User Profiles for Context-Aware Reconfiguration in Software Product Lines

Michael Nieke[1(✉)], Jacopo Mauro[2], Christoph Seidl[1], and Ingrid Chieh Yu[2]

[1] Technische Universität Braunschweig, Braunschweig, Germany
{m.nieke,c.seidl}@tu-braunschweig.de
[2] University of Oslo, Oslo, Norway
{jacopom,ingridcy}@ifi.uio.no

Abstract. Software Product Lines (SPLs) are a mechanism to capture families of closely related software systems by modeling commonalities and variability. Although user customization has a growing importance in software systems and is a vital sales argument, SPLs currently only allow user customization at deploy-time. In this paper, we extend the notion of context-aware SPLs by means of user profiles, containing a linearly ordered set of preferences. Preferences have priorities, meaning that a low priority preference can be neglected in favor of a higher prioritized one. We present a reconfiguration engine checking the validity of the current configuration and, if necessary, reconfiguring the SPL while trying to fulfill the preferences of the active user profile. Thus, users can be assured about the reconfiguration engine providing the most suitable configuration for them. Moreover, we demonstrate the feasibility of our approach using a case study based on existing car customizability.

Keywords: Dynamic Software Product Line · User profiles · Preferences · Reconfiguration · Context-awareness

1 Introduction

SPLs are a technique to capture families of closely related software systems and to allow large-scale reuse [21]. In a *variability model*, the conceptual commonalities and variabilities of software systems are captured, thus defining the set of all possible configurations. Feature Models (FMs) represent the commonalities and variabilities in terms of a hierarchical. structure of features. Features can be enriched by feature attributes, extending the variability of a feature by additional variables. A *configuration* is a set of selected features and values for attributes.

User customization already is of big importance in several domains, e.g., in the automotive domain, you can configure your car based on your own preferences. Multiple drivers of a car may have different preferences regarding how the car is configured and which features are most desirable under certain contexts. For instance, the Volkswagen AG presented a prototypical car at the Consumer Electronic Show (CES) 2015 which supports several user customizations

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part II, LNCS 9953, pp. 563–578, 2016.
DOI: 10.1007/978-3-319-47169-3_44

at runtime, such as the change of the engine profile, allowing a more powerful or power-saving engine performance [6,25]. Moreover, nowadays, most devices have to interact and adapt based on their environment. Consequently, the context of a device needs to be considered in the configuration process of an SPL. Additionally the desire of users to customize a product may arise at runtime under certain contexts. For instance, drivers may want to activate a line assistance systems only when they are driving on a highway. For practical purposes, it may be important also to minimize the number of changes that are required to be performed, thus generating configurations that should be similar to the initial one as much as possible.

In standard SPL development, users can customize the product only when ordering a product or at runtime using settings in the software. The first case lacks the possibility to customize the user's end device after ordering it, while the second case always requires user interaction.

In [15], we already consider and propose an integrated approach to model the influence of contextual information on SPLs. To this end, we use a model-based representation of contextual information and represent the influence of the context on particular features. Based on this representation, we developed `HyVarRec`, a context-aware reconfiguration engine which incorporates the FM, the influence of contextual information on features and the current context to reconfigure the product accordingly. This approach, however, lacks the possibility to take the user preferences into account and, similar to other context-aware approaches, it does not distinguish between required adaptation due to context information and desired customization wishes of users.

In this paper, to overcome these limitations, we integrate customization wishes of users into SPLs. To this end, we introduce a notion of *user profiles* containing sets of prioritized *preferences* reflecting the wishes of users. However, in contrast to constraints, it is not mandatory to satisfy preferences as they only need to be satisfied if it is possible. As a main contribution, we extended our approach [15] to take user profiles into consideration, thus, creating a reconfiguration engine that incorporates contextual information, user profiles and the current context at once. To prove the feasibility of our approach, we extend `HyVarRec` and we apply it to reconfigure and maximize the preferences of some users and minimize the changes to be performed within scenarios inspired by an existing Volkswagen AG SPL for cars.

In Sect. 2, we present background on the concepts our methodology is founded on. In Sect. 3, we introduce a running example based on a real car SPLs and the CES example of Volkswagen as well as exemplary contextual information and user preferences. In Sect. 4, we explain how SPLs can be enriched by user profiles and preferences. In Sect. 5, we present the extension of `HyVarRec` to cope with user preferences. In Sect. 6, we show the feasibility of our methodology by applying `HyVarRec` to several scenarios. In Sect. 7, we discuss related work. In Sect. 8, we close with a conclusion and an outlook to future work.

