Integrated Management of Variability in Space and Time in Software Families (Short Version)

Dipl.-Inf. Christoph Seidl (July 15, 2015)

Abstract Software Product Lines (SPLs) and Software Ecosystems (SECOs) are approaches to capturing families of closely related software systems in terms of common and variable functionality (variability in space). SPLs and especially SECOs are subject to software evolution to adapt to new or changed requirements resulting in different versions of the software family and its variable assets (variability in time). Both dimensions may be interconnected (e.g., through version incompatibilities) and, thus, have to be handled simultaneously as not all customers upgrade their respective products immediately or completely. However, there currently is no integrated approach allowing variant derivation of features in different version combinations. In this thesis, remedy is provided in the form of an integrated approach making contributions in three areas: (1) As variability model, Hyper-Feature Models (HFMs) and a version-aware constraint language are introduced to conceptually capture variability in time as features and feature versions. (2) As variability realization mechanism, delta modeling is extended for variability in time, and a language creation infrastructure is provided to devise suitable delta languages. (3) For the variant derivation procedure, an automatic version selection mechanism is presented as well as a procedure to derive large parts of the application order for delta modules from the structure of the HFM. The presented integrated approach enables derivation of concrete software systems from an SPL or a SECO where both features and feature versions may be configured.

1 Introduction

Software Product Lines (SPLs) and Software Ecosystems (SECOs) treat a set of closely related software systems not as individuals but as software family. This reduces development cost and maintenance effort while still permitting tailored software systems for small groups of users [22]. Examples of software families are the Eclipse IDE[1], the Android mobile operating system[2] and the Linux kernel[3]. Software families allow creation of different products by enabling or disabling configurable functionality (*variability in space*), which yields *variants* of the software family. Furthermore, they are subjected to software evolution when adding new functionality or fixing defects (*variability in time*), which yields *versions* of assets of the software family (e.g., source code, design models or documentation).

In some cases, these dimensions may be treated in isolation to ease development, e.g., when configuration knowledge is maintained by a single institution or when solely ready-made products are shipped to customers. However, for software families with multiple contributors and unsynchronized development cycles that allow creation of products by end-users (such as Eclipse, Android and the Linux Kernel), variability in space and time cannot be handled in complete isolation: Changes as part of software evolution may alter dependencies on certain functionality, and enabling particular functionality may create dependencies on or incompatibilities with certain version ranges of other functionality.

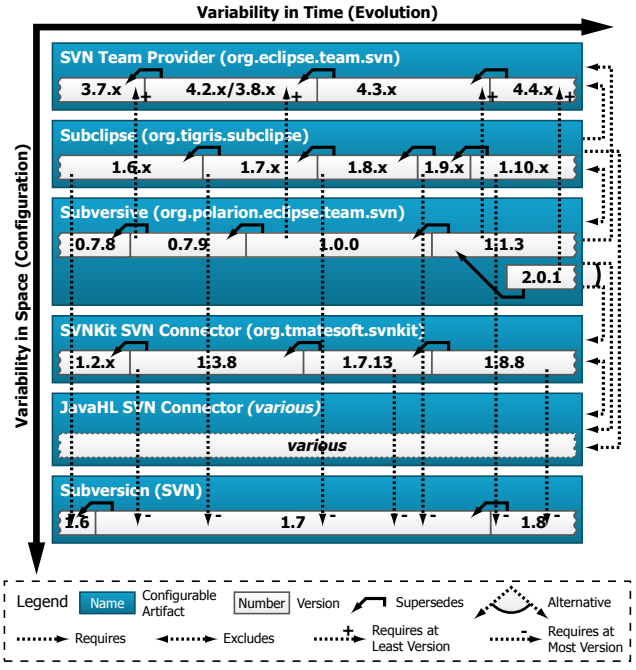

Fig. 1 The Eclipse SECO is an example of a software family subjected to variability in space and time where the two dimensions are interconnected so that they need integrated management.

Christoph Seidl
Technische Universität Dresden, Germany
E-mail: christoph.seidl@tu-dresden.de

[1] `http://eclipse.org`
[2] `http://android.com`
[3] `https://kernel.org`

Fig. 1 illustrates the interdependence of variability in space and time using the two SVN plug-ins Subclipse and Subversive of the Eclipse SECO and related artifacts with their dependencies on different functionality in different versions. The handling of variants and versions is relevant on both conceptual and realization level to cope with configuration knowledge and derivation of executable software systems, respectively.

