DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-Aware Software Product Lines

Michael Nieke, Gil Engel and Christoph Seidl
Technische Universität Braunschweig
38106 Braunschweig, Germany
{m.nieke, g.engel, c.seidl}@tu-braunschweig.de

ABSTRACT

Software Product Lines (SPLs) are an approach for large-scale reuse for software families by means of variabilities and commonalities. We consider three dimensions of variability representing sources of software systems to behave differently: configuration as spatial variability, dependence on surroundings as contextual variability and evolution as temporal variability. The three dimensions of variability strongly correlate: Contextual variability changes the set of possible configurations in spatial variability. Temporal variability captures changes of spatial and contextual variability over the course of time. However, currently, there is no tool support for integrated modeling of these three dimensions of variability. In this paper, we present DarwinSPL, a tool suite supporting integrated definition of spatial, contextual and temporal variability. With DarwinSPL, spatial variability is modeled as feature models with constraints. Additionally, we are able to capture the current context and its impact on functionality of the SPL. Moreover, by providing support for temporal variability, DarwinSPL supports performing arbitrary evolutionary changes to spatial and contextual variability and tracking of previous evolution and planning future evolution of SPLs. We show the feasibility of DarwinSPL by performing a case study adapted from our industrial partner in the automotive domain.

CCS Concepts

•Software and its engineering → Integrated and visual development environments; Software product lines; Software evolution;

Keywords

Temporal Feature Model, Software Product Line, Evolution, Context-Aware, DarwinSPL, Tool Suite

VaMoS '17, February 01 - 03, 2017, Eindhoven, Netherlands

ISBN 978-1-4503-4811-9/17/02…$15.00

DOI: http://dx.doi.org/10.1145/3023956.3023962

1. INTRODUCTION

Software Product Lines (SPLs) are an approach to capture common and different behavior of software systems in terms of variability [17]. We consider three dimensions of variability representing sources for software systems to behave differently: *spatial variability* implies configurable functionality of a software system in terms of different features defining the set of all possible configurations. As software systems may be in use in different environments, their functionality may be influenced by the current context [12]. Thus, the second dimension of variability is *contextual variability*, capturing influences of the environment on functionality of software systems.In the course of software evolution, requirements for SPLs change, are dropped or new requirements arise [14]. These changed requirements result in changed, removed or added features and different impact of context of SPLs. Thus, the third dimension is *temporal variability*, capturing evolution of software systems. As old versions of software systems need to be supported with updates, it is important to keep track of the evolution of SPLs. Spatial, contextual and temporal variability strongly correlate. Contextual variability changes the set of selectable configuration options of the spatial variability. Temporal variability captures evolution of spatial and contextual variability. Theoretical concepts already exist to capture the three dimensions of variability [15, 16]. However, currently, there is no tool support available to model evolving context-aware SPLs in an integrated way. We overcome this by introducing DarwinSPL, a tool suite for modeling SPLs with spatial, contextual and temporal variability. To better explain the three dimensions of variability and to motivate the necessity for their integrated management, we use an excerpt of our case study as running example.

1.1 Running Example

The case study is an existing real world scenario of our industry partner who is a software supplier for several car manufacturers. In modern cars, many electronic control units (ECUs) are responsible for controlling different functionality of cars. As our industry partner provides ECUs for multiple car manufacturers, many variants of the ECUs need to be developed. To cope with this complexity, the supplier models the ECUs as SPL. First, the supplier has to develop several variants of an ECU which capture the geolaction of cars and set up emergency calls in case of an accident. Different variants of the software for the ECU use GPS or Glonass positioning systems. Moreover, the ECU can either be used with the European emergency call system eCall or

with the Russian variation EraGlonass. The different configuration options correspond to spatial variability. Common SPL technology is able to capture spatial variability, e.g., with feature models [13], as Figure 1a depicts.

Multiple drivers of cars using the supplier's ECU often cross the border from Russia to Europe. Thus, they demand systems which use the correct emergency call system based on the car's location. Thus, the supplier needs mechanisms to automatically deploy the correct variant based on the car's location. Accordingly, in Europe, `eCall` should be selected and in Russia `EraGlonass`. We call adapting configurations based on the context contextual variability. As the impact of contexts on `eCall` and `EraGlonass` differs, spatial and contextual variability correlate. With standard SPL tools, it is not possible to model contextual information and its impact on configuration options. Moreover, there is no mechanism that reconfigures variants on an end device based on its current context.

After some time, several car manufacturers demand the supplier to provide functionality for infotainment systems. Thus, the supplier needs to model a second ECU and to evolve the SPL. The new requirements are captured by new features: a `navigation system (Nav)`, a user interface for the emergency call system named `EmergencyCallUI` and an assistance feature which suggests the best gear named `GearAdvice`. The `GearAdvice` has different suggestion types – one for economic and one for sporty driving. This evolution corresponds to temporal variability. As new configuration options are added to the SPL due to temporal variability, spatial and temporal variability correlate. With common SPL development tools, this evolution could only be captured by modifying the old feature model, as Figure 1b depicts. However, the supplier still needs to generate variants for cars having the old version of the ECU, which was not possible if the old feature model had been modified.

