

Context-aware reconfiguration in evolving software product lines



Jacopo Mauro ^{a,*}, Michael Nieke ^b, Christoph Seidl ^b, Ingrid Chieh Yu ^c

^a University of Southern Denmark, Denmark

^b Technische Universität Braunschweig, Germany

^c University of Oslo, Norway

ARTICLE INFO

Article history:

Received 15 August 2017

Received in revised form 4 May 2018

Accepted 4 May 2018

Available online 9 May 2018

Keywords:

SPL

Optimization

Preferences

Context-aware

ABSTRACT

Software Product Lines (SPLs) are a mechanism for large-scale reuse where families of related software systems are represented in terms of commonalities and variabilities, e.g., using Feature Models (FMs). While FMs define all possible configurations of an SPL, when considering dynamic SPLs and environmental conditions, not every possible configuration may be valid in all possible contexts. A change in the environment may, therefore, require the reconfiguration of the SPL.

With common modeling methodologies, it is not possible to capture the correlation of configuration options, contextual influences, user customizations, and evolution. In this paper, we remedy this problem by first defining a novel framework that allows modeling customizable evolving context-aware SPLs. We then provide a reconfiguration engine that computes how the current configuration needs to be reconfigured when the context is altered, the user preferences changed or the SPL artifacts are evolved and the configuration is adapted to reflect the evolved artifacts.

© 2018 Elsevier B.V. All rights reserved.

1. Introduction

Software Product Line (SPL) are a technology for large-scale software reuse for a set of closely related software systems [1], which allows companies to customize their software systems through configuration (cf. Section 2). In addition, the option for after deployment software customization has gained importance in many domains, e.g., in the automotive domain where a purchased car can be tailored to accommodate the wishes of multiple drivers. For instance, the Volkswagen AG presented a prototypical car at the Consumer Electronic Show (CES) 2015 which supports several user customizations at runtime, such as the change of the engine profile, allowing a more powerful or power-saving engine performance [2,3] depending on who is driving the car. Hence, such *user preferences* also form an elementary part in configuring a software system. However, a default configuration is needed in order to avoid imposing the users to make decisions about a full configuration selection.

Apart from user preferences, there may be further influences on a system's current behavior. For instance, a car may change its settings based on the current weather conditions. These factors are generally referred to as the system operational *context* or *environment*, which, due to its possible effect on functionality, has to be considered in the customized configuration selection process as well.

* Corresponding author.

E-mail address: mauro@imada.sdu.dk (J. Mauro).

In common SPL engineering, creating a configuration of a system is an explicit and mostly manual procedure that is usually performed before building the software system. However, over the course of time, SPLs are subject to evolution [4], which require engineers to change features, constraints, and their realizations to create new *versions* of the software. This may cause new incompatibilities and it may change the set of valid configurations of an SPL. Therefore, the evolved artifacts need to be incorporated in the configuration process.

In previous work, we have made various individual contributions in the field of context-aware evolving SPLs:

- We extended the common notion of SPLs with context, allowing the possibility to relate them to features via logical formulas [5].
- We enabled SPL users to express preferences over valid configurations, facilitating user customization of SPLs [6].
- We devised a modeling notation that captures the evolution of configuration knowledge as a first-class entity [7,8].
- We developed DARWINSPL, i.e., a GUI-based modeling tool suite to define, use, and evolve context-aware SPLs [8].
- We developed HyVarRec, an efficient reconfiguration engine that determines valid alternative configurations if a change of context values invalidated the currently deployed configuration or evolution changed the set of valid configurations [5,6].

To the best of our knowledge, there is no common framework able to encompass all the aspects of SPL modeling taking into account the evolution, context, and user preferences to allow modeling of an SPL and its automatic reconfiguration. In this work, we combine the individual contributions of our previous work to form an integrated framework for modeling and reconfiguration of SPLs that encompasses evolution, context, and user preferences to address the shortcomings of existing approaches. As part of this consolidation, we have consequently enabled all modeling notations for contextual information, cross-tree constraint (CTCs) and Validity Formula (VFs) to allow them to capture changes associated with evolution as first-class entities. In addition, we have extended our modeling technique to not only respect user preferences but also manufacturer preferences, which may potentially be conflicting. Finally, we have improved the implementation of our reconfiguration engine HyVarRec to allow for incremental solving addition or removal of constraints. Hence, the contribution of this paper is the consolidation of our previous work, an integration of our concepts and tools to a common framework as well as an improvement of our modeling techniques and our reconfiguration engine.

The remainder of this paper is structured as follows: Section 2 establishes the background on SP engineering and Feature Model (FMs). Section 3 introduces a running example that we use throughout the paper to illustrate our concepts. Section 4 presents our conceptual contributions for modeling evolving context-aware SPLs with preferences. Section 5 elaborates on our practical contribution in the form of the reconfiguration engine HyVarRec. Section 6 presents our case study from the automotive sector to demonstrate the feasibility of our contributions and the integrated framework. Section 7 describes the setup and results of a benchmark we conducted to show scalability of our reconfiguration engine HyVarRec. Finally, Section 8 discusses approaches related to ours before Section 9 closes with a conclusion and an outlook on future work.

2. Background

In this section, we give a brief overview of the basic concepts forming the basis of our work.

2.1. SPL and FM

Software Product Line (SPL) engineering [1] is a method for large-scale reuse within a family of closely related software systems whose members are similar to a large extent but differ in specific parts. At the core of SPL engineering is the modeling of common and variable parts of software systems. On the conceptual side, common and, especially, variable parts are described in terms of *features*, which represent configurable functionality of a system [9]. Optimally, each feature represents a user-visible functionality. To represent the relation between features, a *variability model* can be employed. Among the most popular variability models are *Feature Model (FMs)* [9]. FMs structure features hierarchically in a tree-like notation. Feature diagrams are common visualizations of feature diagrams. Fig. 1 depicts an exemplary feature diagram. Each feature model has exactly one root feature (in the example `Root`) which must always be selected. Each feature can only be selected if its respective parent is selected. Moreover, features have a type: either they are *optional* (e.g., `Feature A`), stating that they *may be* selected if their parent is selected, or *mandatory* (e.g., `Feature B`), stating that they *must be* selected if their parent is selected. Multiple *optional* features can be structured in groups, which also have a type: an *or* group (e.g., `Feature C` and `Feature D`) means that at least one of the group's features has to be selected if the parent is selected, whereas an *alternative* group (e.g., `Feature E` and `Feature F`) requires the selection of exactly one of its features. Each feature model can be represented using a set of constraints with features as variables and encoding the tree-hierarchy as constraints [10]. Thus, conventional constraint solvers can be utilized to reason on feature models.

Additional to configuration options provided by features, feature *attributes* permit specification of more fine-grained configuration options [11]. Each feature can be annotated with a set of attributes. In addition, attributes may be assigned a type (e.g., integer, string, or a pre-defined enumeration) so that the respective values can only stem from the types' domains. As an example, in Fig. 2, the feature `Ambient Lighting` has an attribute representing the light's color. This attribute has an enumeration type that forces it to be one of the values `Green`, `White`, and `Red`.

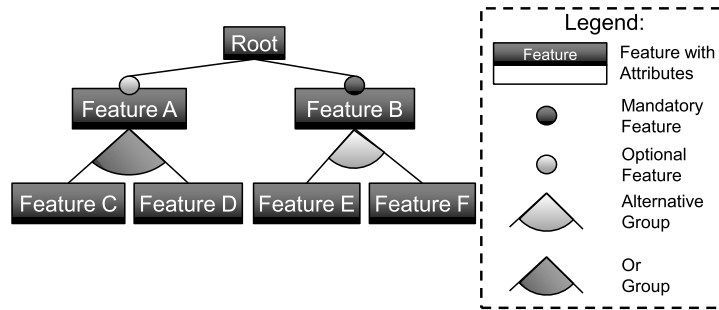


Fig. 1. Example of a feature model.

As not each relation between features can be expressed using the hierarchical structure of FMs, CTCs are used. CTCs are specified as Boolean formulas on features and expressions on attributes. Common concepts of CTCs can be extended to express version restrictions on features by e.g., using a version-aware CTC language [12].

On the implementation side, features are realized using *realization artifacts*, such as code or documentation artifacts. Similar to other software systems, SPLs evolve [4] and so do the implementation artifacts of features. In this paper, we use *feature versions* developed by Seidl et al. [12] to capture the different evolution states of features. Features can have arbitrary versions arranged along a branching development history, where a change in version represents modifications to a feature's implementation to realize evolved functionality. In Fig. 2, an example of versions can be seen by looking at the Connection Gateway feature that comes in two versions: 1.0 and 2.0.

A *configuration* of an FM comprises a set of selected or deselected features and feature versions along with value assignments for attributes of the selected features. A configuration is valid if it does not contradict any of the constraints imposed by the FM, including the structural constraints as well as the CTCs. Hence, a valid configuration describes one member of the software family on a conceptual level without regard to its implementation. The space of all valid configurations is described by the constraints of the feature model and the CTCs. In contrast, a *variant* is the implementation of such a valid configuration, thus describing that same member of the software family on realization level.

2.2. SAT, SMT, and CP

To analyze a feature model and prove properties such as the existence of a valid configuration, different techniques have been proposed and used. For a detailed survey of these techniques, we refer the interested reader to the comprehensive survey of Benavides et al. [13]. In the following, we summarize just three of the most widely used one, namely Propositional Satisfiability (SAT), Satisfiability Modulo Theory (SMT), and Constraint Programming (CP).

The first and probably the most used technology to analyze feature models is SAT. A feature model is encoded into a propositional formula by associating every feature with a variable, each relationship between features or constraint into one or more small formulas and, usually, an additional constraint is added requiring to assign to true the variable that represents the root. The analysis of the feature model can therefore be conducted by using SAT solvers [14], tools that take as input a propositional formula and determines if the formula is satisfiable.

SMT is a generalization of Boolean SAT formulas in which variables are replaced by predicates from a variety of underlying theories. Differently from SAT solvers, they can natively support integer variables and arithmetic over them, thus, making it easier to encode the feature model when they have attributes or arithmetic constraints. For this reason, SMT solvers are used to analyze FM supporting more fine grained configuration options like attributes or versions.

Another similar technique to SMT is Constraint Programming (CP), which allows to express complex relations in form of constraints to be satisfied. CP allows to model and solve Constraint Satisfaction Problems (CSPs) as well as Constraint Optimization Problems (COPs) [15]. Solving a CSP means finding a solution that satisfies all the constraints of the problem, while a COP is a generalized CSP where the goal is to find a solution that minimizes or maximizes an objective function. Similarly to what is done with SAT or SMTs, constraint programming has been used for the analyses on feature models by translating the feature model into a CSP and then use CP solvers to check their satisfiability.

Note that SAT, SMT, and CP solvers tackle problems that are NP-hard [16]. Due to the nature of these problems, there is no approach that dominates each other: often the performance may vary depending on the instance to be solved and the algorithm and heuristics used by the chosen solver. With the exception of few logics partially supported by some SMT solvers, from a theoretical point of view, these approaches all have the same computational power and, therefore, can be used interchangeably for analyzing the properties of feature models.