2 Background

Feature Models (FMs) are a hierarchical representation of the variability of an SPL. FMs consist of features representing configurable functionality of a system [12]. Each feature can have several child features and an FM has exactly one root feature. As an example, Fig. 1 depicts an instance of a FM. In this case, the root feature is `Car` while `Engine Profile` and `Gear Shift` are two of its children. Features can be *mandatory* or *optional* and can be organized in *alternative* or *or* groups. In alternative groups, exactly one feature has to be selected, whereas in or groups, at least one feature has to be selected. For instance, in Fig. 1, the feature `Lane Assist` is optional and the feature `Engine Profile` is mandatory and has its children organized as alternative group. Feature attributes are typed variables associated with a feature which are used to express more fine-grained variability [2]. It is possible to specify a domain for each attribute to limit its value range. For example, in Fig. 1, the feature `Cruise Control` owns an attribute `maxSpeed` of type `int` that has a domain of $[0, 300]$.

Features are realized by other artifacts (e.g., code or documentation) that may evolve and exist in different *versions*. To keep track of the evolution of features, it is necessary to explicitly capture these feature versions: old versions of a feature may indeed be preferred over the newer versions, e.g., to preserve the compatibility to other systems. Moreover, feature versions may have dependencies among each other. Following [27], we use Hyper Feature Models (HFMs) to represent the different versions and new restrictions for them. HFMs define a first-class concept for *feature versions* to represent different revisions of the artifacts associated with a feature (e.g., source code). For example, in Fig. 1, the feature `Lane Assist` has two different versions: a first one denoted as `1.0` and its following version denoted as `2.0`.

In FMs, cross-tree constraints (CTCs) are used to specify additional dependencies between features which are not specified via the hierarchical structure of the FM. CTCs can be defined as Boolean formulas on features and expressions on feature attributes. Additionally, CTCs can be used to restrict the version range of a feature by having a version-aware constraint language [27]. A *configuration* of an FM is a set of selected features, feature versions and values for the relevant feature attributes. A configuration is *valid* if it conforms with the hierarchy of the FM and the CTCs.

In [15], we presented an extension to FMs to capture contextual information allowing to model the dependencies of the selected features or attributes with respect to some external parameters and environmental situations. Contextual information is captured with identifier-value pairs where a context is associated with a value. For instance, in Fig. 1, the context capturing the air pollution is represented by an identifier `Pollution` that can take a value in the domain $[0, 100]$. This context is used to force the selection of a more conservative driving setting when the pollution increases over certain thresholds. This is done by enforcing Validity Formulas (VFs), i.e., propositional formulas associated with a feature that can relate features and attributes with context values. For instance,

in Fig. 1, the feature `Progressive` has associated the VF `Pollution ¡ 50` that forces the feature to be selectable only if the value of the `Pollution` is below 50.

As a configuration of an SPL may be invalidated due to changed values for contextual information and the violation of VFs, the product needs to be reconfigured. In [15], we proposed a first version of a reconfiguration engine that was able to verify if a configuration was valid when given a certain context. In case of invalidity, the reconfiguration engine returned a new valid configuration that was most similar to the original one.

3 Running Example

In this section, we introduce a SPL based on a CES car prototype of Volkswagen that we use to show the feasibility of our approach. The running example is taken from our case study.

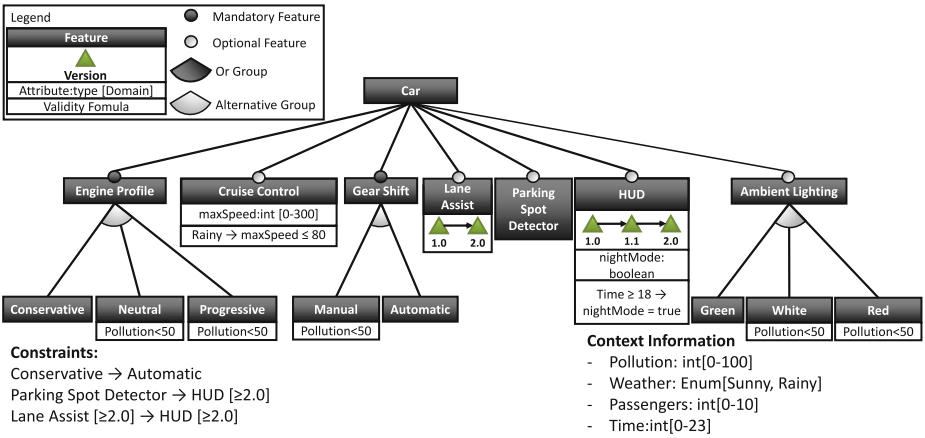

Fig. 1. Hyper feature model and context model for a car SPL.

Figure 1 shows the HFM of our running example. To customize the driving experience itself, it is possible to select different `Engine Profiles` and `Gear Shift` mechanisms as it is already state of the art in current cars. Thus, to drive sporty, it is possible to select a `Progressive Engine Profile` and a `Manual Gear Shift`, whereas a more comfortable driving experience can be achieved by selecting the `Conservative Engine Profile` and an `Automatic Gear Shift`. For drivers who do not want to drive very sporty, but sometimes accelerate to pass other cars, the `Neutral Engine Profile` is a suitable configuration option.

