Towards Product Configuration Taking into Account Quality Concerns

Karina Villela
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserlautern
Germany
karina.villela@iese.fraunhofer.de

Taslim Arif
Fraunhofer IESE
Fraunhofer-Platz 1
67663 Kaiserlautern
Germany
taslim.arif@iese.fraunhofer.de

Damiano Zanardini
Technical University of Madrid
Campus de Montegancedo
28660 Boadilla del Monte
Spain
damiano@fi.upm.es

ABSTRACT

The configuration of concrete products from a product line infrastructure is the process of resolving the variability captured in the product line according to a company's market strategy or specific customer's requirements. Several aspects influence the selection of features for a concrete product, such as dependencies and constraints between features, the different stakeholders involved in the process, the desired degree of quality, and cost constraints. This paper presents the vision of a configurator that will focus on providing indicators of security and performance for features and empowering its users to interactively observe the effect of the selected set of features on these two quality characteristics. We propose the use of reusable expert knowledge and static analysis for obtaining the indicators of security and performance, respectively. The two main issues to be investigated are: (1) to which degree the configuration process should be automated; and (2) how exactly to obtain indicators of security and performance for features that can be used to predict the security and performance of whole configurations.

Categories and Subject Descriptors

D.2.8, D.2.13 [**Software Engineering**]: Metrics, Reusable Software—*product metrics, domain engineering*

General Terms

Design, Performance, Security

Keywords

Product Line Engineering, Feature Models, Product Configuration, Quality Concerns, Static Analysis

1. INTRODUCTION

Configuring concrete products from a product line infrastructure is the process of resolving the variability captured in the product line, based on a company's market strategy or requirements from specific customers. *Feature models* [16] have been the main approach for capturing variability in product lines, so the configuration process usually consists of selecting those features that are applicable to the product and assembling the (partial) product from the product line assets. Several aspects influence the selection of features for a concrete product, such as dependencies and constraints among features, the different stakeholders involved dealing with external and internal features, the desired degree of *quality*, and cost constraints. As real-world feature models normally have hundreds or even thousands of features, the selection of a correct and appropriate set of features can be a very cumbersome task.

Many approaches have been proposed to support *feature selection* in this context. One of the topics that have been extensively investigated is the automated selection of features based on the dependencies and constraints between features, as well as constraints on the cost [23, 20]. Such techniques map the problem of feature selection into a general computational problem that can be decided by a standard solver, such as a Constraint Satisfaction Problem (CSP) solver or a Hierarchical Task Network (HTN) planner.

The participation of different stakeholders (with specific interests) in the product configuration process has also been addressed. Hubaux et al. [12] defined a configuration process in which stakeholders are identified and views are specified based on the specific features that are of interest to them. White et al. [24] proposed a constraint-based diagnosis approach that is capable of suggesting corrections to an invalid feature model configuration from a specific point of view, based on the observation that the source of error in a feature model configuration may vary depending on the perspective used to debug it.

In order to allow product configuration based on quality characteristics, it is necessary to: (1) support the modeling of quality variability; (2) capture the mutual influence between functional and non-functional requirements; and (3) provide a way to specify to which degree a given quality characteristic should be present in a certain product. The automatic configurator proposed by Soltani et al. [20] deals with these points. However, it is not possible to know exactly how quality concerns are treated: they are modeled as pre-conditions, in the same way as required features, and the only provided screenshot showing the output is not explained. Consequently, it is difficult to draw any conclusions on the effectiveness of the technique.

SPLC - Vol. II, September 02 - 07 2012, Salvador, Brazil

This paper introduces our vision of a configurator to be developed in the context of the HATS project [5]. The HATS framework includes five languages (μTVL, DML, CL, SPL, and ABS) and several tools and techniques addressing variability, evolvability, and trustworthiness. Variability is defined using μTVL feature models, and implemented by means of so-called *delta models* in `DML`. A product line is defined in a `CL` file that connects the feature model with the respective delta models. The framework allows specifying which features are to be included in a concrete product (in an `SPL` file), and the validity of such a configuration can be checked. `ABS` code is generated for valid configurations.

Concerning previous work in the area of product configuration, we will focus our contribution on providing indicators of *security* and *performance* for features and clearly determining how those should be treated during the configuration process. Moreover, our configurator should empower its users to interactively observe the effect of selecting or deselecting features on the quality characteristics of interest (e.g., choosing a particular feature could imply some security flaw or a decrease in performance).