3. Running example

In this section, we introduce a running example that was obtained from a real demonstrator from our industry partner in the automotive domain and simplified for presentation purposes. We will use this as running example throughout the

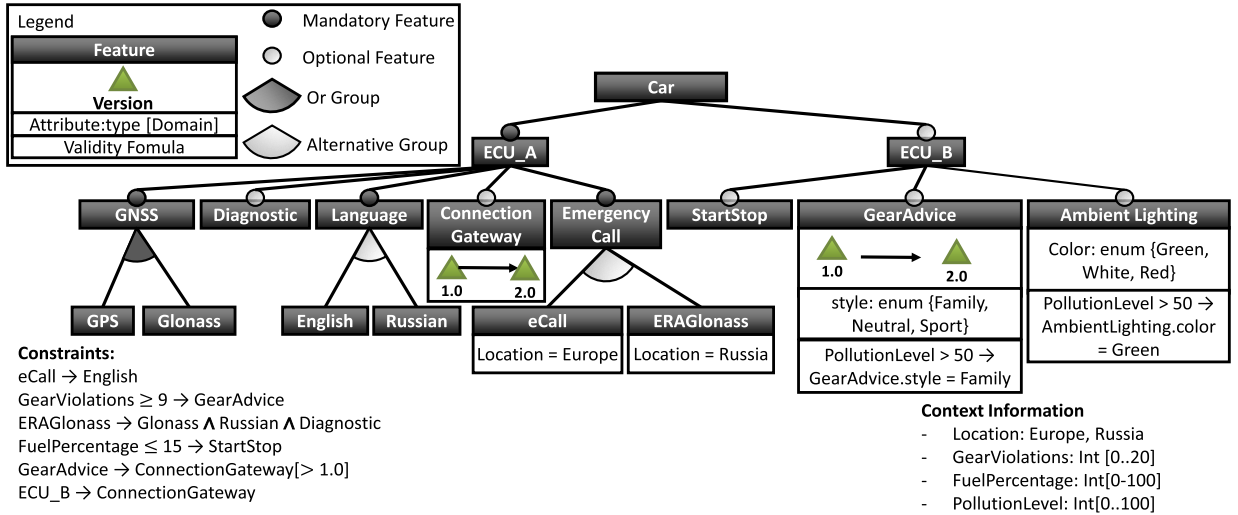


Fig. 2. Hybrid Feature Model for a car SPL.

paper to illustrate the concepts we introduce. In particular, we introduce a Hybrid Feature Model (HyFM), which is a Hyper Feature Model (HFM) [12] extended with feature attributes and combined with contextual information, relations between contextual information and features [5]. Fig. 2 illustrates the HyFM for a car used in a software supplier company for car manufacturers. For simplicity, here we give the final overview of the HyFM that was obtained via different evolution steps (cf. Section 6).

ECU_A represents an electronic control unit (ECU) being responsible for emergency call systems of a car – a system that, upon an accident, sets off an automated distress call that includes the car's satellite position and some relevant additional data. The car supports different emergency call systems – one for Europe (eCall), and one for Russia (ERAGlomass). Glomass is used as positioning service in Russia and more additional diagnostic data is sent. Therefore, the feature ERA-Glomass requires the features Glomass, Russian, and Diagnostic. For the eCall feature, GPS or Glomass can be used but it requires the English language feature. Feature ConnectionGateway provides functionality for other ECUs to communicate with ECU_A.

To extend the driving experience and support drivers, it is possible to select AmbientLighting, GearAdvice, and StartStop. These are provided by the optional electronic control unit ECU_B. Feature GearAdvice suggests the gear that should be used, according to a pre-defined driving style, to maximize either performance or fuel efficiency. The selectable styles are Family, Sport or Neutral. Finally, feature AmbientLighting determines the color of the light in the cabin of the car, i.e., red, green or white.

3.1. Contextual information

To model the impact of the environment on the software system of the car, we provide four sets of contextual information:

- Pollution, which captures the number of contaminants in the air,
- Location, which captures the current position of the car,
- GearViolation, which captures the number of times the shifts used by the driver were not optimal,
- FuelPercentage, which captures the remaining fuel level.

The concrete values for these contexts have different impacts on the features of the car. Depending on the current location of the car, it will have either eCall (in Europe) or ERAGlomass (in Russia). If the driver uses the shifts in a wrong way too many times (i.e., GearViolations > 9), the car automatically selects GearAdvice. Additionally, if the air pollution outside is too high, the Family style for the GearAdvice is enforced to reduce the car exhaust. Also, the AmbientLighting color is set to Green, trying to influence the driver to have a less aggressive driving style. StartStop is automatically selected when the fuel percentage is less than 15%.

3.2. User and manufacturer preferences

To allow customizability, it is possible for users (i.e., drivers) to define their own preferences incorporated in the reconfiguration process. Moreover, the car manufacturers can define their own preferences as a sensible default. For example, the manufacturer may set a default to use the latest version of ConnectionGateway while the user may want to keep using

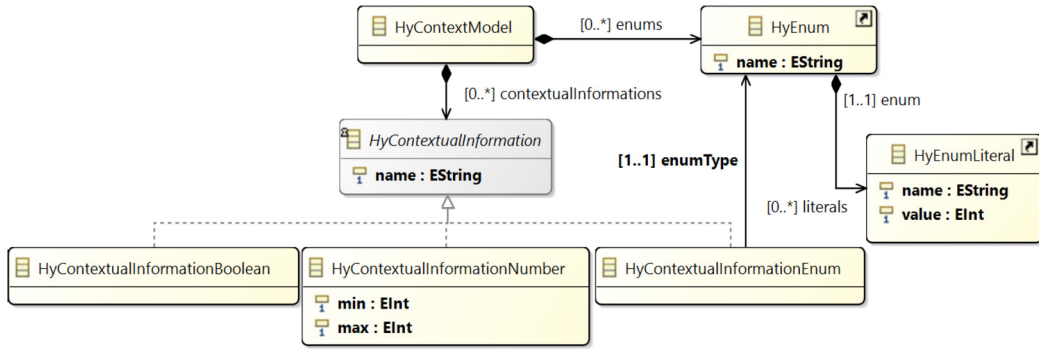


Fig. 3. Metamodel for contextual information.

the old version as it is compatible with his/her smart devices. In this example, we assume that the manufacturer has the following prioritized default preferences:

1. GearAdvice[2.0]
2. GearAdvice
3. ConnectionGateway → ConnectionGateway[2.0]
4. AmbientLighting → AmbientLighting.Color = White

As a default, the GearAdvice should be selected in its newest version ([2.0]). However, if the newest version is not selectable, the manufacturer suggests having the GearAdvice in any version. Moreover, the newest version of ConnectionGateway should be selected if possible, which ensures the best communication between all ECUs of the car. To avoid being too intrusive, the AmbientLighting's color is set to White as default.

As exemplary preferences of a sporty driver, we consider the following.

1. \neg GearAdvice
2. GearAdvice → GearAdvice.style = Sport
3. AmbientLighting \wedge AmbientLighting.Color = Red
4. ConnectionGateway → ConnectionGateway[1.0]

As the sporty driver does not want to be influenced by the car's gear advice, he wants to have the GearAdvice feature to be deselected. If this is not possible, at least he wants it to use the Sport style. To match the fast driving style, the AmbientLighting should always be selected with its color set to red. Moreover, as the driver has an old smartphone not being compatible with newer versions of ConnectionGateway, version [1.0] of the ConnectionGateway should be selected if the feature itself is selectable.

We would like to remark that while the manufacturer and the user preferences use the same formalism, it is important to keep them separate because conceptually they are expressed by two different entities, namely the developer of the HyFM and its user, and because the user preferences have higher priority than manufacturer preferences that are used to specify only defaults to be used when the user does not express his or her customization wishes.

In Section 6, we will show the evolution of the presented HyFM where the car manufacturer starts with only ECU_A and later extends the HyFM with ECU_B. We will explain how this evolution together with the current car configuration triggers the reconfiguration engine. The reconfigurator, together with contextual information as well as both user and manufacturer preferences, calculates a new configuration for the car.

4. Modeling evolving context-aware software product line with preferences

In this section, we introduce how HyFMs can be modeled. In particular, in Section 4.1, we present how relevant environmental information can be defined by extending the notion of standard FM, in Section 4.2 we detail how users can customize a configuration via preferences, and in Section 4.3 as well as Section 4.4, we show how to model the context-aware SPL evolution.

4.1. Modeling context-aware SPLs

For SPLs to adapt to their environment, environmental information needs to be captured and its impact on the SPL needs to be defined.

Fig. 3 shows the *contextual information* metamodel defined to capture environmental information. A context is an object with a name that takes a value in a given domain. In particular, we considered three types of contextual information:

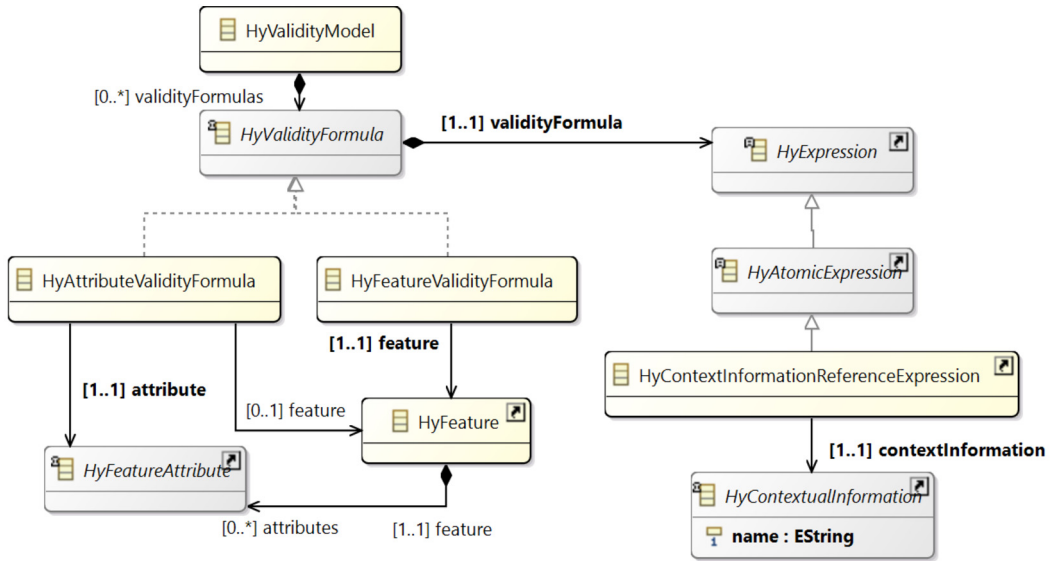


Fig. 4. Metamodel for context-aware expression language and validity formulas.

Boolean (`HyContextualInformationBoolean`), integer (`HyContextualInformationNumber`), and enumeration (`HyContextualInformationEnum`). For an integer context, a maximum and minimum (positive or negative) value need to be specified to define its domain. For enumeration types, a set of possible literals (`HyEnumLiteral`) needs to be defined.