As environmental pollution is a problem for car manufacturers, the economic gear advices of the `GearAdvice` should be enforced if drivers shift too late too often. Thus, evolution of contextual variability is the consequence, which is the correlation of contextual and temporal variability. In previous work, we have developed concepts for all three dimensions of variability. Developing evolving context-aware SPLs requires a tool suite for integrated modeling of the three dimensions of variability, which currently does not exist.

1.2 Requirements Analysis and Contribution

To provide a tool suite integrating all three dimensions of variability, we analyzed potential use cases in cooperation with our industry partner. We extracted a set of requirements that need to be fulfilled for suitable tool support. Based on these requirements, we implemented a tool suite called DarwinSPL. In the following, we describe the requirements and briefly state our corresponding contributions in terms of how we implement them in DarwinSPL.

As pointed out in Section 1.1, different configurations in terms of varying functionality or features are needed. To this end, we define **R1**:

R1: Support for modeling spatial variability

Many SPL tools are already capable to do this, so we do not introduce uncommon functionality with this requirement but have to support the same level of functionality as the existing tools. In DarwinSPL, we realize this by providing editors for attributed feature models and cross-tree constraints (CTCs) for feature models [8, 9].

Adaptation to environmental influences requires to capture contextual variability (**R2**). As this is uncommon functionality for an SPL tool, we define more detailed sub requirements. Contexts are composed of contextual information. Thus, we need to support modeling contextual information. Additionally, modeling influences of contexts on configuration options is necessary. Moreover, we have to be able to reconfigure SPLs using the currently deployed configuration and the captured context. Thus, for contextual variability, we define the following requirements:

R2 Support for modeling contextual variability
- **R2.1:** Support for defining relevant contextual information
- **R2.2:** Support for defining influence of context on configuration options
- **R2.3:** Support for automatically reconfiguring device's variants based on the context

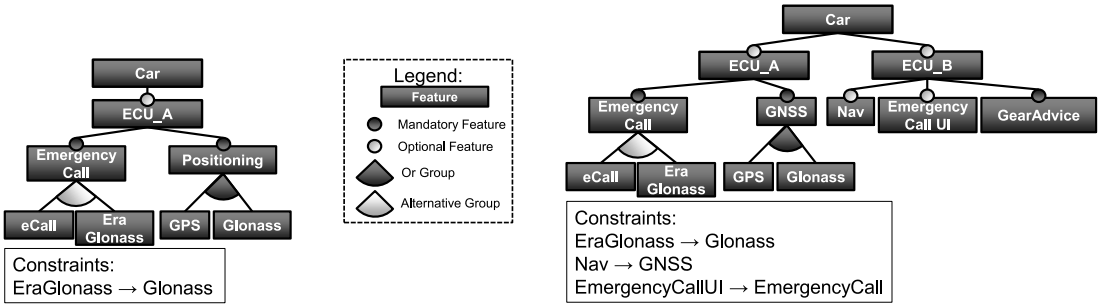
To realize this in DarwinSPL, we use metamodels for contextual information and Validity Formulas (VFs) which capture influences of contexts on features, which we have defined in our previous work [15]. To support the definition of these models, we provide suitable editors. Moreover, to reconfigure SPLs, we provide an integration with HyVarRec [1], a reconfiguration engine for context-aware SPLs [15], and a user interface allowing simulation of a reconfiguration process.

As stated in Section 1.1, SPLs are subject to evolution. Therefore, capturing temporal variability of SPLs is necessary. First of all, we need to capture evolution of spatial and contextual variability. Additionally, to support old devices with variants of old SPLs and to perform analyses on evolution, we have to keep track of the evolution. Moreover, we want to support planning of future evolution of SPLs. As many elements may change during evolution, it may be hard to understand which elements have evolved in which way. Therefore, we need to provide a clear overview of the performed evolution of the elements. Thus, for temporal variability, we have the following requirements:

R3 Support for modeling temporal variability
- **R3.1:** Perform evolutionary changes on feature models with context
- **R3.2:** Track previous evolution and plan future evolution
- **R3.3:** Support for providing overview of the evolution in a clear way

To support temporal variability, we incorporate evolution as first-class entity in the metamodels for spatial and contextual variability. We have defined a concept for evolution as first-class entity in our previous work [16]. Moreover, we adapt the editors for spatial and contextual variability to be evolution-aware. In our metamodels, the whole evolution history is preserved and we add a user interface integration which allows to show the state of an SPL for arbitrary dates. These dates can be in the past, which allows for restoring states of previous versions of SPLs, and the dates can be in the future, which allows for planning future evolution of an SPL. Moreover, we provide support for generating an evolution report, which is an overview of evolution steps that happened between two dates.

With spatial, contextual and temporal variability, complex scenarios of the conceptual side of SPLs can be defined. However, this still lacks the possibility to generate variants.



(a) Before evolution (b) After evolution

Figure 1: Feature models of the running example

This results in **R4**:

R4: Allow variant generation integrating the three dimensions of variability

In DarwinSPL, we realize this by creating configurations in terms of feature selections for each point in time and, thus, respecting temporal variability. Moreover, with the HyVarRec integration, we can create configurations for each context. To generate variants out of these configurations, we provide an integration with DeltaEcore [2], a tool suite for modeling delta-oriented SPLs which also allows variant generation by applying delta modules.

