Dynamic Decision-Making based on NFR for Managing Software Variability and Configuration Selection

André Almeida[1,2], Nelly Bencomo[3], Thais Batista[2],
Everton Cavalcante[2,4], Francisco Dantas[5]

[1]Federal Institute of Education, Science and Technology of Rio Grande do Norte, Parnamirim, Brazil
[2]Federal University of Rio Grande do Norte, Natal, Brazil
[3]Aston University, Birmingham, United Kingdom
[4]IRISA-UMR CNRS/Université de Bretagne-Sud, Vannes, France
[5]State University of Rio Grande do Norte, Natal, Brazil

andre.almeida@ifrn.edu.br, nelly@acm.org, thais@ufrnet.br,
evertonrsc@ppgsc.ufrn.br, franciscodantas@uern.br

ABSTRACT

Due to dynamic variability, identifying the specific conditions under which non-functional requirements (NFRs) are satisfied may be only possible at runtime. Therefore, it is necessary to consider the dynamic treatment of relevant information during the requirements specifications. The associated data can be gathered by monitoring the execution of the application and its underlying environment to support reasoning about how the current application configuration is fulfilling the established requirements. This paper presents a dynamic decision-making infrastructure to support both NFRs representation and monitoring, and to reason about the degree of satisfaction of NFRs during runtime. The infrastructure is composed of: (i) an extended feature model aligned with a domain-specific language for representing NFRs to be monitored at runtime; (ii) a monitoring infrastructure to continuously assess NFRs at runtime; and (iii) a flexible decision-making process to select the best available configuration based on the satisfaction degree of the NRFs. The evaluation of the approach has shown that it is able to choose application configurations that well fit user NFRs based on runtime information. The evaluation also revealed that the proposed infrastructure provided consistent indicators regarding the best application configurations that fit user NFRs. Finally, a benefit of our approach is that it allows us to quantify the level of satisfaction with respect to NFRs specification.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.13 [**Software Engineering**]: Reusable Software; K.6.3 [**Software Management**]: Software Selection

General Terms

Algorithms, Design, Measurement

Keywords

SPLs, Variability, Non-functional requirements, Monitoring

1. INTRODUCTION

The growing demand for software solutions to be used in different domains associated with the interest in satisfying customers in highly dynamic environment, have led software industry to face the need of dealing with variability-rich software [7]. Developers should construct systems to manage variability, allowing dynamic derivation of different software versions in order to meet user requirements.

Software product lines (SPLs) have been widely used to address variability [16]. SPLs enable the creation of a *family* (or product line) of similar products by identifying *commonalities* between members of the family, as well as characteristics that vary among them, the so-called *variabilities*. At design time, SPL engineering uses the so-called *feature models* [15] to express commonalities and variabilities in terms of *features* and their relationships. As an example, consider the feature model presented in Figure 1, which describes an SPL for a mobile application configuration. This feature model specifies two features, *Connectivity* and *Storage*. Alternatives for *Connectivity* are *2G*, *3G*, and *WiFi*. For *Storage*, the alternatives are *Amazon DynamoDB* and *SQLlite*.

Although feature models are used in SPL to capture the essential requirements of a system, they lack of expressiveness for describing non-functional requirements (NFRs) [14].

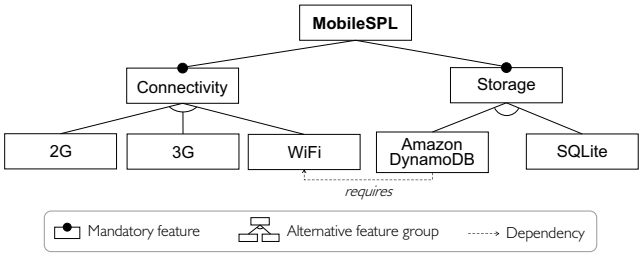

Figure 1: Feature model of *MobileSPL*.

For example, consider a user concerned with the battery consumption of a device and another user particularly interested in improving the performance of the storage system. Can the continuous use of a WiFi connectivity reduce the battery performance so that it would be better to use 3G or 2G connectivity? Can the use of the Amazon DynamoDB storage compromise the performance of the system so that the use of SQLite is preferred to improve performance? The answer to these questions will depend on information that can be gathered only at runtime. We argue it is necessary to *specify* and *monitor* the associated NFRs to feed the decision-making process aiming to meet user requirements. As traditional feature models are not able to capture these concerns, *extended feature models* [3] enable the representation of *properties* in feature models for selecting variants. Such properties can be the basis for expressing NFRs and hence supporting selection of application configurations. However, if the values of these properties change over time, it might affect the user perception of the NFRs satisfaction.