To provide even more comfort, several configurable driver assistance systems are modeled, which are already common in modern cars. For instance, the `Cruise Control` automatically accelerates to a given speed but always keeps a safe distance to cars in front of the own car. However, as some drivers never want

to drive faster than a certain speed, the `Cruise Control` may be customized by defining a maximum speed using the attribute `maxSpeed`, e.g., to conform with the maximum speed permitted by law. As assistance systems are evolving and use new techniques, we model different versions of the `Lane Assist` and the `HUD` which is a heads up display providing additional information to the driver. The `HUD` in versions `1.0` and `1.1` is projecting information alongside the central field of view of the driver such as the current speed limit. To provide even more information and to reduce distraction of the driver, the `HUD` starting with version `2.0` is extended by augmented reality functionality and is able to show information on the windshield which is projected as overlay on the real world. For instance, a navigation system may show an arrow indicating a turn directly on the road at the position of the turn in the real world. Version `2.0` of the `Lane Assist` is using the augmented reality feature of the `HUD 2.0` to indicate the line which was exceeded by the driver. Other features are the `Parking Spot Detector`, which facilitates the parking of the car by indicating free parking spots, and the `Ambient Lighting` of the car, which can be configured to the users' tastes with colors `Green` (useful to influence a more calm and power saving mood), `White` (standard mood) and `Red` (powerful mood).

To allow a more convenient way to configure the car, we introduce the following *partial pre-configurations*:

Sport : `Progressive`, `Manual`, `Red`
Comfort : `Neutral`, `Automatic`, `White`
Eco : `Conservative`, `Automatic`, `Green`

Each of these partial pre-configurations represents a certain type of driver: the sporty, the comfortable, and the eco-minded.

3.1 Contextual Information

The environment may have influence on the software system of the car. For instance, as a result of a changed GPS position of a car, the maximum allowed speed of the `Cruise Control` may be changed to conform to local jurisdiction of a country. To capture the impact of the environment on the software system of the car, we provide the following four types of contextual information.

- `Pollution` captures the amount of contaminants in the air
- `Weather` captures the current weather condition
- `Passengers` captures the number of passengers currently in the car
- `Time` captures the current hour of a day

The values of this context are collected by the car itself (i.e., `Passengers`, `Time`) or by external services (i.e., `Pollution`, `Weather`).

As air pollution is a severe problem in bigger cities, local authorities enforce the usage of the *Eco* partial pre-configuration for each car to keep the air pollution as low as possible. This is captured by making the features contradicting the *Eco* partial pre-configuration only selectable if the `Pollution` is less than 50.

To reduce the frequency of car accidents, local authorities limit the maximum allowed speed depending on the current weather. This is captured by limiting the attribute `maxSpeed` to be less than or equal to 80 km/h if the `Weather` is `Rainy`.

The last context dependent constraint regulates the use of the `nightMode` of the `HUD`. The night mode of electronic devices is a common feature activated to not distract users with the brightness of a display during night. Displays are dimmed and the colors are turned to darker ones. As the `nightMode` is an additional configuration explicitly for the `HUD` it is modeled as an attribute. Furthermore, this allows us to keep the `HUD` customizable, as the `nightMode` could be enabled following the drivers desires. As a safety restriction, a VF is used to enforce the activation of the `nightMode` after 18:00 (6 pm).

3.2 User Preferences

Multiple drivers may want to have different configurations of a car under certain contexts. In this paper, to mimic the preferences of two different drivers, we created two profiles: the *sporty driver* and the *safety-minded* driver.

The *sporty driver* (see Listing 1.1) has a passion for speed and wants to drive very sporty when driving alone. However, when he is accompanied, his highest priority becomes the comfort of the passenger. As he is not familiar with `Manual Gear Shift`, he still prefers the `Automatic` feature. He uses his portable media player in combination with the `HUD`. However, since his media player is very old and not compatible with the latest version of the `HUD`, he prefers the older version of the `Heads Up Display`. As the *sporty driver* often passes other cards when driving fast, he does not like to use the `Lane Assist` as it vibrates when crossing lanes. However, when driving with `Cruise Control`, he loses joy in passing other drivers and, as a result, he wants the support of the `Lane Assist` in this case.

The *safety-minded driver* (see Listing 1.2) is concerned about ecological and safety features. She wants to have all possible assistance systems in the newest version to receive as much support as possible for the driving process, such as the `Parking Spot Detector` or `Cruise Control`. Additionally, as her eyesight has worsened in the last few years, she prefers to enable the `nightMode` of the `HUD` already after 16:00 (4 pm).

4 Modeling Software Product Lines with User Preferences

In this section, we present the two major extensions for the meta model of context-aware SPLs presented in [15]: versions and user profiles.

Modeling Versions
To be able to constrain and target specific versions of the feature, we extend the propositional language to take into account as possible atom a single version of a feature. We use a dedicated construct to restrict the version selection for a feature. Each feature reference can have such a restriction for its versions. In

particular, there are two different types of version restrictions: range restrictions (e.g., $[1.0-2.1]$) and relative restrictions (e.g., $[\geq 2.0]$ or $[1.1]$ specifying an exact version number). Thus, the expression language maintains the same expressive power of [15] being able to use the standard Boolean operators and arithmetic operators for the VFs and CTCs.