Fig. 2 shows at the bottom right the context information considered in our running example. In particular, the car's geographical location is modeled as enumeration labeled `Location`, with two literals representing the car being located in `Europe` or `Russia`. To be able to analyze the driver's shifting behavior, we modeled the `GearViolation` context information as an integer context, representing how many times the driver shifts too late. Moreover, we captured the car's current fuel level with `FuelPercentage` and the air pollution outside the car with `PollutionLevel` – both as integer contexts with minimum 0 and maximum 100.

After capturing the contextual information, its impact on the SPL needs to be modeled, defining in which situations an SPL has to be reconfigured. This is done by two means: contextual CTCs and *Validity Formula* (VFs) stating in which context a feature is selectable.

For contextual CTCs, a generic expression language is used. This language is able to express arbitrary propositional formulas using features, feature attributes and contextual information as variables. Available logical operators are: \wedge , \vee , \rightarrow , \leftrightarrow and \neg . Moreover, feature attribute domains can be constrained by using standard comparison operators such as $=$, \neq , $>$, \geq , $<$ and \leq . All these expressions can be nested arbitrarily using parentheses. The difference between a contextual CTC and a standard one is that in the former, context terms are used while in the latter it is only possible to use terms representing features, attributes, and versions. To analyze values of contextual information, the same operators as for attribute values can be used. Using this extended expression language, it is possible to define arbitrary contextual CTCs and react to contextual changes. For instance, in our running example in Fig. 2, we define a contextual CTC expressing that the `GearAdvice` feature has to be selected as soon as the car driver shifted more than nine times too late. Another contextual CTC enforces the `StartStop` feature to be selected if the fuel level is below 15%.

As contextual CTCs are defined similar to standard CTCs and they can even be defined together in one formula, it is hard to determine the contextual impact on the SPL. To make this impact directly visible for the affected features, the concept of *Validity Formula* (VFs) [5] has been introduced. VFs, modeled in Fig. 4, define in which context a certain feature is selectable. Thus, a VF is referencing a feature or a feature attribute and consists of a propositional formula, defined using our contextual expression language. This propositional formula defines the context under which the feature can be selected. For instance, in Fig. 2, VFs are defined for `eCall` and `EraGlonass`. These VFs restrict `eCall` to only be selectable if the car is in Europe and `EraGlonass` to be selectable if the car is located in Russia. This way, the impact of the context on the features `eCall` and `EraGlonass` is directly visible to developers and users of the SPL.

4.2. Modeling profiles

Section 4.1 introduces concepts for modeling the impact of context on an SPL. This methodology allows SPL developers to model this impact when developing the product line. However, users are accustomed to customizing the behavior of their systems and could potentially have preferences on the final configuration. As an example, as mentioned in Section 3, the sporty driver wants to have `GearAdvice` deselected if possible.

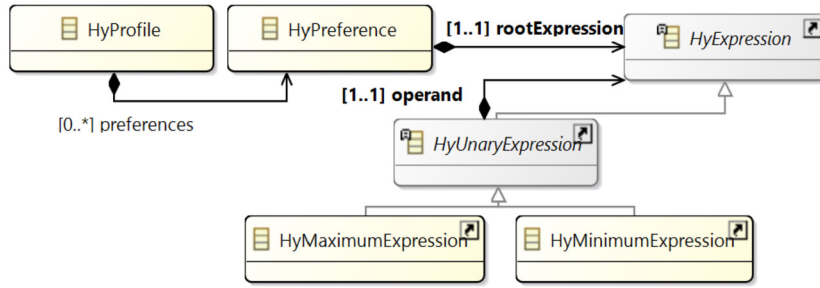


Fig. 5. Metamodel for profiles and preferences.

To allow users to customize the reconfiguration behavior of their system, we introduced the concept of *user profiles* [17]. As users can easily model wrong reconfiguration rules, e.g., contradicting other necessary reconfiguration rules or FM constraints, we also introduce the concept of *preferences*. 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 [15]. Therefore, while CTCs and VFs can potentially forbid the possibility of having admissible valid configurations, this is not possible for preferences.

Similar to VFs, preferences are represented as propositional formulas (*HyExpression*) having features, feature attributes, contextual information and literal values as atoms. Fig. 5 shows an excerpt of the metamodel formalizing preferences. In comparison to VFs and CTCs, preferences may require more expressiveness. In particular, we consider the wish of users to have the highest or lowest possible value for an attribute. To this end, we added the *HyMaximumExpression* and the *HyMinimumExpression* to be able to express that an attribute should have its highest or lowest possible value.

A user profile (*HyProfile*) consists of an ordered list of preferences (*HyPreference*). The order of the preferences in the list represents their priority – higher importance first. Thus, when reconfiguring an SPL, the reconfiguration engine should successively try to fulfill the preferences, starting with the most important one. An example of the user profile for a sporty driver has already been given in Section 3.

As not all users want to define their own preferences, we extend this concept by the notion of *manufacturer profiles*. These profiles are meant to provide a sensible default for customization options and are defined by the manufacturer. As those profiles are only designed as a default, users can override them using their own profiles. Thus, when reconfiguring an SPL, manufacturer profiles are less important than user profiles. On the implementation side, manufacturer profiles can be defined in the same way as user profiles are, using the metamodel depicted in Fig. 5.

4.3. Modeling evolving SPLs

Over the course of time, engineers change SPLs as part of software evolution to address altered or new requirements, which may also change the valid configuration options specified in the FM. Modifying the FM without keeping track of the evolution itself results in loss of information as the old versions of the FM cannot be retrieved and analyses cannot consider all details of the evolution history. Keeping old versions of an FM can be necessary, e.g., to support customers with products based on old versions or to find the historical root of anomalies. Additionally, the future evolution of SPLs needs to be planned as this is an incremental approach which involves many stakeholders. This may be necessary for different stages of future planning. For instance, near-future evolution and far-future evolution of SPLs can be planned at the same time, while engineers work with the current version of the SPL and derive variants. Moreover, to reason on feature model evolution, engineers and tools need to know exactly how elements have evolved. For instance, it may be important to know whether a feature has been moved in the tree-hierarchy of the feature model or whether the original feature has been deleted and a new feature at the new position has been added. To keep track of the evolution, preserve the old versions of the FM and plan future evolution in one model, evolution needs to be treated as a first-class construct.

To capture the evolution of FMs and preserve the information of the old model, we defined *Temporal Feature Model (TFMs)* [7]. TFMs define the concept of *temporal elements*, capturing evolution as first-class entities. Temporal elements have a limited time span in which they are temporally valid. Their temporal validity ϑ is an interval defined by two points in time: the start of their temporal validity, ϑ_{since} , and the end, ϑ_{until} . The temporal validity is a right-open interval: $\vartheta = [\vartheta_{\text{since}}, \vartheta_{\text{until}})$. This means that elements are temporally valid at ϑ_{since} but not anymore at ϑ_{until} . This is necessary to provide seamless temporal validities of elements, e.g., for evolving names of a feature, as it has to always have a valid name. However, if the beginning and the end of the temporal validity of a temporal element are the same, i.e., $\vartheta_{\text{since}} = \vartheta_{\text{until}}$, the respective element is deleted from the model as it is not valid at any point in time and, thus, obsolete. Using the concept of temporal validity, the entire evolution history of elements is captured directly. Thus, it is directly visible when such an element is introduced and invalidated again without the need of investigating each model version.

To allow the arbitrary evolution of an FM, it has to be possible to represent changes on each individual element of the FM, i.e., features, groups, and feature attributes. Moreover, properties of those elements need to evolve too. In standard FMs, not all information possibly affected by evolution is captured in dedicated elements as some of this information is represented as relations between elements. For example, groups are related to a parent feature, which determines their

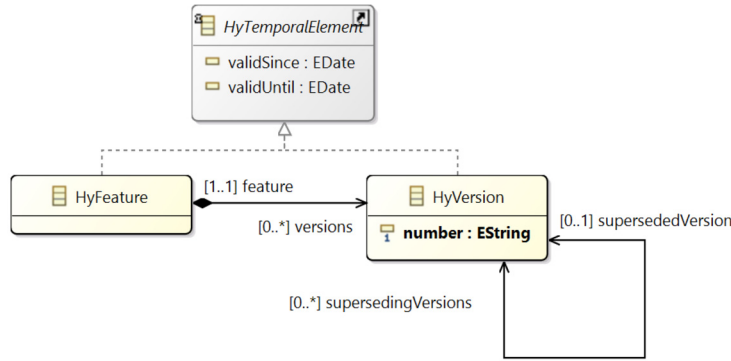


Fig. 7. Metamodel for temporal feature versions.

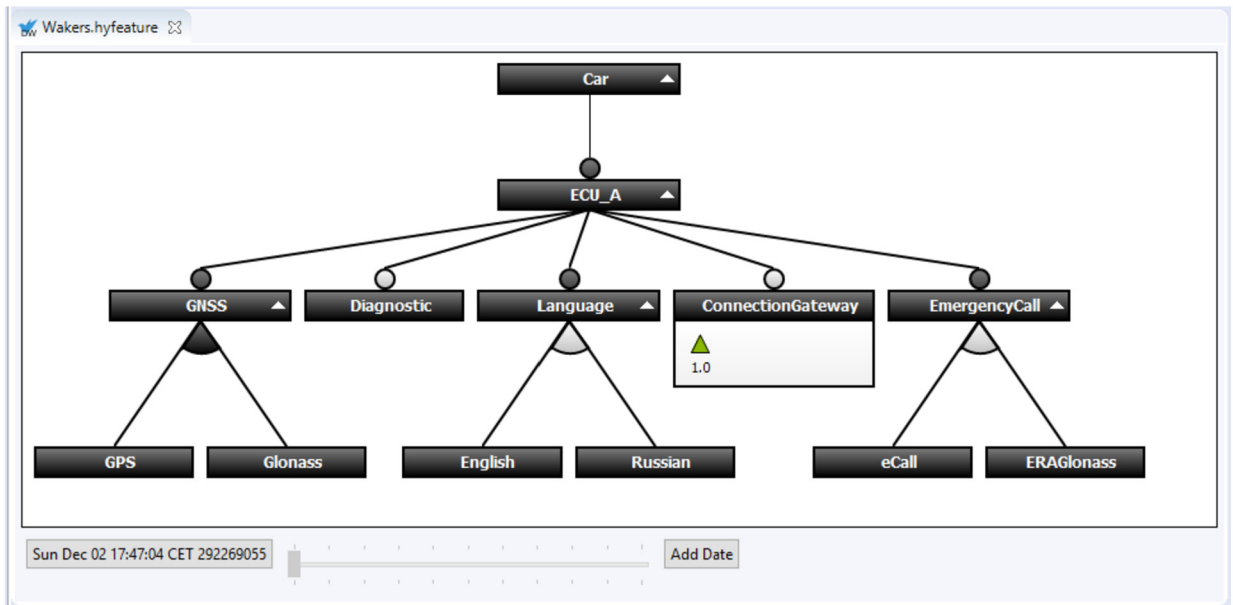


Fig. 8. Metamodel for temporal feature versions.