This paper is organized as follows: Section 2 presents previous work regarding quality variability modeling, product configuration, and static analysis. Section 3 presents our vision of a product configurator for the HATS framework, which addresses performance and security as quality concerns. This section also provides the questions that we have been investigating in order to be able to completely define our approach. Section 4 concludes this paper.

2. RELEVANT PREVIOUS WORK

This section presents previous work that was taken into account in defining the concept of the configurator and which will also be relevant for its detailed design. See Section 3.4 for further information on how previous work relates to the proposed solution.

2.1 Quality Variability Modeling

Concrete products of a software product line may vary with regard to quality characteristics, particularly the ones that are relevant to end-users. This paper is especially concerned with security and performance. According to Exteberria and Sagardui [9], quality variability can be of different types:

- Variability in the relevance of quality characteristics: for example, performance might be considered as important for one product, while security might be regarded as the main issue for another product.
- Variability in the degree of presence of a quality characteristic: for example, one product might require high security, whereas others might require a medium or even low degree of security.
- Indirect variation: functional features may impact differently on different quality characteristics.

Exteberria and Sagardui [9] also identified several requirements for quality variability modeling techniques: (1) allow automatic analysis and reasoning, (2) provide concrete quality characterization, (3) capture optionality, (4) capture the impact of a functional feature on quality, (5) capture the impact of a set of functional features on quality, and (6) capture the interdependencies among quality characteristics.

Several quality variability modeling techniques have been presented in the literature. The major quality modeling techniques are [9]:

Goal-based Model: It proposes two sub-models: one model represents the functional variability and the other model (soft-goal-model) represents the quality characteristics the system should achieve. The correlation between the functional and the soft-goal-models captures the impact of functional goals on soft-goals and is usually represented as ++, +, ?, -, --. These qualitative labels are converted into quantitative values, and priorities are also set among the different soft-goals to do some automatic reasoning on the model [10].

Bayesian Belief Network: This captures functional variability in a feature model, a Bayesian Belief Network is used to capture the impact of functional variants on quality characteristics. This means directed edges are annotated with a number that represents the domain expert's belief regarding the impact of functional variants on the quality characteristics [25].

Definition Hierarchy: A hierarchical logical AND tree is used to model variability. The topmost nodes are the quality characteristics and the other nodes are design decisions. Any edge in the tree represents that this quality characteristic is partially fulfilled by this design decision. Each node also gets a priority to represent the importance of that node in supporting its parent [17].

COVAMOF: This captures functional variability through variation points and quality variability through dependencies. Three types of associations are used to relate dependencies to variation points: (1) abstract, which indicates there is an influence, but there is no information on how, (2) directional, where the influence is not fully known, but there is some information on how the value depends on its binding, (3) logical, where the influence is fully known [19].

Feature-Softgoal Interdependency Graph (F-SIG): It uses a feature model to represent functional variability and softgoal interdependency to represent quality characteristics. Two types of impacts (explicit and implicit) of features on quality characteristics can be modeled and the degree of influence (for example, ++, +, ?, -, --) can be specified [15].

Extended Feature Model: It extends feature models with attributes of features and relationships among those attributes, which are used to model quality concerns [4].

2.2 Product Configuration

According to White et al. [23], developers must find a selection of the features from the feature model that satisfy the requirements of a concrete product and adhere to the rules of the feature model. This configuration process involves reasoning over a complex set of constraints to meet a final goal. Interactive configuration support uses constraints to propagate configuration choices made by the user [8], whereas automatic configuration support provides a set of configurations that satisfy the rules of the feature model and the user's requirements and constraints.

Hubaux [11] classifies the techniques for feature-based configuration into five groups.

Unguided configuration: The feature model is used without following a well-defined approach. Usually the features are merely selected in a top-down fashion.

Staged configuration: The product configuration is performed in stages, each stage eliminating configuration choices in the feature model. Different dimensions can be associated with the configuration stages, for instance the times of the product life or the roles of the parties performing the configuration [6].

Multi-level staged configuration: This extends the staged configuration by adding configuration levels, where each level is represented by a feature model linked to the other level's feature model through inter-level links [7]. This provides modularization and therefore enhances scalability.