In this paper, we argue that NFRs must be dynamically assessed in order to check whether they are being satisfied or not. In order to perform this assessment, NFRs specification must encompass quantifiable properties regarding an application configuration as the foundation for describing such requirements. Furthermore, flexibility of NFRs is an important concern mainly in scenarios whose conditions do not allow the strict meeting of the NFRs, but it is still possible to provide a suitable solution from the user point of view.

We propose an infrastructure for monitoring and reasoning about NFRs at runtime, aiming to continuously verify if NFRs are met and to optimally select features to compose configurations. To achieve this goal, we present: (i) an extended feature model with properties that can be assessed and monitored at runtime (Section 2.1); (ii) a domain-specific language, called *DynamicNFR*, to specify NFRs using the properties defined in the feature model (Section 2.2); (iii) a monitoring system for continuously assessing the values of the properties at runtime; and (iv) a decision-making process to select the best available configuration and its satisfaction degree in terms of the defined NFRs (Section 3). We discuss the correlation between selected configurations and their degree of user satisfaction in Section 4. Finally, we present related work (Section 5) and some conclusions and directions to future work (Section 6).

2. VARIABILITY MODELING AND NFR SPECIFICATION

2.1 Extended Feature Models

SPL approaches usually identify *commonalities* between all members of the family as well as characteristics that vary among them, the *variabilities*, so that these members have a basic set of common features and associated variations that individualize each of these members. Commonalities, variabilities, and variation-related constraints are represented by *feature models* [9] in terms of *features* and their relationships. Feature models are typically structured as a tree in which features are represented by nodes, whereas variations between features are represented by edges and feature groups. Features can be [12]: (i) *mandatory*; (ii) *optional*; (iii) *or-inclusive*, so that at least one feature is selected from a set of related features; and (iv) *alternative*, so that exactly one feature is selected among a set of related features.

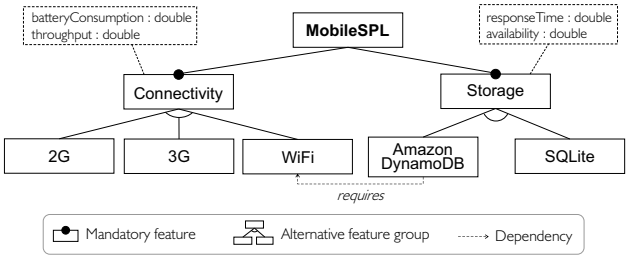

Figure 2: Extended feature model of *MobileSPL*.

There are multiple notations for feature modeling. In our approach, we use an *extended feature model* inspired in the one proposed by Czarnecki et al. [8], which allows introducing *attributes* to features. In this perspective, we ground on this idea with the notion of *properties*, <*name*, *type*> pairs that represent any information about a given feature. These properties are applied to features that have sub-features representing variabilities (the leaves of the feature model). Figure 2 depicts a new version of the feature model described in Figure 1 with some properties. For instance, the *Connectivity* feature has two properties: (i) *batteryConsumption*, which represents the consumption of the battery of a mobile phone when using one of the possible technologies (*2G*, *3G* or *WiFi*), and; (ii) *throughput*, which measures the rate of successful message delivery using the selected technology. For the *Storage* feature, other properties are defined: (i) *responseTime*, which measures the time spent for storage operations, and; (ii) *availability*, which represents the availability rate of the used strategy.

2.2 DynamicNFR: a DSL for Specifying NFRs

After describing the feature model in terms of variabilities and properties, NFRs are specified in order to support the selection of the application configuration. These NFRs are described based on the properties modeled in the feature model by using *DynamicNFR*, a domain-specific language (DSL) proposed in the context of this work. *DynamicNFR* was partly inspired in RELAX [20], a language for specifying requirements for self-adaptive systems. We have taken advantage on the quantifiable operators defined in RELAX for establishing *thresholds* in the NFRs specifications, which are used to bring flexibility to such NFRs. In this work, such thresholds are defined by the user.

Each *specification* of a given NFR is composed of a name and at least one high-level description. Similarly to the model used by NFR-Framework (NFR-F) [6], each NFR can be described in terms of other NFRs. However, different from our proposal, the NFR-F does not use properties that can be dynamically monitored.