To illustrate the problems and solutions of this thesis, a running example of suitable size is utilized: The TurtleBot is a domestic service robot. The software technology group of TU Dresden has developed a driver software for the robot to control its operation. Due to individual robots' different hardware configurations and their limited resources (e.g., CPU or battery life), the driver is maintained not as a monolithic software system but, instead, as a family of configurable software systems that allows derivation of custom-tailored drivers. As the driver software permits definition of configurations by end-users and is being developed by multiple loosely connected parties, the dimensions of variability in space and time are interconnected and need integrated management.

The driver software consists of an `Engine` and a controller for the `Movement`, which provide access to locomotion functionality on a physical and logical level, respectively. Movement may be controlled manually by `Keyboard` and `Gamepad` or using `Autonomous` operation of the robot. The prior requires a `Webservice` to be selected for remote communication with the robot. The latter needs a `Detection` mechanism for obstacles that can be realized by a `Bump` sensor, which is triggered on impact, as well as an `Infrared` or an `Ultrasound` sensor, which are triggered over distance.

During the period of developing the driver software, the TurtleBot's possible hardware configurations were altered as part of evolution, i.a., by providing the option to equip the robot with a different type of engine. Furthermore, development on the software components of the driver yielded new versions that declared different dependencies with regard to features and even feature versions. These versions of configurable assets have to be maintained as not all robots are updated simultaneously or completely. However, current approaches of managing software families cannot cope with the notion of evolution so that these versions and their interdependencies cannot be represented as part of the configuration knowledge. Neither can they be used to create individual software systems with combinations of configurable functionality in different versions (see Fig. 2).

To remedy these problems, this thesis devises an integrated approach for managing variability in space and time within software families. It extends established approaches for handling variability in space to make them suitable for coping with variability in time. It makes contributions in three areas: a variability model, a variability realization mechanism and a variant derivation procedure (see Fig. 3).

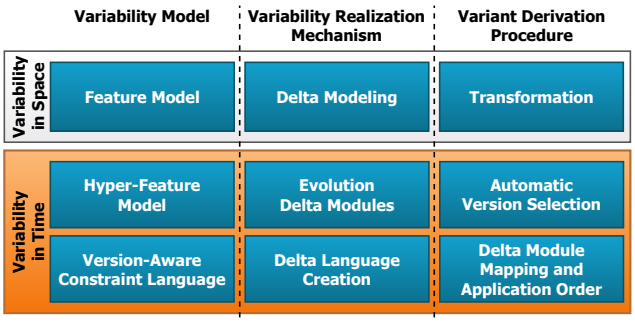

Fig. 3 The thesis extends established technologies for coping with variability in space to be applicable for variability in time and makes contributions in three areas.

Sec. 2 provides information on the approaches for handling variability in space. The following sections explain the individual contributions: Sec. 3 introduces Hyper-Feature Models (HFMs) and a version-aware constraint language as variability model. Sec. 4 extends delta modeling by evolution delta modules and presents a delta language creation infrastructure for the variability realization mechanism. Sec. 5 elaborates on a variant derivation procedure combining the devised variability model and variability realization mechanism. Sec. 6 examines feasibility of the concepts in an evaluation encompassing 3 case studies. Sec. 7 discusses related work and Sec. 8 concludes by summarizing the work of the thesis.

2 Foundations

When employing a structured reuse approach, a software family encompasses configuration knowledge on a conceptual level (*problem space*) and a realization level (*solution space*). The prior is suitable for communication with non-technical stakeholders (e.g., managers or customers), checks on consistency and validity as well as analyses. The latter is required for a technical realization as implementation of a software system.

On a conceptual level, a *variability model* represents the configurable functionality of a software family along with configuration rules governing valid combinations. A *configuration* constitutes a valid selection of configurable elements from the variability model.