Probabilistic configuration: This augments traditional feature models with conditional probabilities regarding the selection of features and legal joint probability distributions (JPD) [8]. JPD assign a probability to each possible configuration of a software product line.

Dynamic configuration: This considers the evolution of the product configuration at runtime [18]. In contrast to static configuration, both context evolution and stakeholder actions entail automated reconfigurations of the running product.

Furthermore, White et al. [23] present a formal model of multi-step configuration that allows making continuous pro-gress on the development of a product according to a change constraint, such as the maximum development budget per year. The output is a path of configurations (from the starting configuration to the desired final configuration) that meet the multi-step constraints. Their technique, which translates the feature selection problem into CSP, can also be used for single-step or staged configuration.

The same research group [22] provides a polynomial-time approximation algorithm, called Filtered Cartesian Flattening, for selecting a highly optimal set of features that adheres to a set of system resource constraints. In Filtered Cartesian Flattening, the problem of feature selection with resource constraints is represented as an approximate Multi-dimensional Multi-choice Knapsack Problem (MMKP), which can be solved by existing MMKP approximation algorithms. Compared with exact techniques such as CSP-based techniques, Filtered Cartesian Flattening allows the evaluation of more configurations within a shorter time frame and generates solutions that are at least 90% optimal.

Soltani et al. [20] propose the extension of feature models with annotations reflecting different business concerns of the stakeholders. Examples of business concerns are "high security", "high customer satisfaction", "high performance" and "potential for high international sales". They make use of Hierarchical Task Network (HTN) planning to select the maximal sets of features which satisfy the stakeholders' requirements and business concerns, and have a total cost lower than or equal to the expected cost.

While these three related works address automatic configuration, Czarnecki et al. [8] support interactive configuration based on probabilistic feature models. In their technique, probabilistic propositional formulas are used to express both hard and soft constraints. Such formulas have well-defined semantics denoting a set of joint probability distribution over features. The authors also show how probabilistic feature models can be mined from a given set of product configurations. On the one hand, the main advantage of this technique is the capability of expressing preference among the legal configurations. On the other hand, either a representative sample set of configurations to apply feature model mining or a domain expert is required.

Abbasi et al. [1] also support interactive configuration. Taking into consideration that the configuration process involves several stakeholders and might not be organized in a linear manner, they extend an existing feature-based configurator with multi-view support and integrate it with a workflow management tool. Views let stakeholders concentrate on those parts of the feature model that are relevant to them and the workflow management tool drives the configuration of these views.

2.3 Static Resource-Usage Analysis

This section gives a brief introduction to resource-usage[1] analysis, a static analysis technique that, given a program, statically infers limits to the resource consumption of the program at runtime. An existing implementation of such a technique will be used in the context of the proposed configurator in order to estimate the quality of a product with respect to a given metric of performance efficiency [14].

2.3.1 Introduction to Resource-Usage Analysis

The goal of automatic static *resource-usage analysis* [21] is to automatically infer *bounds* on the resource consumption of a program without executing it (i.e., *statically*). Such bounds come as symbolic expressions representing functions of the *size* of the program input data.

Given a program or a code fragment, a *resource-usage analyzer* aims at finding such bounds. For example, if information on the maximum number of execution steps is desired, the analyzer could get an upper bound like $size(l) * k$ for a program which traverses a linked list, where $size(l)$ is the size of the list (typically, its length) and k is some constant. This means that traversing the list only requires a number of steps that is proportional to its length.

COSTABS [2] is a termination and resource-usage analyzer for ABS [5]. It infers upper bounds for several metrics, taking into account advanced features of the language such as concurrency and distribution. With regard to concurrency, COSTABS copes with specific problems like the possibility that the data a task is operating on are modified by other tasks while the first one is suspended. In this sense, it is crucial to identify when task interleaving causes a loss of precision and when it does not. As for distribution, COSTABS is able to separate the resource usage of a program into *resource centers*[2], which represent the locations where the different parts of the code are executed. COSTABS builds *recurrence relations*, which describe how the program behaves

[1] "Cost analysis" is another name for this analysis. However, "resource-usage analysis" will be used here in order to avoid confusion with other uses of the term "cost".