Modeling User Profiles

To allow more fine-grained user customization at runtime, we introduce *user profiles* which represent the desires of users. In particular, this means the desire to select or deselect features under certain circumstances. *User profiles* consist of a set of preferences which are linearly ordered according to their importance and priority. Preferences are "weak" or "soft" constraints in the sense that they only have to be satisfied if no other constraint or more important preference contradicts them [26]. Therefore, while CTCs and VFs can potentially forbid the possibility of having admissible valid configuration, this is not possible for preferences.

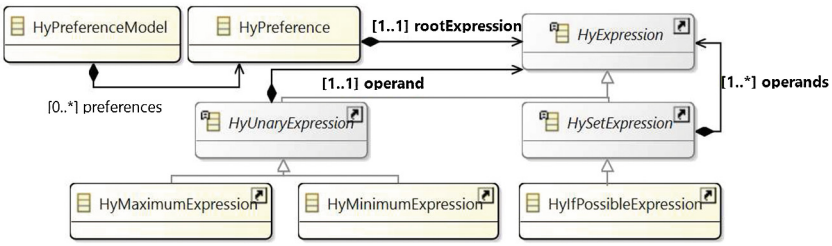

Fig. 2. Preference meta model (excerpt of meta model).

Similarly to VFs, we represent preferences as propositional formulas having as atoms features, versions, attributes, contexts and literal values. The excerpt of the meta model used to formalize the preference is presented in Fig. 2. However, in comparison to VFs, preferences may require a more expressive power. In particular, we consider the wish of users to select as many features as possible from a set of features. As our former expression language does not support expressions to support operations on a set of operands, we extend our expression language with the `HySetExpression`.

A user profile is represented as a `HyPreferenceModel`. Each `HyPreferenceModel` consists of a list of `HyPreferences`. The order of the preferences in the list represents their priority, higher importance first. Moreover, each preference is described by a propositional formula, modeled with a `HyExpression`. To allow for giving a preference on a group of features, we introduce as syntactic sugar the `HyIfPossibleExpression` type, which is a subtype of the `HySetExpression`. Intuitively, this expression is used to state the preference that as many of its operands as possible should be satisfied. For instance, an expression such as `ifPossible({Progressive, Manual})` tries to select both `Progressive` and

Manual if possible, just one of the two if one of them can not be selected or none if both can not be selected. Only one HyIfPossibleExpression expression can be used per preference. Moreover, based on the common preferences usually stated, we also restrict the use of HyIfPossibleExpression to the root of an expression or as a right-hand side of a non nested implication.

Finally, as additional construct to require the preference to maximize or minimize the value of an attribute, we define the HyMaxmimumExpression and HyMinimumExpression, which may be used to state that a given attribute should be set to its maximum or minimum possible value, respectively.

As an example, the preferences of the *sporty driver* and the *safety-minded driver* presented in Sect. 3 can be formalize as presented in Listings 1.1 and 1.2, respectively.

```
1  Passengers ≥ 2 → ifPossible(Comfort)
2  Passengers ≥ 2 → Cruise Control
3  Automatic
4  ifPossible(Sport)
5  Heads Up Display [≤1.1]
6  max(maxSpeed)
7  Cruise Control → Lane Assist
8  ¬Lane Assist
```

Listing 1.1. Preferences of the *sporty driver ordered in decreasing priority.*

```
1  Time ≥ 16 → nightMode = true
2  ifPossible(Eco)
3  Parking Spot Detector
4  Heads Up Display [≥2.0]
5  Cruise Control
6  Lane Assist [≥2.0]
7  Lane Assist
```

Listing 1.2. Preferences of the *safety-minded driver ordered in decreasing priority.*

Note that, as the *sporty driver* does not want to have the Lane Assist assistance system, his profile contains the ¬Lane Assist preference. However, as he wants to select the Lane Assist when using Cruise Control, the preference Cruise Control → Lane Assist has higher priority than the other preference. As far as the *safety-minded driver* is concerned, we would like to remark that since she wants to have a Lane Assist activated, but preferably in version 2.0, we have to model this in two different preferences requiring the activation of the version 2.0 with a higher priority.

5 Reconfiguration Engine

In this section, we describe the contextual reconfiguration tool HyVarRec and explain how the problem of reconfiguration in the presence of preferences is

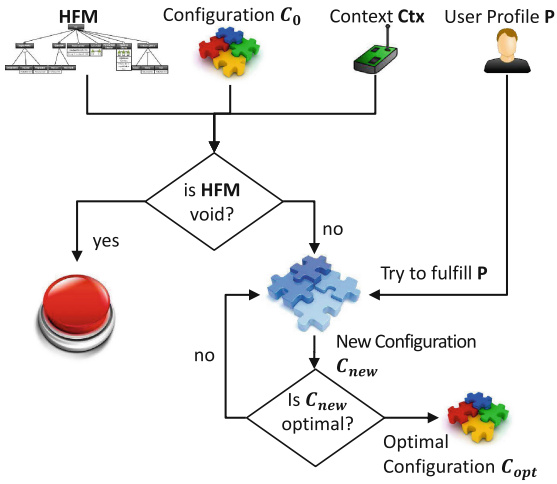

Fig. 3. Work-flow of the contextual reconfigurator.

modeled as a multi-objective optimization problem. We first describe the general execution flow of `HyVarRec` before entering more into the details of the encoding of the constraints and how the FM entities are translated into an optimization problem.