The realization of NFRs are described as *Rules* applied to the properties modeled in the feature model. *Rules* can be *Objectives* or *Constraints*. *Objective* rules are described in terms of maximization (max) and minimization (min) functions applied to a specific feature/property described in the feature model. In turn, *Constraint* rules define threshold operators that are applied in the process for selecting the *best* available configuration. The as_far_as_possible operator establishes that a value for a given property should be as far as possible to the defined threshold. Similarly, the as_close_as_possible operator establishes the opposite, i.e., the value of a property should be as close as possible to the defined

```
1  nfrequirement AchieveEffectiveness
2  begin
3     MaintainPerformance
4     MaintainDependability
5     ImproveBatteryLife
6  end
7  nfrequirement MaintainPerformance
8  begin
9     as_close_as_possible Connectivity.throughput 80
            allowance 10
10    min Storage.responseTime
11 end
12 nfrequirement MaintainDependability
13 begin
14    max Storage.availability
15 end
16 nfrequirement ImproveBatteryLife
17 begin
18    min Connectivity.batteryConsumption
19 end
```

Figure 3: *DynamicNFR* specification of NFRs 1-4.

threshold. Another aspect tackled by the *DynamicNFR* language is enabling users to express how much they are able to accept configurations that do not strictly meet their needs in terms of the defined thresholds. In order to specify the flexibility of a given threshold, operators defined for *Constraint* rules have a third parameter (allowance), a percentage that establishes the range of flexibility for such a threshold.

To illustrate the language, consider the extended feature model presented in Figure 2. Figure 3 shows a *DynamicNFR* specification of the following NFRs:

- **NFR1:** The application should be effective. To achieve effectiveness, the application must be dependable, maintain its performance, and improve battery life.
- **NFR2:** To improve battery life, battery consumption on sending data over a connection should be minimized.
- **NFR3:** To achieve dependability, storage availability should be maximized.
- **NFR4:** To improve performance, throughput must be as close as possible to 80% and storage response time should be minimized.

The `AchieveEffectiveness` NFR (line 1) can be achieved by fulfilling the `MaintainPerformance`, `MaintainDependability`, and `ImproveBatteryLife` NFRs (lines 3 to 5). The `MaintainPerformance` NFR (line 7) is composed of two *Rules*: (i) the as_close_as_possible *Constraint* in line 9 establishes that `throughput` should be near 80% with allowance of 10%, thus meaning that a 72% rate for throughput is in an acceptable range; and (ii) the min *Objective* in line 10 for minimizing `responseTime` for the `Storage` feature. The same applies to the `MaintainDependability` (lines 12 to 15) and `ImproveBatteryLife` (lines 16 to 19) NFRs.

3. DECISION-MAKING ARCHITECTURE

Figure 4 depicts the architecture of our proposed decision-making infrastructure. Each application/domain has a specific *Monitoring System*, which is responsible for gathering data used to select the best available application configuration. Acquired data stored at the *Monitored Data* database

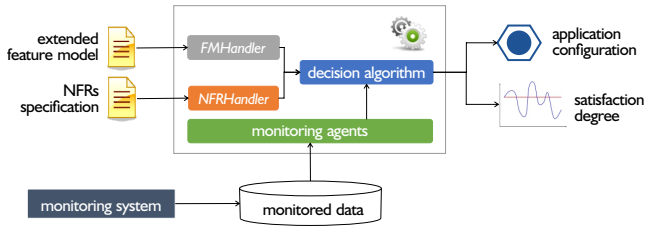

Figure 4: Architecture of the proposed decision-making infrastructure.

are time-stamped to support either just-in-time decisions and historical analysis if required by the decision-making mechanism. In the proposed solution, *Monitoring Agents* are entities responsible for querying the database to support the decision process. The *FMHandler* component analyzes the specification of the feature model and generates the solution space, which consists of the possible configurations of the application, as described in the feature model. In turn, the *NFHandler* component parses the *DynamicNFR* specification by translating it to a function used by the *Decision Algorithm* to select the best available configuration, as detailed in Section 3.2. The next subsections detail these elements.