When SPLs evolve, not only FMs evolve, but realization artifacts do as well. Artifacts may be added, e.g., to provide the implementation for new features in the FM. Moreover, those artifacts may evolve to fix bugs, update a feature's behavior, or for providing compatibility of old features with newly added features. Those changed artifacts may result in different behavior of features. To represent this changed behavior and to capture the evolution of a feature's implementation, we adopt the concept of feature versions introduced by Seidl et al. with Hyper Feature Model [12]. In our running example in Fig. 2, the features **ConnectionGateway** and **GearAdvice** have two different versions. We extend this concept by facilitating the evolution of versions themselves, using temporal elements.

Fig. 7 shows the integration of feature versions as **HyVersions** in TFM. Each **HyVersion** may have one **HyVersion** it supersedes (i.e., the predecessor in the evolution history) and a set of **HyVersions** it is superseded by (i.e., the successors in the evolution history). Similar to all evolvable elements in TFM, **HyVersions** inherit from **HyTemporalElement**, allowing to model their evolution using temporal validities.

Modeling instances of a TFM can be very tedious due to the complex metamodel structure (cf. Fig. 6). Thus, this is not a feasible option for engineers defining such models without suitable tool support. To this end, we hide this complexity from engineers by providing an advanced editor concept [8]. In this editor, engineers model their feature model as for any other feature model editor. When they want to evolve the feature model, they can switch to a new point in time using an *evolution slider* (cf. at the bottom of Fig. 8). This evolution slider contains all points in time for which an evolution step is saved in the model. Additionally, engineers can manually add new dates to this slider. After switching to the date at which they want to perform evolution, engineers can modify the feature model as they are used to without evolution. In the background, the editor saves the performed modifications as evolution in the model. However, in the current state, it is not possible for multiple developers to simultaneously modify a TFM as we do not yet provide any synchronization

mechanisms. Using this editor, engineers are not confused by the complex metamodel but can benefit from the model's captured evolution.

As features may be added/removed or the structure of the FM evolves in TFM, CTCs need to be able to evolve as well. To this end, CTCs were modeled as temporal elements. As features or attributes referenced by a CTC may be removed, a CTC's temporal validity has to be limited to the interval of the intersection of all of its referenced elements. To be able to constrain and target specific versions of features in CTCs, our expression language was extended to take feature versions into account. A dedicated construct was used 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., $[1.1]$ specifying an exact version number or $[\geq 2.0]$).

4.4. Modeling evolving context-aware SPLs and profiles

As SPLs evolve, environmental impact on newly added/removed/changed features has to be considered. Moreover, new contextual information may be taken into account as new sources for this kind of information are available, e.g., new sensors. Thus, contextual information and their impact on the SPL in the form of VFs and contextual CTCs need to evolve.

To this end, the flexibility of temporal elements (cf. Section 4.3) was used and combined with our metamodels for context-aware SPLs (cf. Section 4.1). In particular, `HyContextualInformation`, `HyEnumLiteral` (cf. Fig. 3), and `HyValidityFormula` (cf. Fig. 4) were modeled as `HyTemporalElement`. In this way, each of these elements can evolve. As we are using the default CTCs with an extended expression language for contextual CTCs and we made CTCs evolvable in Section 4.3, contextual CTCs are already evolvable. A model consisting of a TFM, contextual information, VFs and (contextual) CTCs is called *HyFM*.

Moreover, also `HyProfiles` and `HyPreferences` (cf. Fig. 5) were modeled as temporal elements. In this way, the customization of the reconfiguration behavior can evolve as well and profiles can co-evolve with contextual information and VFs. This allows us to model co-evolution of an SPLs with contextual information, the contextual impact and the user preferences.

5. Reconfiguration engine

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

A reconfiguration process is triggered whenever a variant of an SPL needs to be adapted. The necessity for reconfiguration may vary depending on the domain. For instance, this process is started on each context change or whenever engineers evolved SPL artifacts and want all systems to adapt to the new SPL version. In our case, we assume that a reconfiguration process is started on each context change and whenever engineers evolve SPL artifact.

As depicted in Fig. 9, `HyVarRec` requires different sources of input: the `HyFM`, the current configuration C_0 of the remote device, the current values of the contextual information Ctx , the user profile P , and the manufacturer profile M . The primary function of the contextual reconfigurator is to provide valid configurations C_{new} for the context Ctx that maximize first the preferences of user profile P and then the manufacturer profile M . In case of two configurations of equal quality regarding the maximization of the user and manufacturer 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 whose 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 profiles P and M as possible.

We rely on Satisfiability Modulo Theory (SMT) [18] for the reconfiguration engine. In particular, `HyVarRec` tries to solve optimization problems where constraints defined by means of a logical formula 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 [11], we transform the “standard” feature model part of a `HyFM` into propositional formulas on features.

To potentially allow `HyVarRec` to support different FM modeling engines, we require the FM with its entities and constraints to be given as a JSON object.¹ By convention, a feature with id f is represented as `feature[f]`, an attribute with id a is represented as `attribute[a]` while a context with id c is represented as `context[c]`.

As attributes and contexts can take different values, we require users to detail their domain. In particular, to simplify the notation, we require feature attributes and context to have a finite domain represented by a contiguous set of integers. For instance, Boolean attributes or contexts have a domain of $[0, 1]$, Integer attributes or contexts have their specified domain. For attributes or contexts which can have values of an enumeration with n literals, the domain is $[0, (n - 1)]$. Attributes

¹ The JSON input schema formally describing the input for `HyVarRec` can be retrieved from https://github.com/HyVar/hyvar-rec/blob/master/spec/hyvar_input_schema.json.

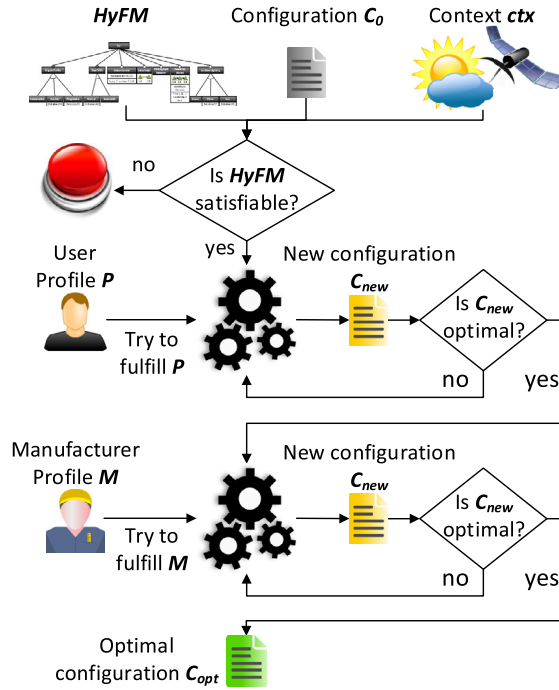


Fig. 9. Workflow of the contextual reconfigurator.

are defined in a list where each attribute is introduced by defining its id using the keyword `id`, the limits of its domain (`min` and `max` keywords) and the feature it is associated with (`featureId` keyword). For instance, the Boolean attribute `a` associated with feature `f` can be defined as follows.

```

"attributes": [
  { "id": "attribute[a]",
    "min": 0,
    "max": 1,
    "featureId": "feature[f]"
  } ]

```

Context can be defined in a similar way. For instance, the context `c` that can take a value between 3 and 10 can be defined as follows.

```

"contexts": [
  { "id": "context[c]",
    "min": 3,
    "max": 10
  } ]

```

To respect their evolution (cf. Section 4.3), both, attributes and contextual information are only translated and defined for HyVarRec if they are temporally valid at the reconfiguration time.

The HyFM with its entities and constraints is given in a textual representation as a list of propositional constraints.² The EBNF grammar defining the propositional constraint is defined in Table 1 following the ANTLR conventions.³ As can be seen, the constraint is a simple Boolean expression supporting the conjunction, disjunction, implication, negation of arithmetic expressions compared with standard operators such as '`<=`', '`=`', '`>=`', '`<`', '`>`', '`!=`'.⁴

The HyFM is defined by listing all constraints that represent its structure, VFs, and CTCs. For instance, the HyFM having only a root feature `f` is represented by only the constraint requiring the root feature to be selected. This can be represented as follows.

```

"constraints": [
  "feature[f] = 1" ]

```

² Please note that the first version of HyVarRec relied on Constraint Programming. The decision of having an intermediate representation for the constraints of the FM allowed a smooth transition between the use of SMT instead of CP as solving engine.

³ ANTLR (ANOther Tool for Language Recognition) – <http://www.antlr.org/>.

⁴ While arithmetic operation like integer division, modulo, or absolute value are not encoded directly, all these more complex operators can be easily encoded using the existing operators. For example, the integer division constraint $a/b = c$ can be encoded as $a = b * c$.

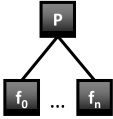
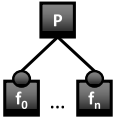
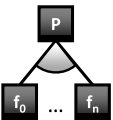
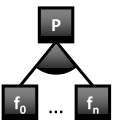
Table 1
Grammar of constraints.

```

constraint : boolean_expr;
boolean_expr : boolean_term (
  ('and' | 'or' | 'impl' | 'iff' | 'xor') boolean_term ) * ;
boolean_term : ('not')? ('true' | 'false' | relation) ;
relation : expr
  (('<=' | '=' | '>=' | '<' | '>' | '!=') expr)? ;
expr : term (('+' | '-' | '*') term) * ;
term : ('context[' | 'feature[' | 'attribute[' id ']' |
  [-]?[0-9]+ | '(' boolean_expr ')') ;
id : [a-zA-Z_][a-zA-Z0-9_]*

```

Table 2
Structural constraint for a HyFM.

HyFM	Translation
	$(\text{feature}[f_0] = 1 \text{ or } \dots \text{ or } \text{feature}[f_n] = 1)$ $\text{impl feature}[P] = 1$
	$\text{feature}[P] = 1 \text{ impl}$ $(\text{feature}[f_0] = 1 \text{ and } \dots \text{ and } \text{feature}[f_n] = 1)$
	$\text{feature}[P] = 1 \text{ impl}$ $(\text{feature}[f_0] + \dots + \text{feature}[f_n]) = 1$
	$\text{feature}[P] = 1 \text{ impl}$ $(\text{feature}[f_0] = 1 \text{ or } \dots \text{ or } \text{feature}[f_n] = 1)$

Differently to attributes and context, features do not need to be explicitly defined because their domain is automatically set to $\{0, 1\}$ where 1 means that the feature is selected. Features can, therefore, be automatically introduced simply by mentioning them in one of the constraints.

We would like to note that the expressive power of these constraints allows capturing all possible constraints of HyFMs. The hierarchical structure of the FM is translated into constraints as shown in Table 2. The translation of CTCs is straightforward since the expressive power of the constraint notation is exactly the one introduced in Section 4. In a similar way a cVF v associated with the feature f can be encoded with the constraint $\text{feature}[f] = 1 \text{ impl } v$. To respect the evolution of CTCs and VFs, we only translate formulas which are temporally valid at the point in time at which we reconfigure (cf. Section 4.3).