2. BACKGROUND

SPLs are an approach to capture commonalities and variabilities of variants of software systems. On the conceptual side, these variants differ in terms of functionality or features. On the implementation side, different implementation artifacts can be used to generate the resulting variant.

Multiple notations exist to capture the conceptual side of an SPL. One of them are feature models, which structure features hierarchically in a tree-like notation [13]. Each feature can only be selected if its parent feature is selected. Features have *variantion types*: either *optional* or *mandatory*. If a feature is mandatory, it has to be selected, if its parent is selected. Moreover, features can be organized in *or* or *alternative* groups. If a parent feature of a group is selected, at least one feature of this group has to be selected. In *alternative* groups, exactly one feature has to be selected. To provide more fine-grained variability of features than just (de)selection, *feature attributes* can be used [9]. Each feature attribute has a name and a type. As not each relation between features and feature attributes can be expressed via the feature model, cross-tree constraints (CTCs) allow to specify additional constraints using propositional formulas with features and feature attributes as variables [8]. Configurations of SPLs consist of a set of selected features and feature attribute value assignments. Configurations are valid if they do not contradict the feature model and CTCs.

On the implementation side, different variability realization mechanisms exist. A transformational variability realization mechanisms is *delta modeling* [11]. In delta modeling, artifacts are transformed by using delta operations, such as add, remove and modify. The delta operations are specified in a domain specific delta language. Delta modules consist of a set of delta operation calls to modify a base artifact. Delta modules and the base artifact represent the artifacts of implementation side of an SPL. To create a variant using a configuration, a mapping from features to implementation artifacts is necessary. In a variant generation process, after specifying a configuration, relevant delta modules are collected, brought into a suitable order and are applied to a base artifact.

3. DARWINSPL

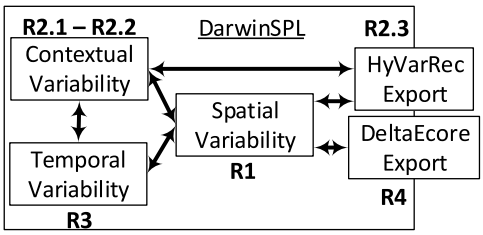


Figure 2: Architecture of DarwinSPL

Figure 2 depicts the general architecture of DarwinSPL. With several editors, it is possible to define spatial, contextual and temporal variability. As the three dimensions of variability strongly correlate, most of the editors are referencing or editing more than one dimension of variability at the same time. We provide an export to HyVarRec for contextual and spatial variability to reconfigure SPLs based on the context. With DarwinSPL, variants can be generated. To this end, we provide an export to DeltaEcore which is responsible for the variant derivation process using spatial variability. In the following, we will show how DarwinSPL fulfills the requirements of Section 1.2.

3.1 Modeling Spatial Variability

DarwinSPL uses feature models to capture spatial variability and, thus, the configuration space. To this end, we provide an editor, which can be used to graphically define feature models, as Figure 3 depicts. The feature model can be structured arbitrarily, features and groups can be created or removed and their variation types can be changed. Additionally, to be able to specify more fine-grained spatial variability, it is possible to define feature attributes. Figure 3 shows the feature `GearAdvice`, which has an attribute, indicating which gear advices should be used. Features and their attributes can be named arbitrarily. Moreover, with drag and drop, it is possible to move features or complete groups. As layouting may be important to improve comprehensibility of feature models, we integrated the possibility to modify and save layouts of feature models.

As not each possible relation between features may be expressed by the structure of feature models, we provide a tex-

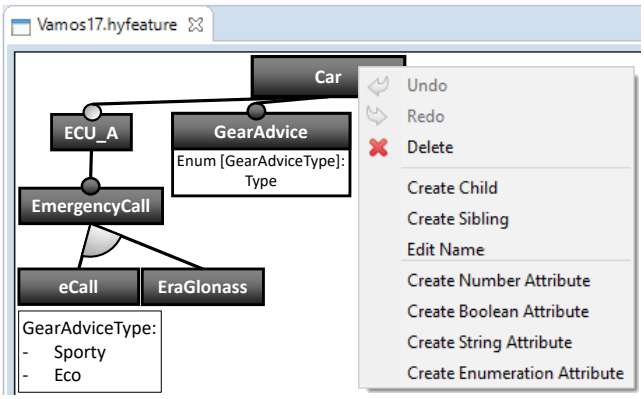


Figure 3: Feature Model Editor

```
Vamos17.hyconstraints
1 EraGlonass -> Glonass
2 eCall -> Glonass || GPS
```

Figure 4: Cross-Tree Constraint Editor

tual language and an editor to define CTCs, shown by Figure 4. Using this editor, it is possible to define propositional formulas with features and feature attributes as variables. With the feature model and CTC editors, DarwinSPL fulfills requirement **R1**.

3.2 Modeling Contextual Variability

We need to capture the influence of contexts on configuration options (**R2.2**) to reconfigure variants of SPLs based on contexts (**R2**). For that reason, we first need to be able to model all relevant contextual information (**R2.1**). Finally, we want to be able to reconfigure an end device's variant based on the defined rules (**R2.3**).