[2] The original name for this notion is "cost center".

w.r.t. the desired notion of resource usage. Afterwards, such relations are solved by the PUBS [3] solver which, when possible, gives a solution in terms of an upper bound.

2.3.2 Metrics

COSTABS offers a number of metrics to be analyzed as *resource models*[3]. A resource model is a function that assigns to every program statement a non-negative number: the amount of resources it uses.

For example, the resource model "number of steps" simply assigns a resource usage 1 to every language construct that is executed in one step according to the selected semantics. In the case of ABS [5], a step corresponds to any instruction belonging to either the functional or the imperative part of the language (e.g., variable assignment).

Another resource model is "memory consumption", which assigns a positive resource usage to instructions allocating memory (e.g., object creation, where the assigned number is the amount of memory allocated when creating the new object), and 0 to all the others.

In general, upper bounds come as symbolic expressions that depend on the program parameters, so that the *asymptotic* resource usage can be computed (i.e., how the number of instructions or the memory consumption vary as the size of the arguments tends towards ∞).

2.3.3 An example

Consider the following ABS fragment:

```
Unit reqFile(Server sId, Filename fId) {
  File file = Nil;   Packet pack = "";   Int lth = 0;
  Fut<Int> l1;       Fut<Packet> l2;

  l1 = sId!getLength(fId);
  await l1?;    lth = l1.get;
  while (lth > 0) {
    lth = lth - 1;
    l2 = sId!getPack(fId, lth);
    [db <= max(db)]
    await l2?;
    pack = l2.get;
    file = Cons(pack, file);
  }
  db!storeFile(fId, file);
}
```

The main part of this method is the loop on the integer value `lth`, which decreases by 1 in every iteration. Suppose that the number of steps is chosen as the resource model and that the separation between resource centers is considered, then the result of COSTABS is as follows (some minor post-formatting of the output has been applied):

```
2 within resource center 'DataBaseImpl'
1+2*nat(-fId/2+dbT/2-2) within resource center 'Node'
```

where `nat(X)` is `X` if `X` is non-negative, and 0 otherwise. The output from COSTABS means that two steps (first upper bound) are performed by objects of the class `DataBaseImpl`, concretely in the methods `getLength` and `storeFile`. The rest is performed in the class `Node`, where the method `reqFile` is declared. The second upper bound can be rewritten as `dbT − fId`, which means that the number of steps is proportional to the difference between (the size of) `dbT` (which is a field of `DataBaseImpl`) and the input parameter `fId`.

3. OUR VISION

[3]The original name for this notion is "cost model".

3.1 General Description

The building blocks of our proposal are: (1) semi-automatic selection of features based on the feature model information and the user's requirements, (2) static analysis of the feature implementations in order to derive internal performance metrics, (3) reusable expert knowledge on security, and (4) software visualization.

Our configurator will support the configuration process of a concrete product through the following steps:

1. The configurator uses information from the feature model about mandatory features to automatically select those for the current product configuration.
2. The configurator asks the user to inform the required features. Required features are those which the user is confident about their inclusion in the product. If they do not infringe on any constraint in the feature model, the configurator will not propose deselecting them in any of the provided solutions.
3. The user visualizes the automatically selected mandatory features in a graphic representation and indicates the required features by selecting them.
4. The configurator checks the correctness (not the completeness) of the set of features that has been selected so far. If the selected set of features triggers *requires* and *excludes* relationships, the configurator will either select features or make them unavailable. If there is an error, the configurator will at least indicate the constraints that have been infringed.
5. If necessary, the user corrects his/her selection of required features.
6. The configurator helps the user to provide cost constraints and/or to inform qualifiers for the quality concerns of interest.
7. The user provides information about the cost constraints and/or quality concerns of interest. For the quality concerns, the user also provides their respective weight.
8. The configurator provides a set of configurations that include the required features and obey the cost constraints, while at the same time trying to achieve the desired degree of quality. For each proposed configuration, the configurator provides its number of features and its cost, and shows the estimated degree of security and performance in bar graphics.
9. The user chooses the most suitable configuration out of the recommended configurations.
10. The configurator updates the estimated degree of quality for the chosen configuration, taking into account the current combination of features.
11. The user decides to derive the concrete product.