`HyVarRec` requires different sources of input, as depicted in Fig. 3: the HFM, the current configuration C_0 of the remote device, the current values of the contextual information Ctx, and the user profile P. The primary function of the contextual reconfigurator is to provide valid configurations C_{new} for the context Ctx that maximize the preferences of user profile P. In case of two configurations of equal quality regarding the maximization of the user preferences, the one that minimizes the difference between the initial configuration C_0 is provided. This means that `HyVarRec` first tries to minimize the number of feature removals needed to transform C_0 into C_{new} and, later, to maximize the number of attributes which values could be kept the same. Finally, `HyVarRec` outputs the configuration C_{opt}, which is the optimal configuration in the given context, satisfying as many preferences of P as possible.

The reconfiguration engine relies on Constraint Programming (CP). A Constraint Problem consists of a set of variables, each of which is associated with a finite domain of values that it can assume and a set of constraints defining all the admissible assignments of values to variables [13]. In particular, `HyVarRec` tries to solve a Constraint Optimization Problem (COP), a constraint problem where constraints are used to narrow the space of admissible solutions and the goal is to not just find any solution but a solution that minimizes (or maximizes) a specific objective function.

In accordance with the methodology presented in [2], we transform the "standard" feature model part of an HFM into propositional formulas on features. Then, we translate each feature and feature attribute to an integer variable. Whereas features have the domain of $[0, 1]$, for feature attributes, the domain

depends on the type of the attribute. Boolean attributes have a domain of $[0, 1]$ and integer attributes have their specified domain. For attributes which can have values of an enumeration with n literals, the domain is $[0, (n-1)]$. These variables, the propositional formulas of the feature model and the constraints are used as input for the reconfigurator.

Features having more than one version are decomposed into a parent feature having one child feature for every version. A constraint forcing the parent feature to be selected if and only if exactly one of its children features is selected is then added. For instance, for feature `Lane Assist` of the running example in Fig. 1, there are two versions: `1.0` and `2.0`. `Lane Assist` can be therefore encoded in $feature[1]$, its version `1.0` in $feature[2]$ and its version `2.0` in $feature[3]$ with the following requirement.

$$\text{feature}[1] = 1 \leftrightarrow (\text{feature}[2] + \text{feature}[3]) = 1$$

This ensures that exactly one version of the `Lane Assist` is selected if the feature itself is selected. Moreover, if any version is selected, the feature itself has to be selected, as well. Hence, the remaining constraints of the HFM can still use `Lane Assist`. Version-aware constraints can encompass multiple versions defined by the expressions introduced in Sect. 4. Considering the following constraint: $e \rightarrow LaneAssist[\geq 1.0]$ encompasses the versions `1.0` and `2.0`. To determine all involved versions, we use the successor/predecessor relation following the approach introduced in [27]. Afterwards, we translate such an expression to the sum of all encompassed versions.

The expressions of the constraints are translated to their respective textual representation. For instance, a `HyAdditionExpression` with operands *op1* and *op2* is translated to: $op1 + op2$. Also the preferences of the user profiles are encoded as expressions over a set of numerical variables. The only element which can not be translated trivially is the `ifPossible` expression. As we mentioned in Sect. 4, we can use the `isPossible` expression in two situations: as root of an expression or as right-hand side of a non-nested implication. In the first case, we translate it as the sum of its operands, as the optimizer tries to maximize the value of all operands in the sum. For instance, we translate the expression `ifPossible({A,B,C})`, with feature `A` translated to $feature[1]$, `B` to $feature[2]$ and `C` to $feature[3]$, to:

$$(\text{feature}[1] = 1) + (\text{feature}[2] = 1) + (\text{feature}[3] = 1)$$

In the second case, we translate the implication to a sum of implications of the single operands of the `ifPossible` expression. For instance, we translate the expression $e \rightarrow ifPossible(\{A, B\})$, with feature `A` translated to $feature[1]$, `B` to $feature[2]$ to:

$$(e_t \rightarrow feature[1] = 1) + (e_t \rightarrow feature[2] = 1)$$

where e_t is the translation of the `ifPossible`-free expression. In this way the semantics of the `ifPossible` is preserved and the maximal amount of the

`ifPossible` features is selected when the left-hand side of the implication is true. Note that `HyVarRec`automatically transforms Boolean expression into integers, thus, allowing the user to write arithmetic expression over Boolean terms that are treated as 1 if true, 0 otherwise.