3.2.1 *Defining Contextual Information*

In our previous work, we defined a metamodel capturing all relevant contextual information [15]. Using this metamodel, we are able to define three different types of contextual information: Boolean types, e.g., if it is night, numerical types, e.g., the temperature and developer-defined enumeration types, e.g., the day of the week. For each contextual information, a domain can be specified. For Boolean types, the domain is fixed to *true* and *false*. However, for numerical types, a minimum and maximum can be specified and for enumeration types, all available literals need to be specified.

In DarwinSPL, we realized this by defining a grammar to specify this information and provide an editor using this grammar. Figure 5 shows how the three types of contextual information can be defined. To be able to define enumeration typed contextual information, a corresponding enumeration and available literals have to be defined first. In lines 1 – 2, we define an enumeration `LocationEnum` which has two possible values: `Europe` and `Russia`. In line 3, we define an enumeration typed contextual information `Location`, representing the current location of a car, which uses `LocationEnum` as value domain. In line 4, we define a numerical contextual information, representing the number of times a driver shifted gears too late, with a minimum of 0 and a maximum of 10. To illustrate the definition of Boolean typed contextual information, in line 5, we define `isNight`, which represents if it is currently day or night. With this editor, DarwinSPL fulfills requirement **R2.1**.

```
Vamos17.hycontextinformation
1 Enum(LocationEnum, EnumLiteral(Europe, 0),
2      EnumLiteral(Russia, 1))
3 EnumContext(Location, LocationEnum)
4 NumberContext(GearViolation,0,10)
5 BooleanContext(isNight)
```

Figure 5: Editor for Contextual Information

```
Vamos17.hyvalidityformula
1 eCall:
2       context:Location = enum:LocationEnum.Europe
3 EraGlonass:
4       context:Location = enum:LocationEnum.Russia
```

Figure 6: Editor for Validity Formulas

3.2.2 *Defining Contextual Impact*

After having modeled contextual information, it is necessary to model influences of contexts on configuration options and, thus, on features. To this end, we introduced a concept of Validity Formulas (VFs) and a metamodel for them in our previous work [15]. With VFs, it is possible to specify in which context features are selectable. Each VF consists of feature reference and a formula representing the context in which the respective feature is selectable. In DarwinSPL, we defined a grammar and a respective editor for VFs. Figure 6 shows the editor for VFs in which we defined two VFs for the running example. The first VF is defined for the feature `eCall` (lines 1 – 2), which specifies that `eCall` can only be selected if the car is in `Europe`. The second VF (lines 3 – 4) is similar to the first but with `EraGlonass` instead of `eCall` and `Russia` instead of `Europe`.

However, with VFs we are only able to specify under which conditions a feature is selectable. To enforce a feature selection based on the context, we extend the grammar for CTCs by allowing to use contextual information. Figure 7 shows a screenshot from the editor for CTCs with a new constraint, defining that if drivers shift too late more than five times, the `GearAdvice`'s type has to be set to `Eco`. With the editor for VFs and context-aware CTCs, we are able to capture influences of contexts on configuration options. Thus, DarwinSPL fulfills requirement **R2.2**.

3.2.3 *Automatic Reconfiguration*

With requirements **R2.1** and **R2.2**, we modeled contexts and their influence on configuration options. However, when the contexts of systems change, it may be necessary to reconfigure deployed variants based on VFs and context-aware CTCs. In previous work, we implemented a reconfiguration engine as web service for context-aware SPLs called HyVarRec [15]. In DarwinSPL, we provide an exporter to HyVarRec and the possibility to simulate a reconfiguration process by allowing users to provide a configuration and values for contextual information. HyVarRec then computes a new configuration, which satisfies all CTCs and VFs. The new configuration and is as similar as possible to the previous configuration. With the integration with HyVarRec, we are able to automatically reconfigure variants of SPLs based on the current context and, thus, fulfill requirement **R2.3**.

3.3 Modeling Temporal Variability

Evolution of an SPL is captured in temporal variability (**R3**). As evolution is a cross-cutting concern, each element of an SPL may be affected. Thus, it is important to be able

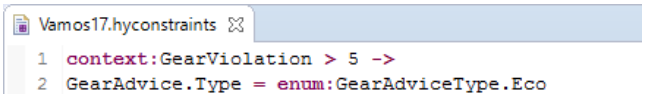


Figure 7: Context-Aware CTC Editor

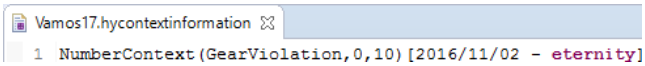


Figure 8: Editor with Temporal Validities

to perform consistent evolutionary changes to each element (**R3.1**). Moreover, we need to keep track of the already performed evolution and to plan future evolution (**R3.2**). As evolution may lead to very complex models, we have to provide an overview of the performed evolution steps (**R3.3**).

3.3.1 Performing Evolutionary Changes

With the notion of temporal elements, it is possible to assign *temporal validities* to each evolvable element. A temporal validity ϑ is defined as interval over points in time: $\vartheta = [\vartheta_{since}, \vartheta_{until})$. Thus, it is possible to define, for each temporal element, time spans when they are available. For instance, temporal validities of features specify when they start and when they end to be available as configuration option. In our previous work, we applied the concept of temporal elements to feature models and presented Temporal Feature Models (TFMs) [16]. Thus, each evolvable element of a feature model is modeled as temporal element in TFMs.