3.1 Handling Feature Models

The feature model is represented as an XML file adapted from the one generated by FeatureIDE [19], a framework for creating XML representations of feature models. In order to encompass an extended feature model with annotated features, we have introduced two new tags into such an XML representation, namely `properties` and `property`. The `properties` tag starts a section in which each property associated to a specific feature is described by a `property` tag. For each variation point (`feature` tag), there is a *Monitoring Agent* responsible for gathering the values associated with the described properties. In turn, the *FMHandler* component parses such an XML representation of the feature model, thus allowing the *Decision Algorithm* to query for the possible configurations that can be generated from the feature model.

Figure 5 presents an XML fragment of the extended feature model description presented in Figure 2. A mandatory feature named `Storage` is defined in line 3 with two alternative features, `AmazonDynamoDB` (line 4) and `SQlLite` (line 5). For each alternative feature, an `agent` attribute is defined to point to the class name of the *Monitoring Agent* responsible for querying the *Monitored Data* database regarding the monitored properties. These properties are defined in lines 7 to 10 with their respective name and type.

3.2 Handling NFR Specifications

The *NFRHandler* component parses a NFR specification by translating it to a function used by the *Decision Algorithm*. Such a NFR specification for selecting a configuration can be viewed as a multi-objective problem as there is more than one NFR to be simultaneously satisfied. In order to simplify the decision process, this work assumes that all NFRs have the same priority (i.e., there is a single-objective function). In order to transform the multi-objective problem into a single-objective one, we define an *utility function* (UF) expressed by Equation 1:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<alt mandatory="true" name="Storage">
   <feature mandatory="true" name="AmazonDynamoDB"
        agent="com.spl.monitoring.
       MonitorAmazonDynamoDB"/>
   <feature mandatory="true" name="SQlLite" agent=
       "com.spl.monitoring.MonitorSQlLite"/>
</alt>
<properties>
   <property name="availability" type="double"/>
   <property name="responseTime" type="double"/>
</properties>
...
```

Figure 5: XML fragment of the description of the *MobileSPL* extended feature model.

$$UF = maximize\left(\sum_{i=1}^{n}\sum_{j=1}^{l} norm(p_{ij}) + \sum_{y=1}^{o}\sum_{k=1}^{m} dist(p_{yk}, c_y)\right) \quad (1)$$

in which p_{ij} represents a property of a commonality i and a variability j in the feature model. As the monitored properties can vary in terms of scale from a feature to another, it is necessary to *normalize* these values in order to have an uniform scale to be used in UF. According to the *Objective* rule to be applied to a property, its value is normalized by applying the *norm* function, as expressed by Equations 2 and 3 (for minimization and maximization, respectively):

$$norm(p_{ij}) = \begin{cases} \frac{p_i^{max} - p_{ij}}{p_i^{\max} - p_i^{min}}, & p_i^{\max} - p_i^{\min} \neq 0 \\ 1, & p_i^{\max} - p_i^{\min} = 0 \end{cases} \quad (2)$$

$$norm(p_{ij}) = \begin{cases} \frac{p_{ij} - p_i^{\min}}{p_i^{\max} - p_i^{\min}}, & p_i^{\max} - p_i^{\min} \neq 0 \\ 1, & p_i^{\max} - p_i^{\min} = 0 \end{cases} \quad (3)$$

As a NFR specification can be composed of a set of threshold definitions, the second part of Equation 1 is related to the normalized *distance* of the current value associated to a property from the threshold established by the associated *Constraint* rules. In Equation 4, the *dist* function takes the value of the property p indicated by the constraint y and the variability k, and the threshold value c:

$$dist(p_{yk}, c_y) = \begin{cases} norm(c_y - p_{yk}), & \text{if as\_close\_as\_possible} \\ norm(p_{yk} - c_k), & \text{if as\_far\_as\_possible} \end{cases} \quad (4)$$

3.3 Decision Algorithm

A strategy for dynamically selecting configurations based on NFRs can be designed in several ways and it can consider the specificities of a given application/domain. Our solution aims to support developers in terms of deploying their own decision algorithm based on their needs. Algorithm 1 presents a generic view of how the proposed solution works. The algorithm takes the XML representation of the feature model (*FM*) and the NFRs specification (*NFRSpec*) as fixed inputs and it returns the best available application configuration (*BSolution*) and the satisfaction degree (*SD*)