Figure 1 shows the prototype of the configurator's screen for step 8. On the left side, the user follows the steps of the configuration process that require information or action from him/her. The information/action is required on the right side. In this case, the user visualizes the automatically selected mandatory features (in green), the required features

(in blue), and the selection of features suggested by the configurator (in yellow). On the top, the user is informed that this is configuration number 5 and he can navigate through the complete set of valid configurations that adhere to the user's requirements. The bars on the right side illustrate the estimated degree of security and performance of the current configuration and also illustrate its cost position regarding the cost constraints.

3.2 Solution Components

In order to realize the above vision, we have devised a component model for the configurator (Figure 2). The main components are described below.

- `UserInteraction`: The user of the configurator interacts with the latter through this component, which provides interfaces to all sorts of user interaction. The most important interactions are : loading ABS model, selecting required features, specifying quality concerns and cost constraints, and choosing a suitable configuration out of the recommended configurations.
- `Visualization`: This component displays the feature model, the intermediate configurations (with mandatory features and features selected by the user), and all recommended configurations (with quality ratings). It also provides an interface `UserAction` that can communicate with the user interaction in the visualization.
- `ViewManagement`: This component generates the initial configuration with mandatory features. It is also responsible for generating the intermediate configuration by applying the feature model dependencies and constraints to the features selected by the user. For example, further features are selected as a consequence of *requires* and features are made unavailable as a consequence of *excludes*. Moreover, it communicates the current selection of features to other components.
- `QA-CostManagement`: This component is responsible for managing quality concerns and cost constraints. It provides the interfaces for storing and retrieving such concerns and constraints. The `UserInteraction` component requires the first interface to load the concerns and constraints provided by the user, and the `Solver` component requires the second interface to get them.
- `ABSModelManagement`: This component provides the interface for loading, updating and retrieving the ABS model, which means the μTVL, CL, Delta, and Core ABS files.
- `Solver`: This component provides the interface for converting the problem comprising the features selected by the user, the quality concerns, the cost constraints, and the annotated feature model into a CSP problem. It is capable of using a standard CSP solver to determine the most suitable configurations.
- μTVL`Annotation`: This component provides the interface for annotating the feature model with different attributes, namely performance, security, and cost.
- `PerformanceAnnotation`: This component provides the interface for annotating the feature model with performance indicators. Since the performance of the features cannot be precisely analyzed in isolation, the annotation obtained cannot be considered as totally precise[4]. However, this value is a useful piece of information to guide the configuration. Finally, when the user has chosen a configuration, the final performance is analyzed again in order to obtain a more trustworthy performance rating.
- `SecurityAnnotation`: This component provides the interface for annotating the feature model with the impact on security. It also provides interfaces for loading, updating, and retrieving a feature model that has security mechanisms as features. Similar to performance, the impact of a feature on security cannot be precisely analyzed in isolation, but the annotation in the security feature model (reusable expert knowledge) is a useful piece of information to guide the configuration. Once the user chooses one configuration for the concrete product, this component will provide a more trustworthy security rating.
- `CostAnnotation`: This component provides the interface for annotating the feature model with cost values, which can be estimates of the effort for the development of the features, real development costs, or the selling price of the features. We assume that this information will be available.
- `ConfigurationManager`: This component provides the interface for retrieving all configurations generated by the solver along with their quality and cost ratings. It requires the `ProblemConversion` interface of `Solver` to convert the configuration problem into a CSP problem, and uses the `Solutions` interface to obtain all possible configurations that satisfy the conditions.
- `ProductDerivation`: This component provides the interface for generating the product specification in PSL, and requires the interface to know the chosen configuration. Tools from the HATS framework can be used to generate ABS code or even code in other languages (e.g., Java) from the product specification in PSL.

3.3 Integrating COSTABS

The `PerformanceAnnotation` component will use the COSTABS resource-usage analyzer to annotate the feature model with performance estimates. The annotation of performance, like any other annotation to the feature model, will be done before any configuration of concrete products from the product line, as a pre-processing task. There is no way to annotate a feature with performance information if the feature has not been implemented yet.

According to the ISO/IEC 25010 Standard - System and Software Quality Model [14], the product quality characteristic of *performance efficiency*, which means the performance relative to the amount of resources used, can be broken down into the sub-characteristics of *time behavior* and *resource utilization*. The first is related to the response and processing times and the throughput rates of a system when performing its function. The latter is related to the amounts and types of resources used when the software performs its function. Further ISO/IEC standards [13] provide internal and external metrics for performance efficiency. External

[4]This is the reason for using the term "indicator", which applies to both performance and security.