DarwinSPL supports evolution of feature models by using TFMs. Validities of TFM elements can be set using the feature model editor. To not force developers to change their habitual behavior when modeling feature models, we developed the editor in such a way, that developers do not have to set validities actively. When performing editor operations, temporal variabilities are set automatically. For instance, if a developer adds a new feature, the feature is added to the model and ϑ_{since} is set to the current point in time. If a developer removes a feature, the feature is not removed from the model but its ϑ_{until} is set to the current point in time. This is done in the same way for all other evolvable elements of the feature model. Thus, each modification of the feature model is translated to an evolution step associated with the current point in time. However, it may be necessary to remove an element completely from the model, e.g., if a new feature was added accidentally. To this end, we provide a context menu entry to delete an element from the model.

For CTCs, contextual information and VFs, DarwinSPL provides textual editors. For these textual editors, we need to apply different procedures to model evolution. Thus, we extended the used grammars. With these extensions, specifying temporal validities for each CTC, contextual information and VF explicitly is possible. As Figure 8 depicts, we can specify temporal validity at the end of each element in square brackets. In the example, we specify that the contextual information `GearViolation` is added November 2, 2016 and is valid forever since then (specified by the `eternity` keyword). If no temporal validity is specified, we assume that the respective element is valid at each point in time. The specification of temporal validities of CTCs and VFs is implemented in the same way for their respective editors.

In textual editors, elements of the feature model are referenced by their names. However, names may change during evolution. This means that, for each single point in time, names are unique but over the course of time they are not.

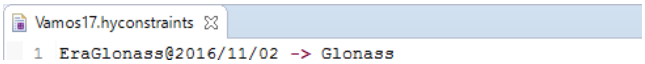


Figure 9: Editor with Evolution-Aware Referencing

Thus, a simple reference by name in the textual editor may be ambiguous. To overcome this, we provide the possibility for evolution-aware referencing which allows to optionally specify a point in time. As Figure 9 depicts, we can reference elements via a combination of a name and a point in time. In the example, we reference the feature which is named `EraGlonass` at November 2, 2016. As names are unique for each single point in time, we have the possibility of unambiguous references with evolution-aware referencing.

With the possibility to specify temporal validities for elements of spatial and contextual variability, we can perform arbitrary evolutionary changes on context-aware SPLs modeled with DarwinSPL and, thus, fulfill requirement **R3.1**.

3.3.2 Tracking and Planning Evolution

Requirement **R3.2** states the necessity to track the whole evolution history and to plan future evolution of an SPL. As we are using the notion of temporal elements for all models of DarwinSPL, we can capture the whole evolution history of an SPL. However, we still need support in the user interface to obtain previous feature model states and to associate feature model modifications with points in time in the future.

For the graphical feature model editor, it is important that temporally non-valid elements are not shown as these elements may reduce clarity. Therefore, the editor only shows elements which are temporally valid at the current point in time. We extend the editor to be evolution-aware, allowing retrieval of previous states and evolution steps of the feature model. To this end, we provide an *evolution slider* which allows to jump to arbitrary points in time. When opening a feature model, the editor extracts the points in time at which the feature model evolved and adds respective dates for the slider. This allows developers to view the whole evolution history of the feature model. Using the slider, different states of feature models are extracted from the integrated notation underneath and presented. Figure 10 shows the feature model editor with the evolution slider. Directly under the editing pane, a button shows the currently selected date. Pressing it will allow to select another date. However, this still lacks the ability of future planning of SPLs. To overcome this, we add the possibility to add new dates to the slider manually. New dates can be added using a button to the right of the slider which opens a date picker. In the running example, first there was just one ECU, `ECU_A`. During evolution, `ECU_B` was added. Figure 11 shows the editor after moving the slider to November 2, 2016, which then shows the second ECU with its child features. With the "Add Date" button, it is also possible to add future dates to allow future evolution of the feature model. Modifications of the feature model are automatically associated with the currently selected point in time of the slider.

In textual editors for CTCs, contextual information and VFs, each entry is always visible, independently of its temporal validity. Additionally, it is possible to define temporal validities with points in time in the future. With the evolution slider and temporal validities in the textual editors, it is possible to track the whole evolution and to plan future evolution steps of an SPL, which fulfills requirements **R3.2**.