```
<configuration featuremodel="MobileSPL">
   <feature name="Connectivity">
      <variability>3G</variability>
   </feature>
   <feature name="Storage">
      <variability>SQLlite</variability>
   </feature>
</configuration>
```

Figure 6: XML description of a selected configuration for *MobileSPL*.

on how the selected configuration meets the NFRs. The first step (line 1) is to parse the *NFRSpec* for generating the utility function (*UF*) used to evaluate a given configuration. For each monitoring cycle of the *Monitoring System*, the required data are loaded (line 3) by associating the feature model and the monitored data (*MData*). Next, the decision algorithm is applied to generate the selected configuration as an XML file containing the selected features (line 4). Figure 6 shows a selected configuration based on the feature model of Figure 2, in which the *3G* and *SQlLite* variants were selected for the *Connectivity* and *Storage* features, respectively. Finally, the *BSolution* configuration is evaluated in terms of the given *NFRSpec* to generate the satisfaction degree (*SD*).

Input : MData – monitored data, FM – feature model, and NFRSpec – NFRs specification
Output: BSolution – best available solution and SD – satisfaction degree

```
1 UtilityFunction ← ParseNFRs(NFRSpec)
2 while monitoring_cycle is active do
3 |   LoadData(FM, MData)
4 |   BSolution ← ApplyDecisionAlgorithm(FM,
  |   UtilityFunction)
5 |   SD ← EvaluateSD(BSolution, NFRSpec)
```

Algorithm 1: Decision-making generic algorithm for selecting an application configuration.

4. DISCUSSION

Our approach is illustrated using HW-CSPL, an SPL developed from the Health Watcher system [18]. In Section 4.1, we describe an instantiation of the architecture proposed in Figure 4 for the specific case of HW-CSPL. In Section 4.2, we discuss findings obtained from our investigation.

4.1 Study Illustration

Extended Feature Models. Figure 7 illustrates the HW-CSPL extended feature model, which contains three mandatory features representing commonalities: (i) *Persistence*, i.e., the persistence mechanism; (ii) *Log System*, i.e., the infrastructure used for storing log information; and (iii) *File Storage*, to define how files are managed. Each of these top-features has different variants. For instance, the *Persistence* feature has three dynamic properties (*price*, *availability*, and *responseTime*) and offers three options for data persistence, respectively represented by: (i) *Relational Amazon RDS*, the cloud database service by the Amazon Web Services platform; (ii) *the Google Cloud SQL* feature, the relational database service by the Google's cloud platform; and (iii) *the RackSpace Databases* feature, the SQL database service by the Rackspace cloud platform.

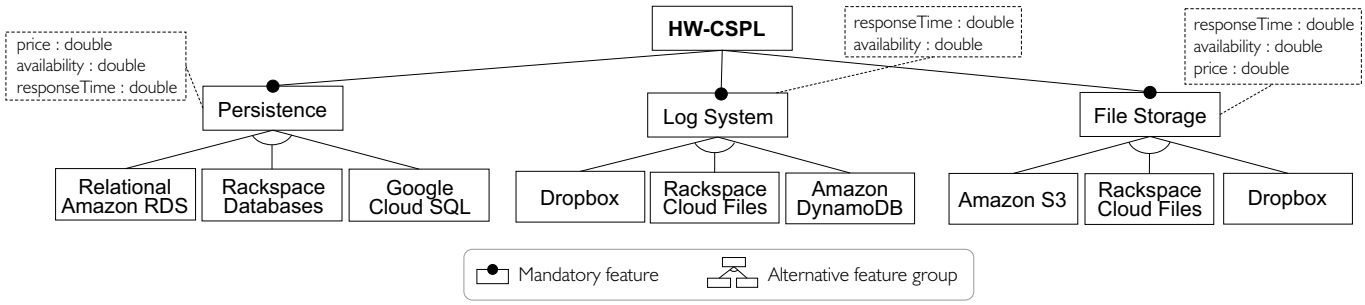

Figure 7: HW-CSPL extended feature model.