On a realization level, a *variability realization mechanism* manifests the changes associated with configurable

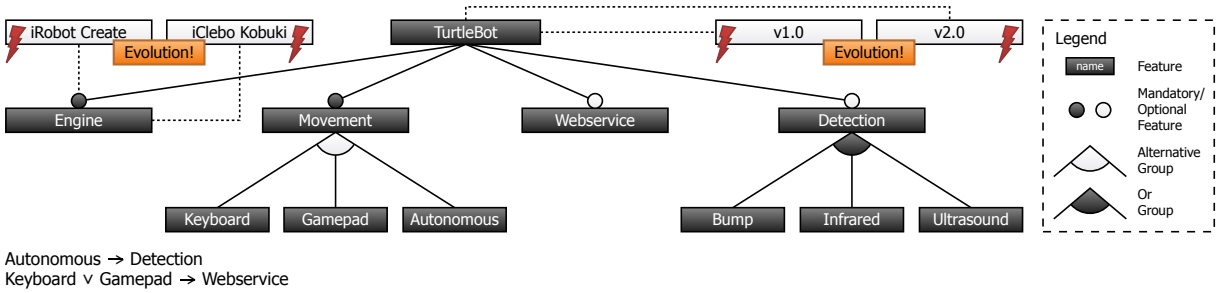

Fig. 2 Evolution yields new versions of features with interdependencies and incompatibilities that can neither be represented in feature models nor be used for variant derivation in existing approaches for software families.

functionality in realization artifacts such as source code (e.g., by extending a class with new methods required by the configured functionality).

A *variant derivation procedure* is utilized to create a concrete software system as *variant* or *product* of the software family. As part of this procedure, a conceptual configuration is provided to the variability realization mechanism which builds the corresponding realization assets with the selected functionality for the individual software system.

These procedures focus on configuration to handle variability in space but are not suitable for integrated management of variability in time. Hence, the thesis extends *feature models* as variability model and *delta modeling* as variability realization mechanism to make them suitable for coping with variability in time. Both these approaches are explained in the following sections.

2.1 Feature Model

A *feature model* [15,6] specifies configuration knowledge of a software family as a tree that decomposes functionality into individual *features* (see Fig. 2). A *feature* is "a logical unit of behaviour specified by a set of functional and non-functional requirements" [3] and represents the atomic unit of a configuration. A feature's *variation type* may be either *mandatory* or *optional*, which requires the feature to be selected or allows its deselection, respectively. A feature's variation type is only evaluated if its parent feature is selected. The root feature is implicitly perceived as being mandatory. Furthermore, features may be contained in groups with variation types making them *alternative* or *or* groups, which require selection of exactly one or at least one of the contained features, respectively.

In addition to the configuration rules imposed by the structure of the feature model, additional *cross-tree constraints* may further restrain configuration options. One possible representation is Boolean logic over features (see Fig. 2) where references to features are evaluated to `true` if and only if a feature is selected. A constraint is satisfied if and only if it holds for a concrete selection of features.

A *configuration* of a feature model is a subset of all features that satisfies the configuration rules of the feature model's structure and its cross-tree constraints. The semantics of a feature model are defined in terms of all its valid configurations.

2.2 Delta Modeling

Delta modeling is an approach to manage variability in software families based on transformations [27,5]. It uses operations that add, remove or modify assets to transform one product of a software family into another product conforming to a certain valid configuration. For these purposes, a *source language* (e.g., Java) is augmented with a *delta language* (e.g., DeltaJava [27, 18]), which defines *delta operations* as dedicated domain-specific modification operations for the source language. A (configuration) *delta module* encapsulates calls to these delta operations to perform changes on realization assets that enable or disable configurable functionality (see Lst. 1). Delta languages have explicitly limited expressiveness to allow modifications related to configuration, but to reduce the risk of unintentionally harming system integrity. For example, identifiers of source language artifacts are usually perceived as being immutable [27] so that a delta language may not provide delta operations to change them.

