IEEE Computer Society, Washington DC, USA. pp 365–370. http://dblp.uni-trier.de/db/conf/icsea/icsea2009.html#BartholdtMO09

Bass L (2003) Software Architecture in Practice. 2nd edn. Addison-Wesley, Boston, USA

Batory D (2005) Feature models, grammars, and propositional formulas. In: Proceedings of the 9th International Conference on Software Product Lines, SPLC'05. Springer, Rennes, France. pp 7–20

Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: A literature review. Inform Syst 35(6):615–636

Benavides D, Trinidad P, Ruiz-Cortés A (2005) Automated reasoning on feature models. In: Pastor O, Falcão e Cunha J (eds). Advanced Information Systems Engineering. Lecture Notes in Computer Science. Springer, Porto, Portugal Vol. 3520. pp 491–503

Crnkovic I, Larsson M, Preiss O (2005) Concerning predictability in dependable component-based systems: Classification of quality attributes. In: de Lemos R, Gacek C, Romanovsky A (eds). Architecting Dependable Systems III. Lecture Notes in Computer Science. Springer, Heidelberg, Germany Vol. 3549. pp 257–278

Silva da DC, Lopes AB, Pinto FAP, Leite JC (2012) Selecting architecture configurations in self- adaptive systems using qos criteria. In: Sixth Brazilian Symposium on Software Components Architectures and Reuse (SBCARS), 2012. IEEE Computer Society, Washington DC, USA. pp 71–80. http://dblp.uni-trier.de/db/conf/sbcars/sbcars2012.html#SilvaLPL12

Garlan D, Schmerl B, Cheng SW (2009) Software architecture-based self-adaptation. In: Zhang Y, Yang LT, Denko MK (eds). Autonomic Computing and Networking. Springer, New York, USA. pp 31–55

Hallsteinsen S, Hinchey M, Park S, Schmid K (2008) Dynamic software product lines. Computer 41(4):93–95
IBM (2003) An architectural blueprint for autonomic computing. Technical report, IBM
Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute
Marler RT, Arora J (2010) The weighted sum method for multi-objective optimization: new insights. Struct Multidisciplinary Optimization 41(6):853–862
McAffer J, VanderLei P, Archer S (2010) OSGi and Equinox: Creating Highly Modular Java Systems. Addison-Wesley Professional, Boston, USA
Moisan S, Rigault JP, Acher M, Collet P, Lahire P (2011) Run time adaptation of video-surveillance systems: A software modeling approach. In: Crowley J, Draper B, Thonnat M (eds). Computer Vision Systems. Lecture Notes in Computer Science. Springer, Sophia Antipolis, France Vol. 6962. pp 203–212
Morin B, Barais O, Jezequel J, Fleurey F, Solberg A (2009) Models@ run.time to support dynamic adaptation. Computer 42(10):44–51
Oliveira N, Barbosa LS (2014) A Self-adaptation Strategy for Service-based Architectures. In: VIII Brazilian Symposium on Software Components, Architectures and Reuse. SBCARS'2014. IEEE, Maceió, Alagoas Vol. 2. pp 44–53
Opencv project (2015). http://opencv.org/
Oreizy P, Gorlick MM, Taylor RN, Heimhigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL (1999) An architecture-based approach to self-adaptive software. Intell Syst Appl IEEE 14(3):54–62
Pearl J (1984) Heuristics - Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley series in artificial intelligence, pp –1382. Addison-Wesley
Rocha LM, Moisan S, Rigault JP, Sagar S (2011) Towards lightweight dynamic adaptation a framework and its evaluation. In: 12th Argentine Symposium on Software Engineering (ASSE 2011), JAII0 2011, Córdoba, Argentina
Rosenberg F, Celikovic P, Michlmayr A, Leitner P, Dustdar S (2009) An end-to-end approach for QoS-aware service composition. In: Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International. pp 151–160
Sanchez LE, Moisan S, Rigault JP (2013) Metrics on feature models to optimize configuration adaptation at run time. In: Paige RF, Harman M, Williams JR (eds). 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE). IEEE, Cambridge, UK. pp 39–44. http://dblp.uni-trier.de/db/conf/icse/cmsbse2013.html#SanchezMR13
Sanchez LE, Diaz-Pace JA, Zunino A, Moisan S, Rigault JP (2014) An approach for managing quality attributes at runtime using feature models. In: VIII Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software SBCARS 2014. IEEE, Cambridge, UK. pp 11–20. http://dblp.uni-trier.de/db/conf/sbcars/sbcars2014.html#SanchezPZMR14
Schobbens PY, Heymans P, Trigaux JC, Bontemps Y (2007) Generic semantics of feature diagrams. Comput Netw 51(2):456–479. Feature Interaction
Siegmund N, Rosenmuller M, Kuhlemann M, Kastner C, Saake G (2008) Measuring non-functional properties in software product line for product derivation. In: Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific. IEEE Computer Society, Washington DC, USA. pp 187–194. http://dblp.uni-trier.de/db/conf/apsec/apsec2008.html#SiegmundRKKS08
Siegmund N, Rosenmüller M, Kuhlemann M, Kästner C, Apel S, Saake G (2012) Spl conqueror: Toward optimization of non-functional properties in software product lines. Softw Quality J 20:487–517
Siegmund N, Rosenmüller M, Kästner C, Giarrusso PG, Apel S, Kolesnikov SS (2013) Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Inform Softw Technol 55(3):491–507. Special Issue on Software Reuse and Product Lines Special Issue on Software Reuse and Product Lines
Viola P, Jones M (2001) Rapid object detection using a boosted cascade of simple features. In: Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition CVPR 2001. IEEE Computer Society, Washington DC, USA Vol. 1. pp 511–5181. http://dblp.uni-trier.de/db/conf/cvpr/cvpr2001-1.html#ViolaJ01