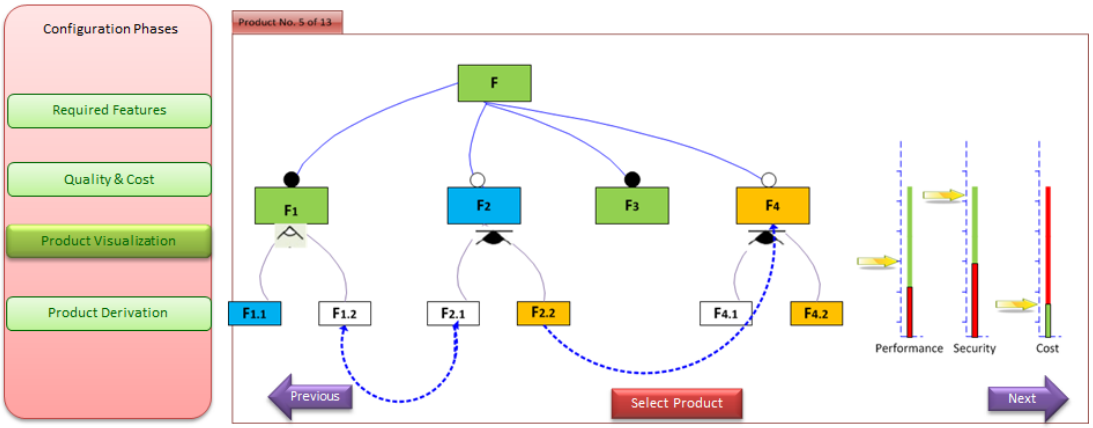

Figure 1: Browsing through the generated configurations with quality and cost rating

metrics are applied to the running system, whereas internal metrics look at static attributes of the software product.

Since COSTABS is a static analyzer for resource usage, our interest is on *internal* performance efficiency metrics. The outcome of COSTABS is platform-independent and factors out many details of the execution. Thus, it is neither possible nor reasonable to try and infer time bounds in milliseconds. Instead, what COSTABS can provide is, for example, the number of semantic steps or memory blocks used, which is less concrete but more flexible. Among the internal performance efficiency metrics suggested by the ISO/IEC standard [13], the most adequate are *response time* and *memory utilization*. We have redefined those metrics as follows:

- *Response time*: the maximal number of steps which are needed to complete a specific task. Here, the upper bound comes as a symbolic expression that provides information about the efficiency of the software response.

- *Memory utilization*: the maximal amount of memory that is allocated at runtime while executing a task.

As shown in Section 2.3, upper bounds come in the form of a symbolic expression which may sometimes be hard to read for the developer, and are not easy to deal with in the context of the product configuraton process. Therefore, there will be some post-processing of the COSTABS output, which will get a normal form for an upper bound and will be able to get its *complexity order* in a user-friendly representation.

Even if there is post-processing as described above, using the performance results returned by the static analyzer in the product configuration process is not trivial. There are two possibilities:

- Based on the complexity order of some task, the user of the `PerformanceAnnotation` component(a software developer) decides whether the response time is to be considered as being low, medium, or high. This alternative requires additional effort and some specific knowledge from the software developer, but is more flexible and precise since the software developer should know exactly what to expect from a piece of code.

- Some heuristic is defined in the component to classify the result of the static analysis into a quality degree. For example, it may make sense to state that an exponential algorithm has low-quality response time, while a linear or sub-linear one has high-quality response time, and polynomials are to be considered of medium quality. This alternative is simpler but less precise since, clearly, there is no ground for dividing algorithms into good ones and bad ones only on the basis of their complexity order.

In any case, the component will allow the user to provide additional information: For example, the software developer may want to consider an input parameter of a method as a constant, because its size is known to be very small in all cases. This would be a valuable piece of information since it permits to obtain a finer-grained estimation of performance.

Once the response time and the memory utilization of some piece of code have been analyzed, the `PerformanceAnnotation` component is responsible for annotating the respective feature accordingly. The `Solver` component will use the annotations in the feature model to find configurations that maximize performance and security. For individual configurations, a global result for response time and memory utilization will be calculated: For example, the product has (1) *high quality* in response time if the response time of every relevant task is high-quality; (2) *medium quality* if most tasks have high-quality response time, and few of them can present some criticalities, and (3) *low quality* otherwise.