Please note that 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 `ConnectionGateway` of the running example in Fig. 2, there are two versions: 1.0 and 2.0. `ConnectionGateway` can be therefore encoded in $\text{feature}[1]$, its version 1.0 in $\text{feature}[2]$ and its version 2.0 in $\text{feature}[3]$ with the following requirement.

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

This ensures that exactly one version of the `ConnectionGateway` 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 HyFM can still use `ConnectionGateway`. Version-aware constraints can encompass multiple versions defined by the expressions introduced in Section 4. Considering the following constraint: $e \rightarrow \text{ConnectionGateway}[\geq 1.0]$ encompasses the versions 1.0 and 2.0. To determine all involved versions, we use the successor/predecessor relation following the approach introduced in [12]. Afterward, we translate such an expression to the sum of all encompassed versions.

Manufacturer profiles and user profiles are encoded into a list of preferences. Both preferences share the same syntax captured by the following EBNF grammar.

```
preference: constraint |
('min' | 'max') '(' 'attribute[' id ']' ')' '
```

A preference can be a Boolean expression, an arithmetic expression, or a requirement to minimize or maximize the value of a given attribute. A Boolean expression indicates that the preference is to satisfy it while for arithmetic expression the preference indicates to maximize their value.

The only difference between manufacturer preferences and the user one's is related to their priority: user preferences have a higher priority. The priority is captured by listing the preferences earlier. For instance, if GearAdvice is translated to *feature*[1], ConnectionGateway to *feature*[2] and version 2.0 of ConnectionGateway to *feature*[3], the second and the third preference of the sporty driver (cf. Section 3) would be translated as follows.

```
"preferences": [
  "feature[1] = 1", "feature[2] = 1 impl feature[3] = 1" ]
```

As for CTCs, the encoding of manufacturer and user profiles into preference expression is straightforward. The last ingredient needed for the input of HyVarRec is the current configuration and context. This must be introduced using the keyword `configuration` listing all the feature selected (`selectedFeatures` keyword), the attributes values (`attribute_values` keyword), and the contexts (`context_values` keyword). As an example, an FM having a feature *f* selected with attribute *a* set to 0 and context *c* set to 1 can be defined as follows.

```
"configuration": {
  "selectedFeatures": [ "feature[f]" ],
  "attribute_values": [
    { "id": "attribute[a]",
      "value": 0 } ],
  "context_values": [
    { "id": "context[c]",
      "value": 1 } ]
}
```

Given all this information, HyVarRec can finally be run. It first translates the list of constraints into SMT expressions encoding the features, attributes, as integer variables and contexts as Integer constants. With this information, HyVarRec is able to determine if a valid configuration exists. If so, HyVarRec proceeds in finding those configurations that satisfy more preferences or are more similar to the initial configuration. To do this HyVarRec translates the preferences into SMT expressions and it uses a recent addition of the Z3 [19] solver to trigger the optimization of the expression. Note that preferences are optimized according to their input order. This means that a preference with higher priority is tried to be satisfied first. The process terminates when no other valid configuration satisfying more preferences is found, meaning that the last determined configuration is the one maximizing the preferences and similarities with the initial configuration.

The output of HyVarRec is a JSON object representing the best possible 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. HyVarRec is written in Python, open source, and freely available from <https://github.com/HyVar/hyvar-rec>. To allow flexible deployment, HyVarRec can be easily installed using the Docker container technology [20]. Moreover, thanks to service-oriented technology [21], HyVarRec can be accessed as a service by using simple HTTP POST requests.

Before concluding the section we would like to remark that the version of HyVarRec presented in this paper differs from the first one presented in [5,6]. The current version of HyVarRec is indeed a complete reimplementaion done in order to use an SMT solver instead of a Constraint Programming solver. The original approach relied on *MiniSearch* [22], a utility that allows programming the search coordinating the activity of a solver supporting the *MiniZinc* language [23], i.e., the most supported language to define CSPs. The reason for this switch is manifold. First of all, *MiniSearch* is not well documented, nor actively supported anymore. On the contrary Z3 [19] is probably the best know SMT solver, well documented with a huge community that uses it and maintains it. Second, none of the most recent and promising constraint solvers support the incremental API needed by *MiniSearch* to add additional constraints avoiding the restart of the solver.⁵ Due to the introduction of preferences that require the addition of constraint at runtime, the use of *MiniSearch* implied the need to restart many times the solver, potentially redoing part of the search already performed before. Conversely, SMT solvers support natively the addition and removal of formulas without the need to restart the solver.

6. Case study

In this section, we show the feasibility of our approach by modeling the running example with our tool suite DARWIN-SPL⁶ [8] and applying HyVarRec on a reconfiguration scenario.

⁵ To the best of our knowledge, only the solver Gecode can be used for this purpose. Unfortunately, Gecode is a classical constraint solver not supporting learning that may be beneficial for solving the reconfiguration problem.

⁶ <https://github.com/DarwinSPL/DarwinSPL>.

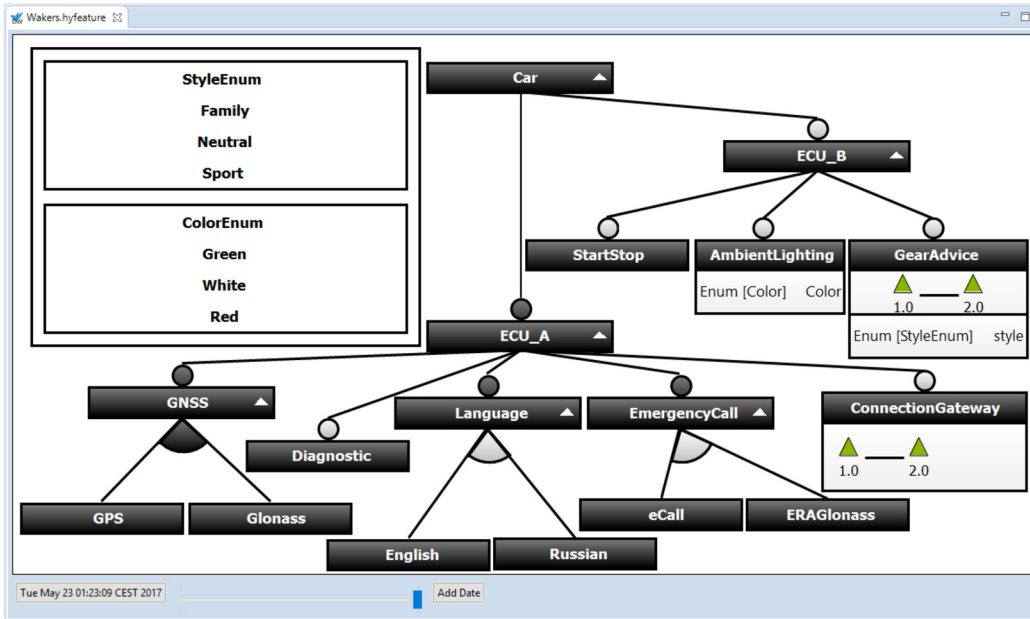


Fig. 10. Screenshot of the feature-model editor after evolution.

Before evolution, the HyFM is the one presented in Fig. 2 but without ECU_B and its subfeatures and all CTCs and VFs using them. Moreover, ConnectionGateway has only version 1.0. In this scenario, the car is located in Europe and, therefore, for the initial configuration of the car it has the features Car, ECU_A, EmergencyCall eCall, Language, English, GNSS, and GPS. In the first step, we also do not consider user or manufacturer preferences, as they become relevant when ECU_B is added.

When the driver is now driving to Russia, this context change will be reported to HyVarRec and a new configuration is computed. As a result, eCall is replaced with ERAGlonass, English is replaced with Russian and in addition to GPS, Glonass is selected. After arriving in Russia, the fuel level is down to 12%.

Now, the manufacturer evolves the SPL using DARWINSPL. Fig. 10 shows a screenshot of the DARWINSPL FM editor modeling the new feature model. New features attributes and versions are added. Moreover, more contextual information can be captured (e.g., GearViolation, FuelPercentage, and PollutionLevel). Fig. 11 shows the editor for modeling contextual information. To handle the new features and reconfiguration rules, also new CTCs and VFs are added as exemplified in Fig. 12 and Fig. 13. Finally, as the manufacturer wants to provide sensible reconfiguration rules defaults, manufacturer profile can be also defined as shown in Fig. 14.

Since after the evolution step some artifacts have evolved possibly requiring some change of configuration, HyVarRec is triggered to deal with those changes. As the fuel level is below 15%, the StartStop feature is selected. As consequence, ECU_B and ConnectionGateway have to be selected. As specified in the manufacturer profile, version 2.0 of the ConnectionGateway is selected. Moreover, the GearAdvice in version 2.0 is also selected, fulfilling the manufacturer profile. HyVarRec reacts on the evolution, computes this scenario correctly thus allowing the new configurations to be deployed.

After being notified by the manufacturer about the updated SPL, the sporty driver sets his preferences. As the user profile is higher prioritized than the manufacturer profile, it overrides some of the reconfigured features. Following his preferences, as shown in Fig. 15, the GearAdvice feature is deselected. Moreover, the AmbientLighting is activated and set to Red. As the ConnectionGateway is still selected, the last preference sets the version to 1.0. HyVarRec correctly computes this configuration, incorporating the user profile.

Imagine now that the driver wants to leave Russia going back to Europe. The driver takes the highway and on the way, she is often distracted and uses the shift often wrong. This triggered the GearAdvice to be enabled as well as the newest version 2.0 of the ConnectionGateway. Then, right before the border, she wants to buy some Russian goods and enters a city where the PollutionLevel increases to 75. This contextual change again enforces the car to reconfigure to Green Ambient Lighting and Family GearAdvice.

Using DARWINSPL, 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 [24].

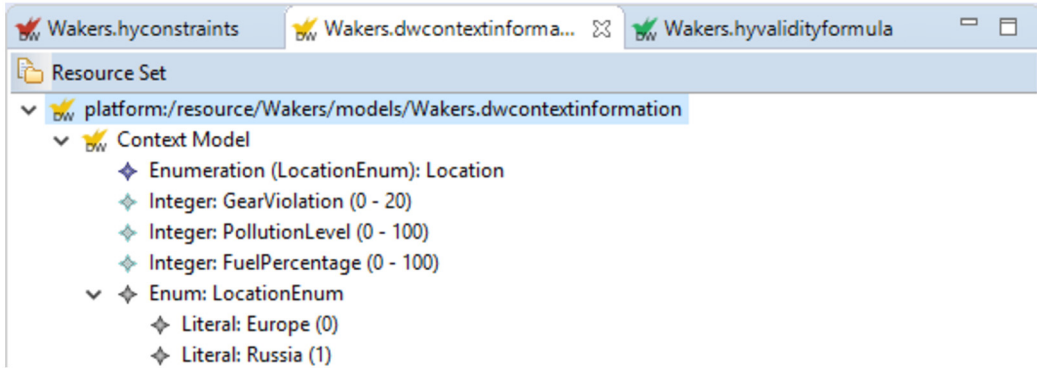


Fig. 11. Screenshot of the contextual information editor.