In order to derive variants with delta modeling, all delta modules relevant for a configuration have to determined and applied in a suitable order. For these purposes, delta modeling may be coupled with a feature model where (logical expressions over) features are mapped to sets of relevant delta modules. Furthermore, delta modules may specify *application order constraints* listing other delta modules that have to be applied as predecessors, e.g., to avoid technical incompatibilities. During variant derivation, these explicit constraints on

```
configuration delta "Gamepad_Java"
  dialect <http://www.emftext.org/java>
  requires <../src/eu/vicci/turtlebot/Movement.java>
{
  Package p = <package::eu.vicci.turtlebot>;
  createClass("public class Gamepad {
                  //...
               }", p);

  Class gamepad = <class::eu.vicci.turtlebot.Gamepad>;
  Class movement = <class::eu.vicci.turtlebot.Movement>;

  //Set Movement as super class of Gamepad
  setSuperClassOfClass(movement, gamepad);

  //...
}
```

Lst. 1 Configuration delta modules enable or disable configurable functionality. The example enables the feature `Gamepad` in Java source code.

the order of delta modules are evaluated by a topological sorting algorithm to bring the relevant delta modules into a suitable sequence. Finally, delta modules and their calls to delta operations are applied sequentially according to this order to transform a *base variant* of the software family into the *target variant* that constitutes the functionality of the selected configuration.

Delta modeling has been applied to a number of programming and modeling languages including Java [27, 18], Class Diagrams [26], Matlab/Simulink [13] and Component Fault Diagrams [28]. However, delta languages have to be created for each source language, which may be tedious and problematic if delta languages are incompatible due to different implementation technology. A common language creation infrastructure could ease the creation process and ensure technical compatibility of delta languages.

As delta modeling is based on transformations, it may handle foreseen changes in the course of configuration [27, 7] as well as, principally, unforeseen changes as needed for evolution (see Sec. 4). However, no approach exists so far that integrates a dedicated notion of configurable versions based on the variant derivation capabilities of delta modeling.

3 Variability Model

Due to the effects of evolution, a feature may be present in multiple versions. However, neither common feature models [15] nor attributed feature models [6] may capture these feature versions and their relations due to their restriction to variability in space. In order to remedy these shortcomings with regard to representing variability in time, the thesis introduces *Hyper-Feature Models (HFMs)* and a *version-aware constraint language*.

3.1 Hyper-Feature Model

Hyper-Feature Models (HFMs) are introduced as an extension to common feature models that defines a new first class entity—a *feature version*. A feature version represents a snapshot of its containing feature's realization at a given point in time. It is perceived as incremental to its predecessor versions. In consequence, versions are arranged along *development lines* that support branching. By supporting features and feature versions, HFMs capture both variability in space and time on a conceptual level in a unified notation. Fig. 4 depicts an example HFM in a graphical notation that shows features and feature versions for the driver software of the TurtleBot robot platform. The full version of the thesis further provides a rigid formal basis as well as a model-based implementation of HFMs.

HFMs refine the atomic unit of configuration from a feature in common feature models to a feature version. Hence, configurations no longer consist solely of features but rather contain features and feature versions. In consequence, the conditions for a valid configuration of a feature model are extended by the following three points:

1. For each selected version, the containing feature also has to be part of the configuration.
2. For each selected feature, there has to be exactly one version in the configuration.
3. All version-aware constraints have to be satisfied (see Sec. 3.2).

These points extend the conditions imposed on a configuration of a common feature model in order to cope with variability in time when using HFMs and the version-aware constraint language (see Sec. 3.2). The semantics of an HFM are, again, defined in terms of the configurations that can be derived from it. Fig. 4 demonstrates a valid HFM configuration with the highlighted features and feature versions.

3.2 Version-Aware Constraint Language

In addition to the configuration knowledge captured in the HFM, cross-tree constraints may be specified. To formulate these constraints also for aspects of variability in time, a dedicated *version-aware constraint language* is provided. It allows establishing constraints over versions and version ranges, e.g., to express that specific versions are incompatible with certain versions of another feature. The version-aware constraint language is based on Boolean logic over features and supports constructs for negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$) and equivalence ($\equiv$). Furthermore, it defines