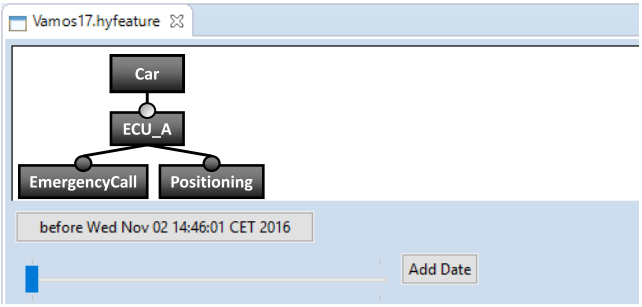


Figure 10: Evolution Slider of Feature Model Editor before Evolution

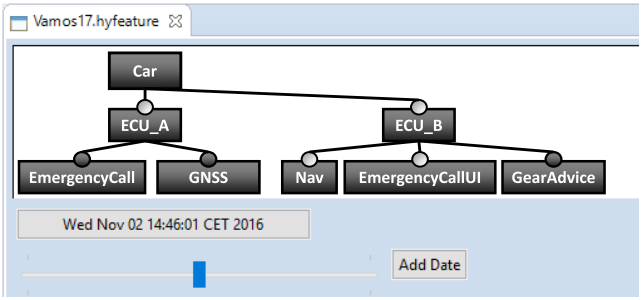


Figure 11: Evolution Slider of Feature Model Editor after Evolution

3.3.3 Providing Evolution Overview

During the lifecycle of SPLs, many evolution steps may have been performed. When analyzing evolution of SPLs, it may be hard to understand which elements evolved in which way, especially for SPLs with large feature models. To keep an overview of the feature model evolution, we implemented several mechanisms with DarwinSPL. The first mechanism is the evolution slider which allows to switch between dates. Using this mechanism, the result of the performed evolution between the switched dates becomes visible. However, not each evolution step may be observable appropriately. For instance, using the slider, it is not possible to distinguish between renaming a feature or removing the old feature and adding a new feature with the new name. Additionally, for big feature models, one may overlook changes when using the slider. To keep an overview of names of a feature, we provide an extra "Name History" view. Figure 12 depicts the overview for the feature which was initially named `Positioning` and then renamed to `GNSS`. Apart from providing an overview of the history of all names of a feature, it is possible to modify these names and their temporal validities.

DarwinSPL can generate an evolution report which is a compact overview of feature model evolution. This report is subdivided into dates for which an evolution step occurred. For each date, a list of evolution changes performed at that date is presented. The report is a website and allows to narrow down the evolution history by setting a start and end-date.

For CTCs, contextual information and VFs, the textual editors do not hide any elements and, thus, an overview of the evolution of these artifacts is provided. Using the evolution slider, the name overview and the evolution report, DarwinSPL provides mechanisms to give a detailed overview of the evolution history of the feature model. Thus, the textual editors, DarwinSPL fulfills **R3.3** partially but for the evolution of the feature model, it fulfills **R3.3**.

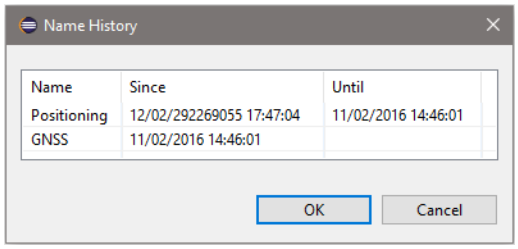


Figure 12: Overview of Name Evolution of a Feature

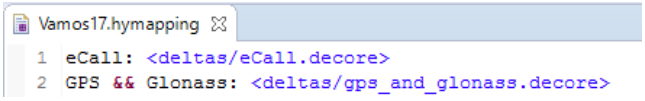


Figure 13: Screenshot of Mapping Editor

3.4 Supporting Variant Generation

In previous sections, we described how we realized modeling the conceptual side of an SPL. Generating variants is possible by composing implementation artifacts based on a configuration (i.e., feature selection and feature attribute value assignment) (**R4**). This requires the definition of a configuration, suitable implementation artifacts and a mapping between feature selections and these artifacts.

To satisfy **R4**, we provide a configurator to define configurations that integrates with DeltaEcore, a tool suite for model-based definition of delta-oriented SPLs. In the configurator, it is possible to select features by double clicking them and to set values for feature attributes. The variant generation process can use this configuration. Additionally, this process requires implementation artifacts and a mapping between features and the implementation artifacts.

With DeltaEcore, it is possible to define delta languages for arbitrary Ecore metamodels. Using delta languages, delta modules can be specified which we use as implementation artifacts for DarwinSPL. Given a set of delta modules, DeltaEcore is able to generate variants. To integrate with DeltaEcore, we provide the possibility to define mappings between feature selections and delta modules consisting of two parts: an application condition that defines under which conditions mappings should be activated and a set of delta modules which should be applied when the mapping is activated. In DarwinSPL, we provide a textual editor to define mappings. Application conditions can be specified using arbitrary propositional formulas with features and feature attributes as variables. Figure 13 depicts an exemplary mapping. If the feature `eCall` is selected, the `eCall` delta module is applied and if `GPS` and `Glonass` are selected together, the `gps_and_glonass` delta module is applied.

After having specified a configuration and a mapping to existing delta modules, all necessary artifacts to generate the respective variant are available. The variant generation can be started in the configurator. DarwinSPL automatically translates the feature model, the configuration and the mapping to a native configuration and mapping of DeltaEcore. DeltaEcore evaluates the mapping using the configuration, selects corresponding delta modules and applies them. However, DeltaEcore does not support feature attributes. Therefore, this approach is currently limited as we can not use feature attributes to generate variants.