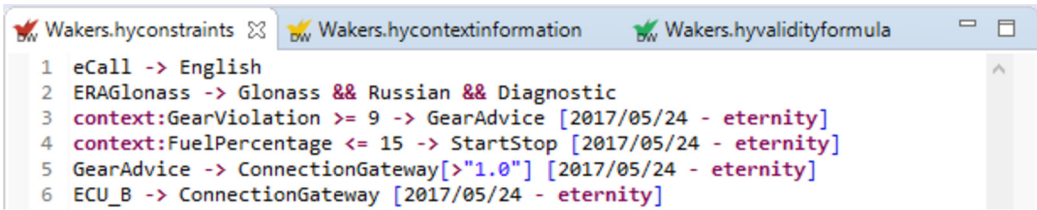


Fig. 12. Screenshot of the cross-tree constraint editor.

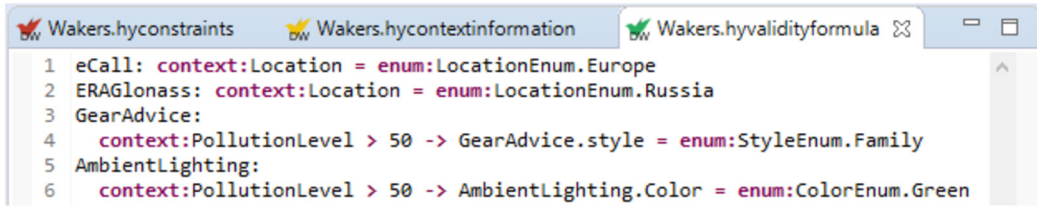


Fig. 13. Screenshot of the validity formula editor.

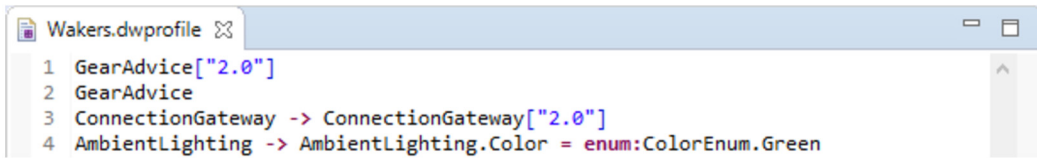


Fig. 14. Screenshot of the manufacturer profile editor.

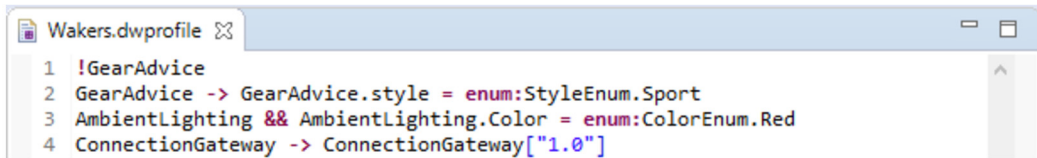


Fig. 15. Screenshot of the sporty driver's profile editor.

7. Validation

In this section, we present the tests to validate the performances of HyVarRec. To the best of our knowledge, due to the novelty of these approaches, there are neither established benchmarks nor big industrial instances of context-aware SPLs. For this reason, following common practice [13], to have at least a preliminary validation of HyVarRec, we first benchmarked it against random generated context-aware FMs. To allow further comparisons with reconfiguration tools that support only

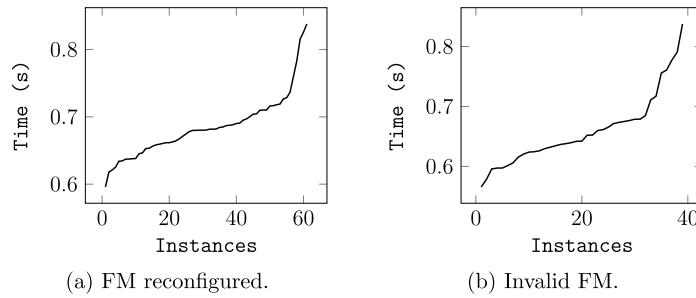


Fig. 16. Bench100 results.

plain FM without context information, we then run HyVarRec over satisfiable instances of the SPLOT benchmark [25].⁷ We would like to remark that the findings of [26,27] investigating the analysis of non-context-aware FM, show that these tasks are easy for SAT solvers. Since from the theoretical point of view, VFs can be seen as constraint implications not very dissimilar to CTCs, and the fact that SMT solvers generalize SAT solvers, we, therefore, expected to solve the majority of context-aware FM with hundreds of features in few seconds. The results below confirm this expectation.

7.1. Randomly generated context-aware FM

The context-aware FMs were generated using the AFMwC tool,⁸ i.e., an extension of the BeTTy FM generator [28], which injects into randomly generated FMs contexts and VFs. For the first benchmark, we generated 100 FMs each with 100 features (Bench100), while for the second, we generated 100 FMs each with 500 features (Bench500). We used the generator with default parameters for fixing the number of CTCs and VFs. For this purpose, up to 10 contexts were generated randomly, each context having possibly up to 10 values. HyVarRec was run on every single instance with a time cap of 300 seconds on an Intel i7-5600U CPU 4 core machine having 8 GB RAM and Ubuntu 16.04 operating system. For every instance in the benchmark, we tried to find a configuration having the root feature selected.⁹

Note that the AFMwC FM generator does not guarantee the consistency of the FM and the fact that the generated FMs may be invalid, i.e., they do not admit the existence of a valid configuration. This allows us to validate HyVarRec also in the cases when no possible reconfiguration could be performed. In total, HyVarRec proved that 39 (resp. 58) instances were invalid for the Bench100 (resp. Bench500) benchmark.¹⁰

The running times of HyVarRec over the instances of Bench100 are presented in Fig. 16. In particular, we distinguish the results obtained considering the valid instance where HyVarRec was able to produce a configuration (Fig. 16a) from those where no valid configuration was possible (Fig. 16b). The instances were sorted based on their running times. As can be seen, for all the instances HyVarRec took less than a second: 0.686 seconds on average to produce a valid configuration if any existed, 0.66 seconds on average to prove that no valid configuration can be obtained. Looking at the distribution of the times, it seems that HyVarRec often took slightly less time to prove that no valid configuration exist. This is probably due to the fact that HyVarRec was able to rule out the possibility of a valid configuration by quickly finding out an unsatisfiable set of constraints, while the computation of a valid configuration required the exploration of a bigger part of the search space. Nevertheless, all the instance having up to 100 features were solved in less than a second.

Fig. 17 presents instead the results obtained considering the Bench500 benchmark. From the qualitative point of view, it is clear that these are very similar to the results obtained for the Bench100 benchmark. From a quantitative point of view, HyVarRec took longer to solve these instances. HyVarRec was indeed able to find a valid configuration in 2.93 seconds on average while it took 2.65 seconds on average to prove that no valid configuration could exist. Every instance was solved in less than 4 seconds.

The obtained results confirm that HyVarRec can be used to handle FM with up to hundreds of FM. Clearly, due to the complexity of the tests and the NP-completeness of the problems, developers need to be aware that checking the validity may require in rare exceptions more time.

⁷ Due to a change in the JSON input format between the current version of HyVarRec and its previous MiniSearch backed up version, currently, it is impossible to compare the different versions using the same benchmark. We would like to remark that due to the different nature of solvers, it is possible that the previous version could be faster than the current one for some instances.

⁸ <https://github.com/magnurh/AFMwC-thesisProject>.

⁹ Please note that, from the theoretical point of view, every reconfiguration task of a FM can be reduced to checking if the root feature is selected. Indeed, in case the reconfiguration requires other constraints c_1, \dots, c_n to be fulfilled, these constraints can be added as a CTC simply requiring that if the root feature is selected then also the conjunction of c_1, \dots, c_n must hold.

¹⁰ Note that the number of invalid instances of Bench500 was expected to be higher than in Bench100 since the instances in Bench500 have more CTC that can be easily violated.

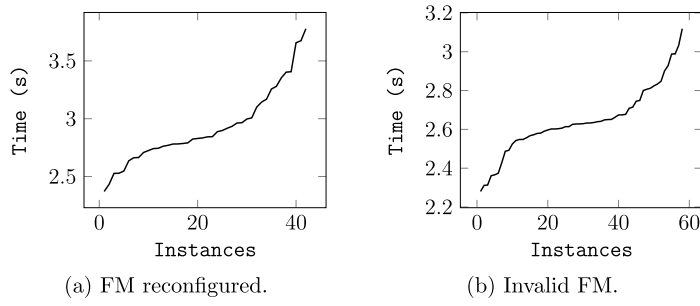


Fig. 17. Bench500 results.

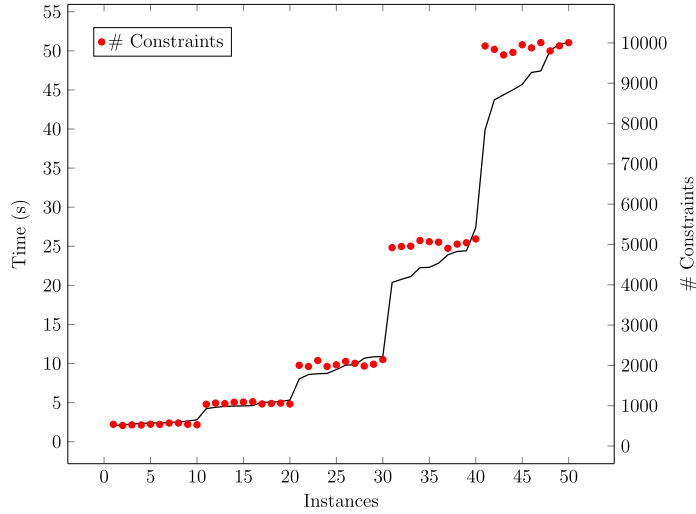


Fig. 18. SPLOT random instance results.

7.2. Non-context-aware FM

Even though HyVarRec was not designed to compete with dedicated analysis tools able to process FM without attributes and context, it can still be used to process these simple form of FMs. In this remaining part of the section, to allow further comparison with standard analyzers, we present the results obtained by running HyVarRec over instances of the SPLOT benchmark [25]: an established benchmark that contains FM generated automatically to support empirical studies on the performance and scalability of automated techniques for reasoning on feature models.¹¹

From the SPLOT feature repository, we considered both randomly generated and real FM running them as before on an Intel i7-5600U CPU 4 core machine having 8 GB RAM and Ubuntu 16.04 operating system.

For the random instances, we consider all the consistent random instance present in the SPLOT repository. These are instances where each type of mandatory, optional, inclusive-OR and exclusive-OR features were added with equal probability. The number of children per parent feature varied from 1 to 6, and the CTCs were generated as a single random 3-CNF formula. In the repository, there are 5 datasets of 10 instances each. Each dataset had a given number of features ranging from 500 to 10000.

Fig. 18 shows the times taken by HyVarRec to produce a configuration for the SPLOT consistent instances. The red dots represent the number of constraints obtained by converting the FM into the HyVarRec representation. Due to the nature of the benchmarks, these constraints are almost in a 1 to 1 proportion with the number of features of the FM.