Due to possible loss of information, which is unavoidable in static analysis, there is no guarantee that the recommended configurations which will be the best in terms of performance. However, performance annotations are still useful for guiding the process since, in most cases, the recommended configurations will be close to the optimal result.

3.4 Research Issues and Potential Extensions

Some research issues to further investigate and potential extensions have been identified:

User empowerment X auto-configuration: According to our vision, the configuration process is automatized as much as possible. However, decisions are taken by the

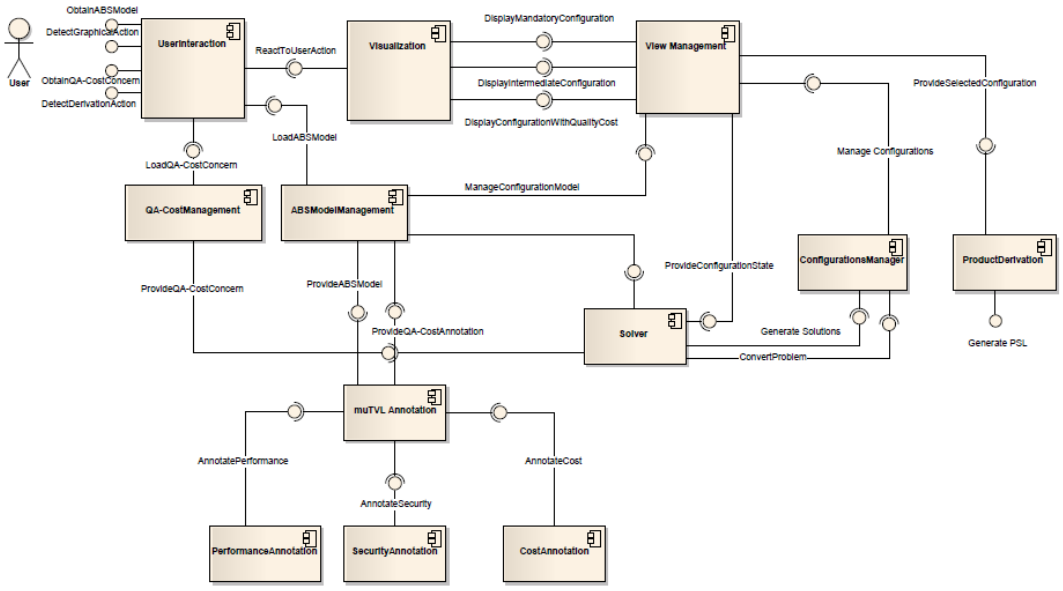

Figure 2: Component Model of the Proposed Configurator

user who also provides essential information for the efficiency of the configuration process. We believe this to be an interesting balance between automatic and interactive configuration.

Performance annotations: The performance of features cannot be precisely analyzed in isolation. On-the-fly analysis of the performance of product configurations would provide more precise results, especially in concurrent settings. We have decided to run COSTABS at preprocessing time, and annotate the feature with performance information, so that the configurations recommended by `Solver` already take this information into consideration. Once the user chooses one of the recommended configurations, its performance can be analyzed in order to obtain more precise information.

Security annotations: Security mechanisms are the first step for achieving security, but security mostly depends on how those mechanisms are implemented. We have decided to capture in a reusable feature model and in delta models expert knowledge about security mechanisms and the way they contribute to security. We have been investigating the modeling techniques from Section 2.1 in order to identify the one that best fits both security and performance variability modeling. As in performance, once the user chooses one of the recommended configurations, its security can be analyzed in order to obtain more precise results.

Performance of the Solver: We intend to build upon the conversion of the feature selection problem to CSP proposed by [23]. White et al. [23] reported a response time of 3 seconds for a 3-step configuration process based on a feature model with 500 features. We want to find out the response time for some thousand features and investigate whether the work reported in [22] offers a better alternative.

Quality concerns per sub-tree: The user might want to specify that a quality concern is relevant only for a part of the feature model. We decided not to deal with this issue for the moment, which will impact on the `QA-CostManagement` and `Solver` components.