To potentially allow `HyVarRec` to support different FM modeling engines, we require the FM with its entities and constraints to be given as a list of propositional constraints as described before. In contrast, the preferences are given as arithmetic expressions or preferences to maximize or minimize a given attribute. The EBNF grammar defining the propositional constraint and the preferences is formally defined in https://github.com/HyVar/hyvar-rec/blob/master/SpecificationGrammar/SpecificationGrammar.g4 following the ANTLR conventions.[1]

`HyVarRec` is an anytime solver: it first determines a valid configuration, if one exists, and then proceeds in finding those that satisfy more preferences or are more similar to the initial configuration. The optimization proceeds in phases: when a valid configuration is found, additional constraints are imposed to search only valid configurations that satisfy additional user preferences. The process terminates when no other valid configuration can be found, meaning that the last determined configuration is the one maximizing the preferences and similarities with the initial configuration. In this way, `HyVarRec` allows users to interrupt the computation as soon as a good-enough configuration was found even though it might not be the best possible configuration. This may be extremely useful when handling large and complex SPLs and there are strong limitations on the computation and time resources to get a valid configuration as with Dynamic Software Product Lines (DSPLs).

`HyVarRec` relies on *MiniSearch* [24] to conduct the search and on *MiniZinc* [18] for the definition of the constraints. `HyVarRec` uses the Gecode constraint solver [10], the default solver adopted by MiniSearch because it implements natively the incremental API needed by MiniSearch to post additional constraint without restarting the solving process from scratch. However, other MiniZinc solvers that do not support the recently defined MiniZinc incremental API could still be used at the price of restarting their engine during the optimization process. Hence, the majority of the other state of the art solvers (e.g., the Java-based Choco [22], the SAT based MiniSatid [16]) can be integrated in `HyVarRec` by relying directly on the MiniSearch capabilities.

The output of `HyVarRec` is a sequence a JSON objects each of which represents a configuration satisfying more preferences or more similar to the initial one. When the entire search space is explored and the optimal configuration is found, we translate the respective configuration to an output format and notify the user that this is the optimal configuration. The schema of the JSON output format of `HyVarRec` is formalized in https://github.com/HyVar/hyvar-rec/blob/master/spec/hyvar_output_schema.json.

The main difference of `HyVarRec` w.r.t. its previous version is the introduction of the language to express the preferences. This has an impact also on the

[1] ANTLR (ANother Tool for Language Recognition) - http://www.antlr.org/.

formalization of the COP problem due to the fact that the metrics to optimize are established by the preferences. Moreover, to ease it use by external tools, `HyVarRec` now requires a unique JSON object that unifies all the input entities in a unique representation.

`HyVarRec` is written in python, open source, and freely available from https://github.com/HyVar/hyvar-rec.

6 Case Study

In this section we show the feasibility of our approach by applying `HyVarRec` on the car SPL detailed in Sect. 3.

Imagine that the *sporty driver* leaves home in the morning driving alone. In the residential areas, the measurement of the `Pollution` level is below 50. Due to his preferences, the car is running on `Automatic Gear Shift` and `Progressive Engine Profile`. The `Ambient Lighting` is selected and configured to `Red` and he is using the `HUD` in version `1.1` to connect his media player. As he is not using the `Cruise Control`, the `Lane Assist` is disabled.

On the way, he picks up a friend, which causes the *Comfort* partial pre-configuration to be prioritized over *Sport* as he is no longer alone in the car. Consequently, the car configuration is changed to `Neutral Engine Profile` and `White Ambient Lighting`. The `Automatic Gear Shift` is kept but the `Cruise Control` gets activated. However, the `Weather` is `Sunny` with low traffic on the highway. Therefore, the `maxSpeed` of the `Cruise Control` allows him to speed up to the maximum speed of 300 km/h. As the `Cruise Control` was activated, the `Lane Assist` is enabled too, but uses version `1.0` as the `HUD` is activated in version `1.1`.

The car enters the city and the `Pollution` increases to 75. This contextual change enforces the car to reconfigure to the *Eco* partial pre-configuration. Therefore, the car now runs on `Conservative` engine mode with `Green Ambient Lighting`. As he prefers an older version of the `HUD`, the `Parking Spot Detector` is not enabled when he arrives at his destination.

Assume that the time is now 15:00 (3 pm) and the friend, who is the `safety-minded driver`, is driving the car home. As she prefers the *Eco* partial pre-configuration, the car remains in `Conservative` engine mode with `Green Ambient Lighting` and `Automatic Gear Shift`. Additionally, the `HUD` is updated to the newest version and, based on her preferences, the `Parking Spot Detector` is enabled with augmented reality support of the `HUD`. For a convenient ride, the `Cruise Control` is enabled, as well as the newest version of the `Lane Assist`.