The plot shows that the times taken by HyVarRec varies based on the number of constraints of the FM. HyVarRec indeed spends a considerable percentage of its running time to parse the constraints but, when this operation is completed, the search task is performed quickly. Roughly speaking we can say that the running times are almost 4/5 seconds for every 1000 constraints. HyVarRec indeed takes in the worst case 51 seconds to solve an instance with 10005 constraints. We deem that the possibility of using HyVarRec to propose configuration with FM of up to 10000 features in less than a minute is more than enough for the majority of the daily use cases.

¹¹ Please note that analyzers performed more than one check of the FM attempting for instance to identify not only its invalidity but possibly also other anomalies such as dead features or false optional features [13]. Hence, to be fair, we do not report a direct comparison with these analyzers.

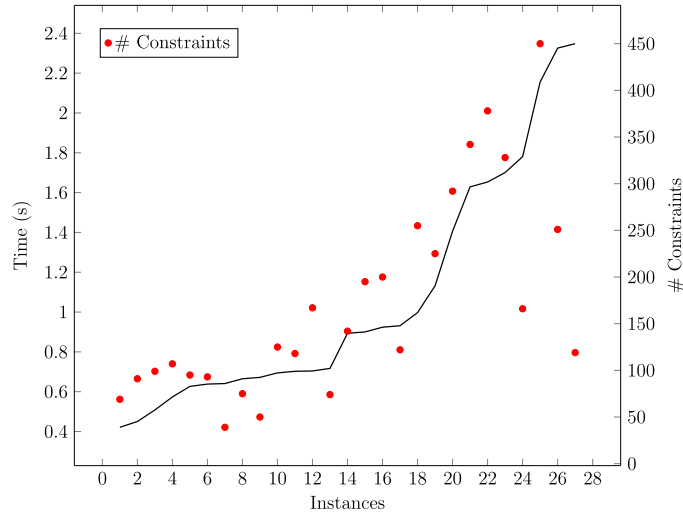


Fig. 19. SPLOT real instance results.

After this test, we considered real-world FMs of the SPLOT repository by considering all available FMs having more than 80 features. Thus, we were able to retrieve 27 FMs ranging from 80 to 451 features. Fig. 19 shows the time taken by HyVarRec to solve those instances. Even in this case, there is a correlation with the number of constraints and the time taken to configure the FM. In the worst case, HyVarRec took less than 2.5 seconds to configure even the biggest FM. Without surprises, we corroborate the findings of [27] that the analysis of real FM is an easy task.

8. Related work

Before giving an overview of related work, we start by comparing this work with our previous work published in various international workshops and symposiums [5–8]. In particular, in this work we have made significant extensions in several directions: VFs and a first prototype of HyVarRec were initially introduced in [5]. Here we extended this work by allowing the possibility to define contextual CTCs, thus allowing users to create dependencies between context and features more freely. Moreover, we have completely rewritten HyVarRec to rely on SMT [18] as solving engine instead of CSPs in order to better support the statement of preferences. User preferences were first introduced in [6] that however still relies on the use of the old version of HyVarRec that involved the restart of the back-end solver for every preference that the user defined. This limitation is now surpassed: the current version of HyVarRec does not require the restart of the SMT solver. Furthermore, differently from [6], this paper also formalizes and integrates the manufacturer preferences into the reconfiguration process. [7,8] introduced our approach to model evolution in non-context-aware SPLs and the modeling framework DARWINSPL. In this paper, we address the evolution of context and the integration of HyVarRec into DARWINSPL to allow the reconfiguration directly from the GUI. Differently from [5–8], we also provide a detailed presentation of all the FM model artifacts and the input of HyVarRec, and a validation of HyVarRec on an extensive benchmark of random and industrial instances.

In the literature, a wide range of approaches for the development of context-aware software exists. Among the works using SPL, one of the closest to our approach is [29]. Features are constrained to contextual information via the Context Variability Model. The combination of the Context Variability Model and the FM results in what the authors called a Multiple Product Line FM. The contextual information is not captured as in our case within the original FM but imposed as additional cross-tree constraints. Similarly, in [30,31], both the context and the variability model are captured by using two distinct feature models that are connected using rules that establish how to configure a system based on contextual information. These FMs could be composed using the approach presented in [32]. In [33], FMs are enriched with contextual information but, differently from our case, their connection with the features is given using some external rules or constraints. In [34], the contextual information used to model adaptation of applications is described by an ontology representing a global context model. Moreover, local context models tailored to the specific needs of a particular application are defined by the authors as a view over the global context in the form of a feature model. Rules are then used to generate the feature model from the global context. Our approach deviates from these ones as we explicitly connect the contextual information with the features. This approach and ours have a different focus but can be combined. We argue that starting from a complex contextual model, such as the ones in [29,30,33], it is always possible to encode the relevant information of its concrete instances into our representation. Therefore, our approach is not incompatible with complex contextual models that can be used, provided that, in a pre-compilation phase, their concrete instances are encoded into a map of identifiers-values. Bashari et al. [35] propose a reference framework for modeling DSPLs. To model contextual information, they suggest to use the *Web Ontology Language* (OWL) [36] and relate these to feature models. Also, Gámez et al. [37] use OWL to model

context and relate them to elements of the Common Variability Language. During the reconfiguration process, they use models at runtime to check how to reconfigure their systems. In terms of modeling context, OWL is more expressive than our proposed metamodel but lacks the possibility to express preferences and to incorporate evolution.

Pfannenmüller et al. [38] focus on the reconfiguration process for DSPLs or self-adaptive systems. They use feature models supporting attributes to model system variability and contexts and use a SAT solver to perform the reconfiguration process. Compared to HyVarRec, instead of supporting user defined preferences, they associate costs or priorities to features to decide in case of conflicts or ties which configuration to choose for the adaptation step. HyVarRec preferences are instead more expressing allowing the user to state in a concise way priorities and costs involving not only a single feature but subset of features, attributes, and context.

In [39], a model-driven engineering approach for transforming a generic feature model according to a context model was proposed. This is a completely different approach from ours: we do not handle and manipulate FM but we introduce into existing FM relations with the contextual information. Thus, with our methodology, the original FM will remain unmodified.

Other related works are [40–42], 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 aforementioned 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 [43] 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 this information.

Some approaches like [44,45] assign values or costs to features that allow during the (re-)configuration to optimize certain properties (e.g., costs and productivity) to create the 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.

Preferences are well studied from a qualitative and quantitative way in the field of decision theory [46], Constraint Programming [15]. Preferences are often incorporated in COPs [47,48] and are also denoted as soft constraints [15]. HyVarRec is built on top of these approaches and it exploits all the experience accumulated in the Constraint Programming and SMT community to speed up the search for the configuration maximizing user preferences.

Preferences are often assumed to be given by users. However, there are approaches where preferences are derived by inspecting the history or use of an application or software. For instance, in [49], preferences are learned by analyzing the history of the different users, creating a profile of preferences for every user. Similarly, in [50], 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 [51], which deals with preferences for music recommendation systems. While these approaches are interesting and may prove useful for our future work, in this work, we assume that users elicit their preferences explicitly.

In the field of SPL evolution, different work exists as well. For instance, Seidl et al. provide the notion of feature versions with the Hyper Feature Model (HFM) [12]. We integrated this concept of feature versions in our HyFMs. Botterweck et al. introduce with *EvoFM* a concept to model the evolution of the FM [4]. With *EvoFM*, each feature has timestamps at which it is available as a configuration option. Compared to HyFMs, with *EvoFM* it is also possible to have a feature configurable at point t_0 in time, not configurable at point t_1 and again configurable at point t_2 ($t_0 < t_1 < t_2$). To model the evolution of the FM with *EvoFM*, a set of pre-defined evolution operations has to be applied. As we are modeling evolution as an own entity in the metamodel, we are independent of concrete evolution operations but can define arbitrary evolution operations on top of it. With *EvoFM*, the user is bound to the pre-defined operations. The flexible concept of *temporal elements* allows us to model the evolution of CTCs, contextual information or VFs. This would not be possible using *EvoFM*. To the best of our knowledge, no other approach exists for modeling evolving context-aware SPLs.

Capilla et al. [52] combine feature model evolution with self-adaptive systems. They propose an approach where features can be added dynamically to a feature model which is used at runtime. To this end, they utilize the notion of feature super-types in order to decide where to add the new feature in a feature tree. However, they do not consider the reconfiguration step based on the evolved feature models, which instead we are able to perform thanks to the combination of TFMs and HyVarRec.

9. Conclusion

In this article, we introduced a methodology to model an SPL that can be reconfigured based on its environment. To this end, we introduced the concept of contextual information which represent the observable context of an SPL. To define the reconfiguration behavior, we proposed contextual cross-tree constraint (CTCs) and Validity Formula (VFs), specifying how the SPL should reconfigure based on the context. We also allow users and manufacturers to influence the reconfiguration

process by customizing it. Users can customize their SPL based on their desires and, additionally, can make them dependent on the current context. The preferences are given in lists, ordered by priority.

Additionally, SPLs now allow modeling evolution of the reconfiguration behavior and user/manufacture preferences. To this end, we integrated Temporal Feature Model (TFMs), a methodology to model the evolution of the FM as an own entity using the concept of *temporal elements*. We applied this concept also to contextual information, VFs and profiles. This all together allows us to model evolving context-aware SPLs in an integrated way. We also provide DARWINSPL, an open-source tool support for the modeling of evolving context-aware SPLs.

Finally, we developed the context-aware reconfiguration engine HyVarRec that considers the current configuration, the user and manufacturer profiles, and the context to produce a valid configuration maximizing user preferences. HyVarRec can be invoked when context changes or after the SPL evolved. We showed the feasibility of our methodology by modeling a realistic SPL of a customizable car, encompassing contextual information. HyVarRec successfully created new valid configurations and maximized the given user profiles, handling SPLs with up to thousand features in less than a minute.

For future work, we are interested in creating user profiles by learning from user behavior and understanding their policies [53]. Finally, we are interested in evaluating our approach with our industry partner using the toolchain presented in [24], analyzing the scalability of our approach considering a real-world scenario with simulated cars to reconfigure, and extending HyVarRec for more complex analysis along the lines of [54].