4. CONCLUSIONS

This paper presents an infrastructure for helping developers in configuring a product from a product line infrastructure based on quality (performance and security) concerns. The presented solution builds upon the HATS framework in order to support the user in selecting features from a feature model such that the choice goes in the right direction w.r.t. the quality of the final product. To the best of our knowledge, there is no such approach in the literature.

Future work will focus on further investigating the research issues mentioned in Section 3.4 in order to get a sound basis for the development of our solution. From the design point of view, work in the very near future will be devoted to defining the criteria according to which a product is to be considered as having a high, medium, or low degree of a certain quality characteristic. This is, in general, a difficult task because it involves considering, among other things, the context in which the product will be used. Moreover, there is no general definition for such categories. From the technical point of view, we will analyze the existing (partial) implementations of several parts of our solution (in particular, of the `PerformanceAnnotation`, `Solver`, and `Visualization` components) and make the necessary extensions/adaptations so that they can be linked to the new parts. The complete solution will be evaluated within the HATS project.

5. REFERENCES

[1] E. Abbasi, A. Hubaux, and P. Heymans. A toolset for feature-based configuration workflows. In *SPLC 2011*,

pages 65–69. IEEE, August 2011.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. COSTABS: A Cost and Termination Analyzer for ABS. In *PEPM 2012*, pages 151–154. ACM Press, January 2012.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, February 2011.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortez. Automated Reasoning on Feature Models. In *CAiSE 2005*, pages 491–503. Springer, June 2005.

[5] D. Clarke, N. Diakov, R. Haehnle, E. Johnsen, I. Schaefer, J. Schaefer, R. Schlatte, and P. Wong. Modeling spatial and temporal variability with the hats abstract behavioral modeling language. In *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 162–164. Springer, 2004.

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2), 2005.

[8] K. Czarnecki, S. She, and A. Wasowski. Sample spaces and feature models: There and back again. In *SPLC 2008*, pages 22–31. IEEE, Setember 2008.

[9] L. Etxeberria and G. Sagardui. Evaluation of Quality Attribute Variability in Software Product Families. In *ECBS 2008*, pages 255–264. IEEE, March 2008.

[10] B. González-baixauli, J. C. S. do Prado Leite, and J. Mylopoulos. Visual Variability Analysis for Goal Models. In *RE 2004*, pages 198–207. IEEE, 2004.

[11] A. Hubaux and P. Heymans. On the evaluation and improvement of feature-based configuration techniques in software product lines. In *ICSE 2009*, pages 367–370. IEEE, May 2009.

[12] A. Hubaux, P. Heymans, P.-Y. Schobbens, D. Deridder, and E. Abbasi. Supporting multiple perspectives in feature-based configuration. *Software and Systems Modeling*, pages 1–23, November 2011.

[13] ISO/IEC. *Software Engineering - Product Quality - Part3: Internal Metrics*, 2002.

[14] ISO/IEC. *Systems and Software Engineering - Systems and software product Quality Requirements and Evaluation - System and software quality models*, 2011.

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

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[17] J. Kuusela and J. Savolainen. Requirements engineering for product families. In *ICSE 2000*, pages 61–69. ACM, April 2000.

[18] J. Lee and K. C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *SPLC 2006*, pages 131–140. IEEE, August 2006.

[19] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF : A Framework for Modeling Variability in Software Product Families. In *SPLC 2004*, volume 3154, pages 197–213. Springer, 2004.

[20] S. Soltani, M. Asadi, M. Hatala, D. Gasevic, and E. Bagheri. Automated planning for feature model configuration based on stakeholders' business concerns. In *ASE 2011*, pages 536–539. IEEE, November 2011.

[21] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[22] J. White, B. Dougherty, and D. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Systems and Software*, pages 1268–1284, August 2009.

[23] J. White, B. Dougherty, D. Schmidt, and D. Benavides. Automated reasoning for multi-step feature model configuration problems. In *SPLC 2009*, pages 11–20. Carnegie Mellon University, August 2009.

[24] J. White, D. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortez. Automated diagnosis of product-line configuration errors in feature models. In *SPLC 2008*, pages 225–234. IEEE, Setember 2008.

[25] H. Zhang, S. Jarzabek, and B. Yang. Quality Prediction and Assessment for Product Lines. In *CAiSE 2005*, pages 681–695. Springer, June 2003.