```
1  nfrequirement EffectivenessAchieved
2  begin
3     PerformanceMaintained
4     DependabilityMaintained
5  end
6  nfrequirement PerformanceMaintained
7  begin
8     min LogSystem.responseTime
9     as_far_as_possible FileStorage.responseTime 1
          allowance 5
10    as_far_as_possible Persistence.responseTime 1.5
           allowance 20
11 end
12 nfrequirement DependabilityMaintained
13 begin
14    max Persistence.availability
15    as_close_as_possible FileStorage.availability
          99.50
16 end
```

Figure 8: *DynamicNFR* specification of NFRs 1-3.

Monitoring System. As previously discussed, a *Monitoring System* is required for each application/domain to acquire data related to the properties defined in the extended feature model to support the selection of the best available configuration. For HW-CSPL, we have used *QoMonitor* [2] as the monitoring solution. In order to monitor the defined properties, *QoMonitor* requires to: (i) write a Web service that invokes a service on a specific cloud platform; (ii) register such a Web service in the *Monitoring System*; and (iii) define the time interval between information gathering, thus establishing the monitoring cycle.

NFRs Specification. NFRs are specified by using the *DynamicNFR* language based on the extended feature model and the properties whose values are gathered by the monitoring system. Figure 8 illustrates the *DynamicNFR* specification of the following three NFRs: (1) the application should be effective, i.e., it must be dependable and maintain its performance; (2) to improve performance, the response time for log services should be minimized and the response time for storage and persistence should be as far as possible from 1 s (with a tolerance of 5%) and 1.5 s (with a tolerance of 20%), respectively; and (3) to achieve dependability, the availability for persistence should be maximized and the availability for storage should be as close as possible to 99.5%.

Decision Algorithm. The *NFRHandler* generates the utility function (UF) presented in Equation 5 based on the requirements specification presented in Figure 8. This function aims to maximize the expected utility for a given configuration selected from the HW-CSPL extended feature model. For each *Rule* in the NFRs specification, an operand is generated by using the process described in Section 3.2. For instance, $dist\_far(FileStorage.responseTime, 1.05)$ is the mathematical representation for the *Rule* in Figure 8 (line 9). The value 1.05 represents the defined threshold (1.0) increased by the allowance of 5% defined in the specification. The different configurations analyzed by UF were selected by using a well-known algorithm, the so-called Branch-and-Bound (B&B) [13], as detailed in our previous work [1]. In order to apply the B&B technique, the problem is described as a tree in which each node represent a partial solution. The algorithm analyzes the UF value for the current selected configuration (node) and it verifies whether the given partial configuration improves such a selected configuration, thus branching a sub-node. The best available configuration is returned by the algorithm when there is no other node to search and analyze.

$$\begin{aligned} UF = maximize(&norm\_min(LogSystem.responseTime) \\ &+dist\_far(FileStorage.responseTime, 1.05) \\ &+dist\_far(Persistence.responseTime, 1.8) \\ &+norm\_max(Persistence.availability) \\ &+dist\_close(FileStorage.availability, 99.5)) \end{aligned} \quad (5)$$

4.2 Utility Function vs. Satisfaction Degree

In our investigation, we have monitored 1000 different cycles based on the utility function defined in Equation 5. The values of UF were generated based on the translation of the NFRs specified in Figure 8 by using the process presented in Section 3.2.

Figure 9 illustrates the variation of the user satisfaction during these cycles[1]. This variation refers to the average values for the final configuration, which is composed of the respective variabilities (cloud services) regarding the *Persistence*, *Log System*, and *File Storage* features of HW-CSPL. The satisfaction degree has varied from 70% to 100% (92.73% in average) and there was a concentration of values around 100% when the value for UF is maximum, thus indicating a tight relationship between UF values and the degree of user satisfaction. It is important to mention that the B&B algorithm used in our study ensures selecting the optimal solution.

As shown in Figure 9, it is noteworthy that there are cases in which the value of UF is maximum and equal to 5.0 (the

[1]The monitored data used to generate the chart are available at `http://www.dimap.ufrn.br/splmonitoring`.

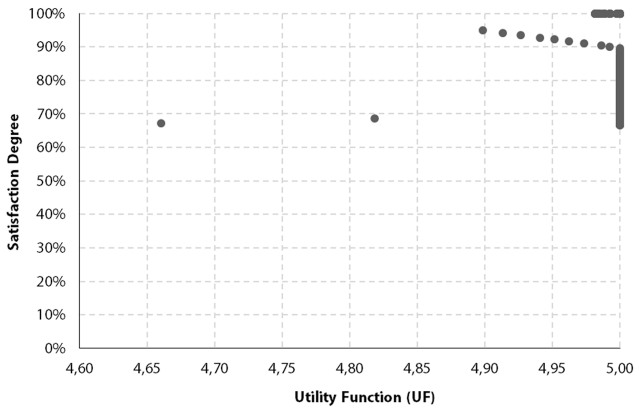

Figure 9: Utility function vs. satisfaction degree.