During the ride back, the time passes 16:00 (4 pm) and it starts to rain. These two contextual changes cause the car to reconfigure. The maximum speed of `Cruise Control` is reconfigured to 80 km/h. Moreover, her preference causes the `nightMode` of the `HUD` to be switched on. The rain causes the `Pollution` level to drop to below 50, which would potentially enable a new reconfiguration of the car, but as she prefers the `Eco` profile, the selected sub-features of `Engine Profile`, `Gear Shift`, and `Ambient Lightning` remain unchanged.

Using our implementation, we were able to model all necessary features, constraints, contextual information, validity formulas, and preferences contained in the case study. Furthermore, we were able to provide the modeled case study as input for `HyVarRec`, generating the first initial configuration by hand and simulate the effects of the context changes. The resulting configurations match the scenario we described in this section, thus showing the suitability of our methodology to provide optimal configurations with respect to the user preferences.

`HyVarRec` has recently been adopted as the reconfiguration engine in an integrated toolchain to develop, deploy and reconfigure SPLs in an industrial setting [4].

7 Related Work

Preferences are a widely studied topic in the field of social sciences and artificial intelligence. Preferences are often assumed to be given by users. However, there are approaches where preferences are derived by inpsecting the history or use of an application or software. For instance, in [30], preferences are learned by analyzing the history of the different users, creating a profile of preferences for every user. Similarly, in [1], user preferences for web search engine optimization are learned from user behavior. In particular, preferences are widely investigated in the domain of product recommendation systems, e.g., in [29], which deals with preferences for music recommendation systems. While these approaches are extremely interesting and may prove useful for our future work, in this work, we assume that users elicit their preferences explicit.

Preferences are also well studied in a qualitative or quantitative way in the field of decision theory [8] and in Constraint Programming [26]. Preferences are often incorporated in COPs [3,7] and are also denoted as soft constraints [26]. `HyVarRec` is built on top of these approaches and it exploits all the experience accumulated in the Constraint Programming community to speed up the search of the configuration maximizing user preferences.

In the domain of SPLs, multiple work has been done on modeling contextual information to be used in software systems. Several authors introduce an additional feature model which consists of the contextual information modeled as features [9,11,28]. The relation of context and features is modeled as CTCs between the original feature model and the context feature model. However, the expressiveness of feature models is relatively coarse grained. Thus, for contexts which may have a big value domain, e.g., the `Pollution` or the `Time` contexts of our running example, features are not suitable to model each of these values. Additionally, for the sake of separation of concerns, a different and suitable notation for contextual information seems sensible instead of using feature models for it. To the best of our knowledge, none of these methodologies consider the reconfiguration of the SPL based on user preferences.

Other related works are [5,14], which propose approaches for reconfiguration similar to ours encompassing contextual information in the feature model. However, differently from us, for the reconfiguration, they explicitly model *triggers*

which are fired based on values of the contextual information and have composition rules which model how the SPL has to be reconfigured. Additionally, they do not consider user preferences. The afore mentioned approaches model the influence of context on the feature selection directly, i.e., prescribing the selection of a feature in a certain context. In our approach, we model in which contexts a certain feature is selectable, allowing a better integration with user preferences.

In [20] a concept for DSPLs encompassing contextual information and user preferences is introduced. A *Decision Maker* decides if and how the DSPL has to be reconfigured. However, no notion of user preferences or contextual information is given, thus making unclear how to model and incorporated preferences with the DSPL. Moreover, there are no details on how the *Decision Maker* processes these information.

Some approaches like [17,19] assign values or costs to features that allows during the (re-)configuration to optimize certain properties (e.g., costs and productivity) to create a best configuration considering a multi-dimensional optimization problem. However, to simulate user preferences, these approaches require each user to specify values and cost for each feature which is not suitable for large models. Moreover, optimizing certain properties severely differentiates from trying to fulfill an ordered set of preferences.

8 Conclusion

In this paper, we introduced an approach to allow users to influence the reconfiguration of a context-aware SPL based on their preferences. To this end, we extended our preliminary work [15] to propose a formalism that can capture, in a concise way, not only contextual information but also user preferences as weak or soft constraints [26]. Conceptually, user preferences differ from (hard) constraints in the sense that they specify desired features of users and are not mandatory to be satisfied. Therefore, users can customize their SPL based on their desires and, additionally, can make them dependent on the current context. The preferences are summarized in *user profiles* as a linearly ordered set, ordered by priority.

Furthermore, we extended the context-aware reconfiguration engine `HyVarRec` to be able to consider user profiles. The new version of `HyVarRec` tries to maximize the number of satisfied preferences contained in the current profile while generating a valid configuration. We showed the feasibility of our methodology by modeling a realistic SPL of a customizable car, encompassing contextual information. We modeled two different user profiles and defined a scenario in which the car is influenced by contextual changes. We simulated the scenario by utilizing `HyVarRec` with the respective input. `HyVarRec` successfully created new valid configurations and maximized the given user profiles.