Acknowledgements

This work was supported by the European project HyVar (grant agreement H2020-644298), by the Federal Ministry of Education and Research of Germany within the project CrESt (funding number 01IS16043S) and by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

- [1] K. Pohl, G. Böckle, F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag New York, Inc., 2005.
- [2] W.G. Darryll Harrison, CES 2016: Volkswagen brings gesture control to mass production with the E-Golf Touch, Online, <http://media.vw.com/release/1123/>, 2016.
- [3] S. Robarts, Volkswagen's Golf R touch concept shows off the car cockpit of the future, Online, <http://www.gizmag.com/volkswagen-golf-r-touch/35472/>, 2015.
- [4] G. Botterweck, A. Pleuss, D. Dhungana, A. Polzer, S. Kowalewski, Evofm: feature-driven planning of product-line evolution, in: *Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering*, PLEASE '10, ACM, New York, NY, USA, 2010, pp. 24–31, <http://doi.acm.org/10.1145/1808937.1808941>.
- [5] J. Mauro, M. Nieke, C. Seidl, I.C. Yu, Context aware reconfiguration in software product lines, in: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems – VaMoS '16*, <https://doi.org/10.1145/2866614.2866620> dl.acm.org/citation.cfm?id=2866614.2866620.
- [6] M. Nieke, J. Mauro, C. Seidl, I.C. Yu, User profiles for context-aware reconfiguration in software product lines, in: *ISOIA*, in: *Lect. Notes Comput. Sci.*, vol. 9953, 2016, pp. 563–578.
- [7] M. Nieke, C. Seidl, S. Schuster, Guaranteeing configuration validity in evolving software product lines, in: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '16, ACM, New York, NY, USA, 2016, pp. 73–80, <http://doi.acm.org/10.1145/2866614.2866625>.
- [8] M. Nieke, G. Engel, C. Seidl, DarwinSPL: an integrated tool suite for modeling evolving context-aware software product lines, in: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, VAMOS '17, ACM, New York, NY, USA, 2017, pp. 92–99, <http://doi.acm.org/10.1145/3023956.3023962>.
- [9] K. Kang, Feature-oriented Domain Analysis (FODA): Feasibility Study, Technical Report CMU/SEI-90-TR-21 – ESD-90-TR-222, Software Engineering Inst., Carnegie Mellon Univ., 1990, <https://books.google.de/books?id=yYi5PgAACAAJ>.
- [10] D. Batory, Feature models, grammars, and propositional formulas, in: *SPLC*, vol. 3714, Springer, 2005, pp. 7–20.
- [11] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, in: *Advanced Information Systems Engineering*, in: *Lect. Notes Comput. Sci.*, vol. 3520, Springer, Berlin, Heidelberg, 2005.
- [12] C. Seidl, I. Schaefer, U. Altmann, 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 '14*, <https://doi.org/10.1145/2556624.2556625>, <http://dl.acm.org/citation.cfm?doid=2556624.2556625>.
- [13] D. Benavides, S. Segura, A.R. Cortés, Automated analysis of feature models 20 years later: a literature review, *Inf. Syst.* 35 (6) (2010) 615–636.
- [14] F. Maric, Formalization and implementation of modern SAT solvers, *J. Autom. Reason.* 43 (1) (2009) 81–119, <https://doi.org/10.1007/s10817-009-9127-8>.
- [15] F. Rossi, P. v. Beek, T. Walsh, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, Elsevier Science Inc., New York, NY, USA, 2006.
- [16] S.A. Cook, The complexity of theorem-proving procedures, in: *ACM Symposium on Theory of Computing*, ACM, 1971, pp. 151–158.
- [17] M. Nieke, J. Mauro, C. Seidl, I.C. Yu, User Profiles for Context-Aware Reconfiguration in Software Product Lines, Springer International Publishing, Cham, 2016, pp. 563–578.
- [18] L. De Moura, N. Bjørner, Satisfiability modulo theories: introduction and applications, *Commun. ACM* 54 (9) (2011) 69–77.
- [19] L.M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: *TACAS*, in: *Lect. Notes Comput. Sci.*, vol. 4963, Springer, 2008, pp. 337–340.
- [20] Docker Inc, Docker, <https://www.docker.com/>, last retrieved Jan 2016.
- [21] Jolie, Programming Language, <http://www.jolie-lang.org/>, last retrieved Jan 2016.
- [22] A. Rendl, T. Guns, P.J. Stuckey, G. Tack, MiniSearch: a solver-independent meta-search language for MiniZinc, in: *CP*, in: *Lect. Notes Comput. Sci.*, vol. 9255, Springer, 2015.
- [23] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, G. Tack, MiniZinc: towards a standard CP modelling language, in: *CP*, in: *Lect. Notes Comput. Sci.*, vol. 4741, Springer, 2007.

- [24] C. Chesta, F. Damiani, L. Dobriakova, M. Guerineri, S. Martini, M. Nieke, V. Rodrigues, S. Schuster, A toolchain for delta-oriented modeling of software product lines, in: 7th International Symposium on Leveraging Applications, ISoLA 2016, Imperial, Corfu, Greece, 2016.
- [25] M. Mendonça, M. Branco, D.D. Cowan, S.P.L.O.T.: software product lines online tools, in: OOPSLA, ACM, 2009, pp. 761–762.
- [26] M. Mendonça, A. Wasowski, K. Czarnecki, Sat-based analysis of feature models is easy, in: SPLC, in: ACM International Conference Proceeding Series, vol. 446, 2009, pp. 231–240.
- [27] J.H.J. Liang, V. Ganesh, K. Czarnecki, V. Raman, Sat-based analysis of large real-world feature models is easy, in: SPLC, ACM, 2015, pp. 91–100.
- [28] S. Segura, J.A. Galindo, D. Benavides, J.A. Parejo, A.R. Cortés, BeTTY: benchmarking and testing on the automated analysis of feature models, in: International Workshop on Variability Modelling of Software-Intensive Systems, ACM, 2012, pp. 63–71.
- [29] H. Hartmann, T. Trew, Using feature diagrams with context variability to model multiple product lines for software supply chains, in: SPLC, IEEE Computer Society, 2008.
- [30] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, J.-P. Rigault, Modeling context and dynamic adaptations with feature models, in: 4th International Workshop Models@run.time, 2009, p. 10, <https://hal.archives-ouvertes.fr/hal-00419990>.
- [31] N. Ubayashi, S. Nakajima, Context-aware feature-oriented modeling with an aspect extension of VDM, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC '07, ACM, New York, NY, USA, 2007.
- [32] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, R. France, Composing your compositions of variability models, in: A. Moreira, B. Schätz, J. Gray, A. Vallecillo, P. Clarke (Eds.), Model-Driven Engineering Languages and Systems, in: Lect. Notes Comput. Sci., vol. 8107, Springer, Berlin, Heidelberg, 2013, pp. 352–369.
- [33] P. Fernandes, C. Werner, E. Teixeira, An approach for feature modeling of context-aware software product line, J. UCS 17 (5) (2011) 807–829.
- [34] S. Neskovic, R. Matic, Context modeling based on feature models expressed as views on ontologies via mappings, Comput. Sci. Inf. Syst. 12 (3) (2015) 961–977.
- [35] M. Bashari, E. Bagheri, W. Du, Dynamic software product line engineering: a reference framework, Int. J. Softw. Eng. Knowl. Eng. 27 (02) (2017) 191–234, <https://doi.org/10.1142/S0218194017500085>, arXiv:<http://www.worldscientific.com/doi/pdf/10.1142/S0218194017500085>, <http://www.worldscientific.com/doi/abs/10.1142/S0218194017500085>.
- [36] S. Bechhofer, Owl: web ontology language, in: Encyclopedia of Database Systems, Springer, 2009, pp. 2008–2009.
- [37] N. Gámez, L. Fuentes, J.M. Troya, Creating self-adapting mobile systems with dynamic software product lines, IEEE Softw. 32 (2) (2015) 105–112, <https://doi.org/10.1109/MS.2014.24>.
- [38] M. Pfannemueller, C. Krupitzer, M. Weckesser, C. Becker, A dynamic software product line approach for adaptation planning in autonomic computing systems, in: 2017 IEEE International Conference on Autonomic Computing (ICAC), 2017, pp. 247–254.
- [39] T. Possompès, C. Dony, M. Huchard, C. Tibermacine, Model-driven generation of context-specific feature models, in: SEKE, Knowledge Systems Institute Graduate School, 2013, pp. 250–255.
- [40] P.A. da S. Costa, F.G. Marinho, R.M. de C. Andrade, T. Oliveira, Fixture – a tool for automatic inconsistencies detection in context-aware SPL, in: ICEIS, 2015.
- [41] F.G. Marinho, R.M.C. Andrade, C. Werner, A verification mechanism of feature models for mobile and context-aware software product lines, software components, architectures and reuse (SBCARS), <https://doi.org/10.1109/SBCARS.2011.9>.
- [42] F.G. Marinho, R.M. Andrade, C. Werner, W. Viana, M.E. Maia, L.S. Rocha, E. Teixeira, J.B.F. Filho, V.L. Dantas, F. Lima, S. Aguiar, Mobiline: a nested software product line for the domain of mobile and context-aware applications, Sci. Comput. Program. 78 (12) (2013) 2381–2398, <https://doi.org/10.1016/j.scico.2012.04.009>, special section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011), <http://www.sciencedirect.com/science/article/pii/S0167642312000871>.
- [43] C. Parra, X. Blanc, L. Duchien, Context awareness for dynamic service-oriented product lines, in: Proceedings of the 13th International Software Product Line Conference, SPLC '09, Carnegie Mellon University, Pittsburgh, USA, 2009, <http://dl.acm.org/citation.cfm?id=1753235.1753254>.
- [44] A. Murashkin, M. Antkiewicz, D. Rayside, K. Czarnecki, Visualization and exploration of optimal variants in product line engineering, in: Proceedings of the 17th International Software Product Line Conference, SPLC '13, ACM, New York, NY, USA, 2013.
- [45] L. Ochoa, O. González-Rojas, T. Thüm, Using decision rules for solving conflicts in extended feature models, in: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, ACM, New York, NY, USA, 2015.
- [46] J. Doyle, R.H. Thomason, Background to qualitative decision theory, AI Mag. 20 (2) (1999).
- [47] C. Boutilier, R.I. Brafman, C. Domshlak, H.H. Hoos, D. Poole, Preference-based constrained optimization with cp-nets, Comput. Intell. 20, <https://doi.org/10.1111/j.0824-7935.2004.00234.x>.
- [48] C. Domshlak, F. Rossi, K.B. Venable, T. Walsh, Reasoning about soft constraints and conditional preferences: complexity results and approximation techniques, arXiv.
- [49] S. Young, J.-h. Hong, T.-s. Kim, A formal model for user preference, in: Proceedings 2002 IEEE International Conference on Data Mining, 2002, <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1183908>.
- [50] E. Agichtein, E. Brill, S. Dumais, R. Ragno, 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 '06, ACM, New York, NY, USA, 2006, <http://doi.acm.org/10.1145/1148170.1148175>.
- [51] K. Yoshii, M. Goto, K. Komatani, T. Ogata, H.G. Okuno, Hybrid collaborative and content-based music recommendation using probabilistic model with latent user preferences, in: ISMIR, vol. 6, 2006.
- [52] R. Capilla, A. Valdezate, F.J. Díaz, A runtime variability mechanism based on supertypes, in: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), 2016, pp. 6–11.
- [53] S. Reiff-Marganiec, A structured approach to VO reconfigurations through policies, in: Proceedings Third Workshop on Formal Aspects of Virtual Organisations, FAVO 2011, Sao Paulo, Brazil, 18th October 2011, in: EPTCS, vol. 83, 2011, pp. 22–31.
- [54] J. Mauro, M. Nieke, C. Seidl, I.C. Yu, Anomaly detection and explanation in context-aware software product lines, in: SPLC, ACM, 2017, pp. 18–21.