Temporal variability also affects configurations and mappings. At a certain point in time, in the configurator, it should only be possible to create configurations using fea-

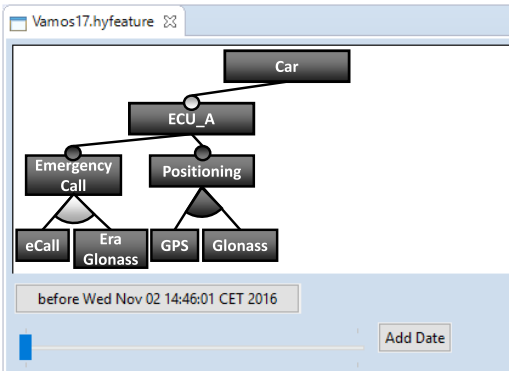


Figure 14: Feature Model Editor before Evolution

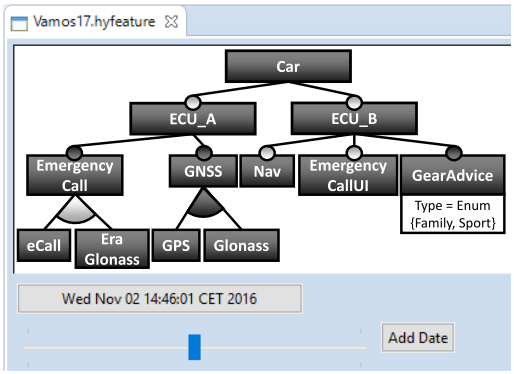


Figure 15: Feature Model Editor after Evolution

tures and attributes temporally valid at that point in time. Thus, in the configurator, we use the same evolution slider as for the feature model editor. Using the slider allows to only select features and assign values for feature attributes for features which are temporally valid at the selected point in time. This prevents developers from creating configurations with elements which are not temporally valid.

Evolution of feature models involves evolution of mappings. Newly added features require a mapping and for removed features, no mapping should be temporally valid anymore. Additionally, the mapping itself evolves, too, as new or different implementation artifacts need to be mapped. To support evolution of the mapping, we use the same approach to specify temporal validities for each mapping as for CTCs and VFs. Being able to define configurations, a mapping and integrating with DeltaEcore fulfills requirement **R4**.

4. CASE STUDY

To show feasibility of using DarwinSPL to model evolving context-aware SPLs, we implemented a case study provided by our industry partner using DarwinSPL for the conceptual side of the SPL. In Section 1.1, we described the use case of our industry partner. The single emergency call ECU had to be modeled as SPL. As the system has to always use the correct emergency call system based on the country the car is currently in, the environment's impact on the system needed to be modeled. On the implementation side, we used delta modules created using DeltaEcore.

Figure 14 depicts the feature model for the first ECU before evolution, modeled using the feature model editor. Figure 16 depicts the CTCs, contextual information and VFs, modeled using the textual editors. For brevity, we only show the state of the textual editors after evolution. The elements added during evolution are annotated with an explicit temporal validity. With the `Location` as enumeration typed contextual information with `Russia` and `Europe` as possible values, we captured the car's current geolocation. Additionally, we added VFs to model the location's impact on the SPL. Thus, `eCall` can only be selected if the car is located in `Europe` and `EraGlonass` if the car is in `Russia`.

Furthermore, we generated a variant of the SPL with `eCall`. Then, we simulated the car's traversal to Russia's border, exported to HyVarRec and started a reconfiguration process, which successfully provided a new valid configuration with `EraGlonass`.

As our industry partner had to evolve the SPL and model a second ECU, we modeled this with DarwinSPL as well.

Vamos17.hyconstraints

```
1 EraGlonass -> Glonass
2 Nav -> GNSS@2016/11/03 [2016/11/02 - eternity]
3 EmergencyCallUI -> EmergencyCall [2016/11/02 - eternity]
```

Vamos17.hycontextinformation

```
1 Enum(LocationEnum,EnumLiteral(Europe,0),
2      EnumLiteral(Russia,1))
3 EnumContext(Location, LocationEnum)
4 NumberContext(GearViolation,0,100)
```

Vamos17.hyvalidityformula

```
1 eCall: context:Location = enum:LocationEnum.Europe
2 EraGlonass: context:Location=enum:LocationEnum.Russia
3 GearAdvice: context:GearViolation > 9 ->
4  GearAdvice.Type=enum:Type.Family[2016/11/02-eternity]
```

Figure 16: Textual Editors after Evolution

Using the evolution-aware temporal feature model editor, we added a new date to the slider (November 2, 2016) and modified the feature model: we added four new features, renamed a feature, added an enumeration typed feature attribute and an enumeration for it. Moreover, new CTCs had to be added. In the editor for CTCs, we added two new CTCs and set their temporal validity's begin to November 2, 2016. We did the same for new contextual information (`GearViolation`) and a new VFs. Figure 15 depicts the resulting feature model in its editor and Figure 16 shows the resulting state of the textual editors. With the new SPL version and new implementation artifacts, we were able to generate variants using new features. By using the evolution slider in the configurator, we were able to generate new variants for the old single ECU. In summary, we were able to implement a context-aware SPL using DarwinSPL. Moreover, with the evolution-aware editors, we have been capable to model evolution of the SPL. Using the evolution slider, we kept track of evolution to generate variants of the old version of the SPL and to only generate variants with elements which are temporally valid. Additionally, we employed HyVarRec to reconfigure variants based on different contexts. One drawback are the textual editors, as their syntax is not very intuitive and even we as their developers had to check the correct syntax when developing this SPL.