best configuration for the current scenario), whereas the satisfaction degree is less than 100%. This means that even if the best possible configuration is selected, it does not fully meet the user expectations. If NFRs are described by using only the max and min operators, one can have a 100% of satisfaction with UF = 5.0 (max) as the approach always finds a maximum and minimum values. However, as the as_far_as_possible or as_close_as_possible criteria are also used in our approach, even if a configuration has UF = 5.0, it might not fully meet the requirements as it is out of the thresholds defined by these operators. For instance, this is the case of a response time of 2 s (the best available) despite NFRs indicate that 1 s is the optimal response time expected by the user. The opposite situation also occurs with UF < 5.0 and satisfaction degree of 100%. This means that even when the best configuration value found by the B&B algorithm is lower than 5.0, the presented solution satisfies the NFRs.

5. RELATED WORK

Non-functional properties in SPLs. The Benavides et al.'s work [3] is one of the first works that consider non-functional properties in SPLs. The authors introduce the idea of annotating features with *attributes* in order to enable reasoning about product derivation based on measures of such properties. Moreover, they present a technique based on Constraint Satisfaction Problem (CSP) solvers to find an optimal product. After mapping the feature model to a CSP, a CSP solver evaluates the values of the attributes attached to the features and then it computes an optimal configuration for a small number of features. However, in such a proposal, the measurement of the values of the properties is an open problem.

Few proposals in literature systematically consider measurements of non-functional properties within SPLs or enable to optimize the feature selection for a specific non-functional property. As an example, Siegmund et al. [17] introduce *SPL Conqueror*, an approach that is quite similar to ours in terms of annotating features with measurable properties in order to select features/configurations that meet both functional and non-functional requirements, as well as to find the best product configuration based on such properties. The activities performed by the *SPL Conqueror* tool are: (i) to specify the extended feature model with the respective properties; (ii) to specify mechanisms/tools responsible for assessing the defined properties, similarly to our monitoring agents; (iii) to specify constraints based on the defined properties to remove feature/variants that do not meet NFRs; (iv) to find an optimal variant that meets NFRs by using a CSP solver; and (v) to perform a post-derivation optimization, which stands for applying optimizations after selecting a variant/set of features. Nevertheless, *SPL Conqueror* lacks of support for monitoring non-functional properties at runtime, so that it regards properties as static ones. Therefore, the decision-making process about selecting application configurations does not consider dynamic information. For this reason, it is not possible to ensure that an application configuration that initially meets user NFRs continues to satisfy such requirements after its deployment/execution.

Quantification of NFRs satisfaction due to Uncertainty. Several research initiatives have started tackling the challenge of quantifying levels of uncertainty associated with NFRs. As an example, the RELAX language [20] can be used to specify requirements of self-adaptive systems under uncertainty. A system whose requirements have been specified using RELAX is able to temporarily *relax* a non-critical requirement to ensure that critical requirements (invariants) can still be satisfied, so that analysts can identify the requirements that are RELAXable. *DynamicNFR* uses the quantifiable operators defined in RELAX to establish thresholds that are used to bring quantification of satisfaction levels of NFRs based on feature properties. Different from RELAX that solves such a quantification using temporal fuzzy logic, in *DynamicNFR* the user is required to specify the thresholds of tolerance, which are then used to quantify how close (or far) are the levels of NFRs satisfaction from those ideals. As in RELAX, the authors in [11] also use fuzzy logic to reason about NFRs under uncertainties.

The authors in [4, 5] use Bayesian machine learning techniques to quantify the impacts of NFRs on configurations to therefor support decision-making. In [10], the authors exploit probability theory and probabilistic model checking to label possible alternative behaviours (or execution flows) indicating the likelihood of meeting the NFRs to enable informed decision-making. As in the case of any implementation of RELAX and [11] (and different from ours), these research initiatives use their specific techniques (either machine learning or model checking) to quantify levels of uncertainty and support decision-making. In our case, the threshold values provided by the user are used to guide the decision algorithm to provide an optimal solution. Our approach can therefore quantify how distant (close or far) the solution provided is from the required levels of satisfaction.

6. CONCLUSION AND FUTURE WORK

SPL techniques are used to represent commonalities and variabilities by using feature models to depict a clear view of possible configurations that an application can have. These configurations are conceived to satisfy both functional and non-functional requirements and are usually selected at design time. However, due to dynamic changes, the selected configuration may not fulfill the specified NFRs. In this paper, we argue that it is imperative to dynamically assess if such NFRs are met at runtime in order to select a better configuration when needed. For this purpose, we have introduced: (i) an extended feature model with annotated properties that can be monitored at runtime in order to represent the possible application configurations; (ii) *DynamicNFR*, a language to specify NFRs by using such properties; (iii) a monitoring system for assessing the values of the prop-