For future work, we are interested in extending our concepts by means of capturing evolution of the Feature Model (FM) and the user profiles and to address some limitations of the current model. For instance, we would like to allow the possibility to use context to force directly the selection of a particular

feature. Additionally, we are interested in creating user profiles by learning from user behavior or understanding their polices [23] and how they can be enforced. Finally, we are interested in evaluating our approach using the toolchain presented in [4], analyzing the scalability of our approach considering varying and larger FMs.

Acknowledgments. This work was partially supported by the DFG (German Research Foundation) under grant SCHA1635/2-2 and by the European Commission within the project HyVar (grant agreement H2020-644298).

References

1. Agichtein, E., Brill, E., Dumais, S., Ragno, R.: Learning user interaction models for predicting web search result preferences. In: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2006. ACM, New York (2006)
2. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: Pastor, O., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005). doi:10.1007/11431855_34
3. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: Preference-based constrained optimization with CP-nets. Comput. Intell. **20**, 137–157 (2004)
4. Chesta, C., et al.: A toolchain for delta-oriented modeling of software product lines. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 497–511. Springer, Cham (2016)
5. da Silva Costa, P.A., Marinho, F.G., de Castro Andrade, R.M., Oliveira, T.: Fixture - A tool for automatic inconsistencies detection in context-aware SPL. In: ICEIS (2015)
6. Darryll Harrison, W.G.: CES 2016: Volkswagen brings gesture control to mass production with the E-Golf Touch (2016). http://media.vw.com/release/1123/
7. Domshlak, C., Rossi, F., Venable, K.B., Walsh, T.: Reasoning about soft constraints, conditional preferences: complexity results and approximation techniques. arXiv (2009)
8. Doyle, J., Thomason, R.H.: Background to qualitative decision theory. AI Mag. **20**(2), 55–68 (1999)
9. Fernandes, P., Werner, C., Murta, L.: Feature modeling for context-aware software product lines. In: Seke (2008)
10. GECODE (2015). http://www.gecode.org/
11. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: SPLC IEEE Computer Society (2008)
12. Kang, K.: Analysis, Feature-oriented Domain (FODA): Feasibility Study; Technical report CMU/SEI-90-TR-21 - ESD-90-TR-222. Software Engineering Inst., Carnegie Mellon Univ. (1990)
13. Mackworth, A.K.: Consistency in networks of relations. Artif. Intell. **8**(1), 99–118 (1977)
14. Marinho, F.G., Andrade, R.M.C., Werner, C.: A verification mechanism of feature models for mobile and context-aware software product lines. In: Software Components, Architectures and Reuse (SBCARS) (2011)

15. Mauro, J., Nieke, M., Seidl, C., Yu, I.C.: Context aware reconfiguration in software product lines. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems - VaMoS 2016 (2016)
16. Minisatid. https://github.com/broesdecat/Minisatid
17. Murashkin, A., Antkiewicz, M., Rayside, D., Czarnecki, K.: Visualization and exploration of optimal variants in product line engineering. In: Proceedings of the 17th International Software Product Line Conference, SPLC 2013. ACM, New York (2013)
18. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74970-7_38
19. Ochoa, L., González-Rojas, O., Thüm, T.: Using decision rules for solving conflicts in extended feature models. In: Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering, SLE. ACM, New York (2015)
20. Parra, C., Blanc, X., Duchien, L.: Context awareness for dynamic service-oriented product lines. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009. Carnegie Mellon University, Pittsburgh (2009)
21. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations: Principles and Techniques. Springer, New York (2005)
22. Prud'homme, C., Fages, J.-G., Lorca, X.: Choco3 Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2014)
23. Reiff-Marganiec, S.: A structured approach to VO reconfigurations through policies. In: Proceedings Third Workshop on Formal Aspects of Virtual Organisations. EPTCS, FAVO 2011, Sao Paolo, Brazil, 18 October 2011, vol. 83, pp. 22–31 (2011)
24. Rendl, A., Guns, T., Stuckey, P.J., Tack, G.: MiniSearch: a solver-independent meta-search language for MiniZinc. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 376–392. Springer, Heidelberg (2015). doi:10.1007/978-3-319-23219-5_27
25. Robarts, S.: Volkswagen's Golf R touch concept shows off the car cockpit of the future (2015). http://www.gizmag.com/volkswagen-golf-r-touch/35472/
26. Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Elsevier (2006)
27. Seidl, C., Schaefer, I., Aßmann, U.: Capturing variability in space and time with hyper feature models. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems - VaMoS 2014 (2014)
28. Ubayashi, N., Nakajima, S.: Context-aware feature-oriented modeling with an aspect extension of VDM. In: Proceedings of the ACM Symposium on Applied Computing, SAC 2007. ACM, New York (2007)
29. Yoshii, K., Goto, M., Komatani, K., Ogata, T., Okuno, H.G.: Hybrid collaborative and content-based music recommendation using probabilistic model with latent user preferences. In: ISMIR, vol. 6 (2006)
30. Young, S., Hong, J.-H., Kim, T.-S.: A formal model for user preference. In: Proceedings of IEEE International Conference on Data Mining (2002)