5. RELATED WORK

There are many tools supporting SPL definition but in the context of this paper, not all of them can be discussed. We will discuss some of these tools in the following: FeatureIDE [3] is an open-source tool suite for SPL engineering. It supports standard feature model definition. With

FeatureIDE, it is possible to define implementation artifacts for different variability realization mechanisms. There is no support for context-aware SPLs or evolution.

FeatureHouse is an approach for composing software artifacts by superimposition [7]. With FeatureHouse feature structure trees (FSTs) can be used to represent spatial variability of an SPL. As each feature in the FST is associated with one implementation artifact, abstraction is limited. To overcome this, FeatureHouse can be combined with FeatureIDE and its feature models. Moreover, context-aware SPLs and evolution cannot be modeled.

Pure:Variants [4] and Gears Product Line Engineering Tool and Lifecycle Framework™ [5] are industry-scale tool suites for SPL development from requirements engineering to variant generation. However, both do not support context-aware SPLs or evolution.

DeltaEcore is a tool suite, which supports the definition of delta-oriented SPLs, which we introduced in Section 2. DeltaEcore addresses temporal variability in terms of Hyper Feature Models (HFMs), which capture feature versions which can be mapped to different implementation artifacts. However, it is not possible to model evolution of feature models themselves and CTCs or context-aware SPLs.

EvoFM supports evolution by providing evolution operations on the feature model as first-class entities [10]. The methodology of EvoFM is very dependent on the evolution operations and, thus, not very flexible. Evolution is limited to feature models and no support for context-aware SPLs is available. Temporal elements are independent of evolution operations and can model arbitrary evolution. To the best of our knowledge, there is no implementation of EvoFM and no integration with variant realization mechanisms available.

MOSKitt4SPL [6] uses feature models for the conceptual side of SPLs. It supports the definition of arbitrary contexts and reconfiguration rules with a finite automaton. However, there is no integration with variant realization mechanisms. Moreover, evolution of an SPL cannot be modeled.

6. CONCLUSION

DarwinSPL is a tool suite for integrated modeling of spatial, contextual and temporal variability using a set of provided editors. As temporal variability is a cross-cutting concern, we implemented support in all editors of DarwinSPL to handle evolution which allows to keep track of the whole evolution and to plan future evolution. With DarwinSPL we support the whole SPL development process, as we integrate with DeltaEcore to create implementation artifacts, map them to feature models and generate variants. Moreover, as contextual variability may require reconfiguration of variants due to changed contexts, we integrate with HyVarRec, a reconfiguration engine for SPLs. Using the case study of our industry partner, we showed the feasibility to develop evolving context-aware SPLs with DarwinSPL.

In our future work, we want to improve DarwinSPL to better integrate all editors and to merge them to one editor. Moreover, currently DeltaEcore is responsible for evaluating configurations and mappings which we will implement for DarwinSPL to support feature attributes. As the three dimensions of variability also introduce more complexity, we plan to implement several analyses by deeper integration with HyVarRec, which already supports many analyses. Finally, we will integrate more comfort functionalities and a more appealing user interface.

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future — Managed Software Evolution and by the European Commission within the project HyVar (grant agreement H2020-644298).

7. REFERENCES

[1] https://github.com/HyVar/hyvar-rec.
[2] http://deltaecore.org.
[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/\featureide/.
[4] https://www.pure-systems.com.
[5] http://www.biglever.com/.
[6] https://tatami.dsic.upv.es/moskitt4spl/.
[7] S. Apel, C. Kastner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *Proc. of the 31st Intl. Conference on Software Engineering*, ICSE, pages 221–231, 2009.
[8] D. Batory. *Feature Models, Grammars, and Propositional Formulas*, pages 7–20. Springer Berlin Heidelberg, 2005.
[9] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, volume 3520 of *LNCS*. Springer Berlin Heidelberg, 2005.
[10] G. Botterweck, A. Pleuss, A. Polzer, and S. Kowalewski. Towards feature-driven planning of product-line evolution. In *Proc. of the 1st Intl. Workshop on Feature-Oriented Software Development*, FOSD, pages 109–116, 2009.
[11] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *Proc. of the 9th Intl. Conference on Generative Programming and Component Engineering*, GPCE, pages 13–22, 2010.
[12] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *Proc. of the 12th Intl. Software Product Line Conference*, SPLC, pages 12–21, 2008.
[13] 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.
[14] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. of the IEEE*, 68(9):1060–1076, 1980.
[15] J. Mauro, M. Nieke, C. Seidl, and I. C. Yu. Context aware reconfiguration in software product lines. In *Proc. of the 10th Intl. Workshop on Variability Modelling of Software-intensive Systems*, VaMoS, pages 41–48, 2016.
[16] M. Nieke, C. Seidl, and S. Schuster. Guaranteeing configuration validity in evolving software product lines. In *Proc. of the 10th Intl. Workshop on Variability Modelling of Software-intensive Systems*, VaMoS, pages 73–80, 2016.
[17] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.