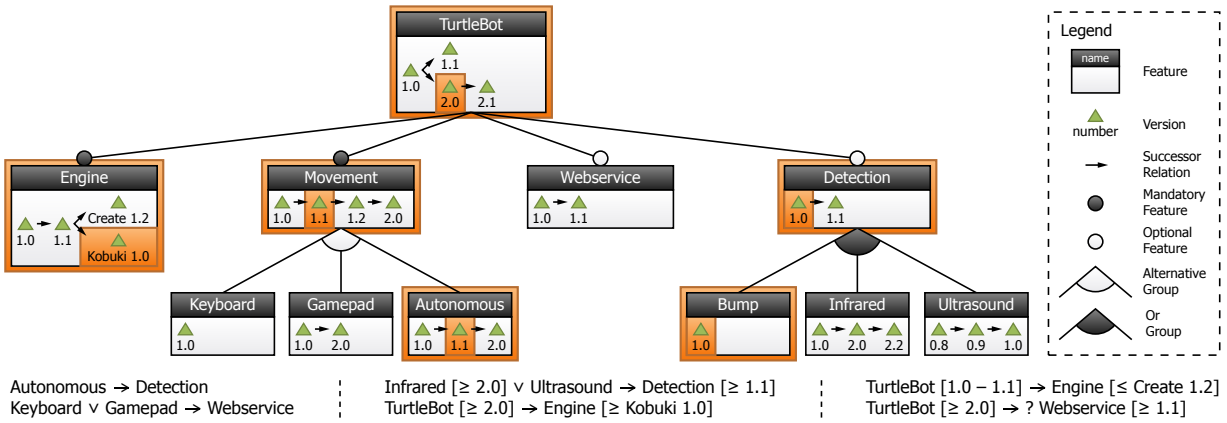

Fig. 4 Hyper-Feature Models (HFMs) capture variability in time on a conceptual level as feature versions, which are arranged along development lines. Version-aware constraints specify interdependencies and incompatibilities of versions and version ranges. A configuration of an HFM obeys the configuration rules of the feature model, contains exactly one version per selected feature and satisfies all version-aware constraints.

dedicated language constructs for *version restrictions* of individual features:

A *version range restriction* specifies a range of possible versions that satisfy the restriction by providing a lower and an upper version bound. This type of restriction is useful if both the upper and lower version bounds of the range are defined within the HFM. It has the form "feature [lower – upper]". For example, "TurtleBot $[1.0 - 1.1]$" is a version range restriction that is satisfied if a configuration contains the feature `TurtleBot` and one of the versions from the set {1.0 (TurtleBot), 1.1 (TurtleBot)}.

A *relative version restriction* defines a set of potential versions by means of a restriction in relation to one specific referenced version using an operator $op \in \{>, \geq, =, \leq, <\}$ interpreted as "greater than" etc. This type of restriction is useful if not necessarily all bounds of a version range are known, e.g., when expressing that a version should be "newer than" an existing version which should also include versions that are only added in the future. It has the form "feature [op version]". For example, "TurtleBot $[> 1.0]$" is a relative version restriction that is satisfied if a configuration contains the feature `TurtleBot` and one of the versions from the set {1.1 (TurtleBot), 2.0 (TurtleBot), 2.1 (TurtleBot)}.

Both version range restrictions and relative version restrictions may be used in a *conditional* form by prepending them with a question mark (?) as a convenience construct. In this case, the version restriction is only evaluated *if* the restricted feature is part of the configuration. This construct allows avoiding unintentionally making a feature mandatory through specifying a version restriction. For example, to not (unintentionally) make the feature `Webservice` mandatory for all configurations containing feature `TurtleBot` in at least version 2.0, a version restriction may be specified as conditional by "TurtleBot $[\geq 2.0] \rightarrow$?Webservice $[\geq 1.1]$" to only be evaluated if `Webservice` is selected.

With the version-aware constraint language, it is possible to formulate interdependencies and incompatibilities of different ranges of versions in order to cope with variability in time. Fig. 4 demonstrates the use of version-aware constraints for the TurtleBot driver software. The full version of the thesis further provides a rigid formal basis as well as a model-based implementation of the version-aware constraint language.

4 Variability Realization Mechanism

To manifest the changes associated with features and feature versions in realization assets (e.g., source code), the variability realization mechanism delta modeling [27] is employed. However, for the approach to be applicable, it has to be extended to cope with variability in time, and suitable delta languages have to be created for all source languages used in the realization assets (e.g., DeltaJava for Java). The following sections present solutions to both these challenges.