erties; and (iv) a decision-making process for selecting the best available configuration and for evaluating the degree of user satisfaction regarding this configuration. Our solution aims to be independent from specific domains/applications, thus enabling developers to plug their own monitoring solutions and/or decision algorithms. In order to evaluate the proposed solution, a SPL related to a Cloud Computing application was used to assess if a given NFRs specification was met at runtime. The results showed that the selected configuration had high satisfaction degree for almost 90% of the monitoring cycles.

The presented approach can be improved in several ways. First, we will expand the specification of NFRs to encompass not only properties related to features, but also properties related to configurations. In this perspective, NFRs can be specified based on properties that are assessed for a configuration/set of features, thus reducing the number of *Rules* described in a specification. Furthermore, as SPLs typically involve multiple stakeholders that might specify different NFRs, it is important to manage such a multiplicity of concerns and to provide means to solve possible conflicts between NFRs among different stakeholders. Moreover, we intend to consider a multi-objective perspective for the utility function in order to provide solutions that simultaneously optimize the defined *Objective* rules, mainly in cases of conflicting NFRs. Finally, we envision proposing a systematic strategy to define thresholds used in *Constraint* rules.

7. REFERENCES

[1] A. Almeida, F. Dantas, E. Cavalcante, and T. Batista. A Branch-and-Bound algorithm for autonomic adaptation of multi-cloud applications. In *Proc. of 14th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing*, pages 315–323. IEEE, 2014.

[2] C. Batista, G. Alves, E. Cavalcante, F. Lopes, T. Batista, F. C. Delicato, and P. F. Pires. A metadata monitoring system for Ubiquitous Computing. In *Proc. of 6th Int. Conf. on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 60–66, 2012.

[3] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proc. of 17th Int. Conf. on Advanced Information Systems Engineering*, volume 3520 of *LNCS*. Springer, Germany, 2005.

[4] N. Bencomo and A. Belaggoun. A world full of surprises: Bayesian theory of surprise to quantify degrees of uncertainty. In *Companion Proceedings of the 36th Int. Conf. on Software Engineering*, pages 460–463. ACM, 2014.

[5] N. Bencomo, A. Belaggoun, and V. Issarny. Dynamic decision networks to support decision-making for self-adaptive systems. In *Proc. of 8th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 113–122. IEEE, 2013.

[6] L. Chung and J. C. P. Leite. On non-functional requirements in Software Engineering. In *Conceptual modeling: Foundations and applications*, volume 5600 of *LNCS*, pages 363–379. Springer, Germany, 2009.

[7] P. Clements and L. Northrop. *Software product lines: Practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., USA, 2001.

[8] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. Generative Programming for embedded software: An industrial experience report. In *Proc. of the 2002 Conf. on Generative Programming and Component Engineering*, volume 2487 of *LNCS*, pages 156–172. Springer, Germany, 2002.

[9] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[10] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proc. of the 35th Int. Conf. on Software Engineering*, pages 33–42. IEEE, 2013.

[11] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic resource provisioning for cloud-based software. In *Proc. of the 9th Int. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 95–104. ACM, 2014.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, USA, Nov. 1990.

[13] T. Murata, H. Ishibuchi, and H. Tanaka. Multi-objective genetic algorithm and its applications to flowshop scheduling. *Computers & Industrial Engineering*, 30(4):957–968, 1996.

[14] J. Myopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[15] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, principles, and techniques*. Springer, Germany, 2005.

[16] K. Pohl and A. Metzger. Variability management in Software Product Line Engineering. In *Proc. of the 28th Int. Conf. on Software Engineering*, pages 1049–1050. ACM, 2006.

[17] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. *Information and Software Technology*, 55(3):491–507, 2013.

[18] S. Soares, P. Borba, and E. Laureano. Distribution and persistence as aspects. *Software – Practice and Experience*, 36(7):711–759, 2006.

[19] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79(1):70–85, 2014.

[20] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. RELAX: A language to address uncertainty in self-adaptive systems requirement. *Requirements Engineering*, 15(2):177–196, 2010.