4.1 Evolution Delta Modules

Changes associated with variability in space and time may both be realized through transformation. Hence, delta modeling can, principally, represent these changes within delta modules [27,7,19]. However, despite the similarity of the changes to be performed, a distinction into *configuration delta modules* and *evolution delta modules* is required due to discriminating differences inherent to variability in space and time [20,29,23]:

1. **Intent**: A configuration delta module is employed to enable/disable functionality associated with (part of) a particular configuration whereas an evolution delta module is used to realize changes to meet new or altered requirements on the software system.
2. **Predictability**: A configuration delta module performs changes that yield an a priori known variant of the system whereas an evolution delta module performs changes that yield an a priori unknown version of the system.
3. **Expressiveness**: The expressiveness of a configuration delta language is intentionally limited to the purposes of configuration whereas that of an evolution delta module needs to be expressive enough to alter all parts of a system affected by evolution.

Analogously to the distinction of delta modules, delta operations are also distinguished into *configuration delta operations* and *evolution delta operations*. Configuration delta operations are intentionally limited in their expressiveness to not harm system integrity. In contrast, evolution delta operations may provide more powerful operations. Evolution delta operations may exclusively be used within evolution delta modules to update configurable functionality (see Lst. 2)

```
1  evolution delta "Engine_Create 1.2_Java"
2    dialect <http://www.emftext.org/java>
3    requires <../src/eu/vicci/turtlebot/Engine.java>
4  {
5    Class engine = <class::eu.vicci.turtlebot.Engine>;
6    Method driveMethod =
7      <method::eu.vicci.turtlebot.Engine#drive()>;
8
9    //Rename Engine to CreateEngine
10   renameNamedElement("CreateEngine", engine);
11
12   //Extract super class Engine from CreateEngine
13   extractSuperClass("Engine", engine, [driveMethod]);
14
15   //Make new Engine abstract
16   Class newEngine = <class::eu.vicci.turtlebot.Engine>;
17   setAbstractModifier(true, newEngine);
18   //...
19 }
```

Lst. 2 Evolution delta modules perform updates of configurable functionality. The example updates the Java source code of feature `Engine` to version Create 1.2.

4.2 Delta Language Creation

In order to manifest changes in realization assets using delta modeling, suitable delta languages for all source languages whose artifacts are affected by variability need to be supplied (e.g., DeltaJava for Java). For this purpose, the thesis presents a delta language creation infrastructure that allows devising delta languages for arbitrary source languages if they provide a metamodel based on EMF Ecore. This is feasible for arbitrary textual and graphical languages. Fig. 5 illustrates the general architecture of the delta language creation infrastructure.

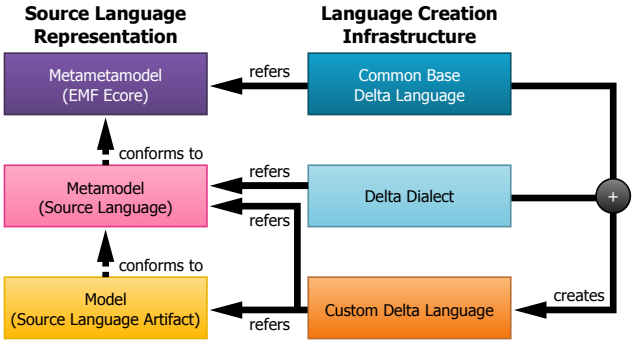

Fig. 5 The delta language creation infrastructure uses a language-agnostic common base delta language, which is extended by a language-specific delta dialect to retrieve a custom delta language for a source language given as metamodel.

A *common base delta language* encompasses all constructs of a delta language that can be specified without knowledge of the concrete source language, e.g., dependencies on other delta modules, variable declarations or use of identifiers. A *delta dialect* defines configuration and evolution delta operations for one specific source language presented as metamodel, e.g., to add and remove certain elements. When a delta module modifies an artifact of a specific source language, the respective delta dialect may be combined dynamically with the common base delta language to form a *custom delta language* suitable for the source language.

As certain delta operations are common to most existing delta languages [27,18,26,13,28], the thesis defines signatures and semantics of seven types of *standard delta operations* to set and unset single values; add, insert and remove elements in (possibly ordered) sets of values; modify attribute values and detach elements from their container. To use these operations within a delta dialect, they have to be instantiated for the elements of a particular source language, e.g., to add a method to a class in Java. As the metamodel of the source language contains the general structure of the source language, the thesis defines a procedure to instantiate a large number of these operations fully automatically for a specific source language. Furthermore, *custom delta operations* permit arbitrary signatures and semantics so that specific operations may be realized, e.g., by utilizing domain knowledge to maintain well-formedness of a realization artifact.

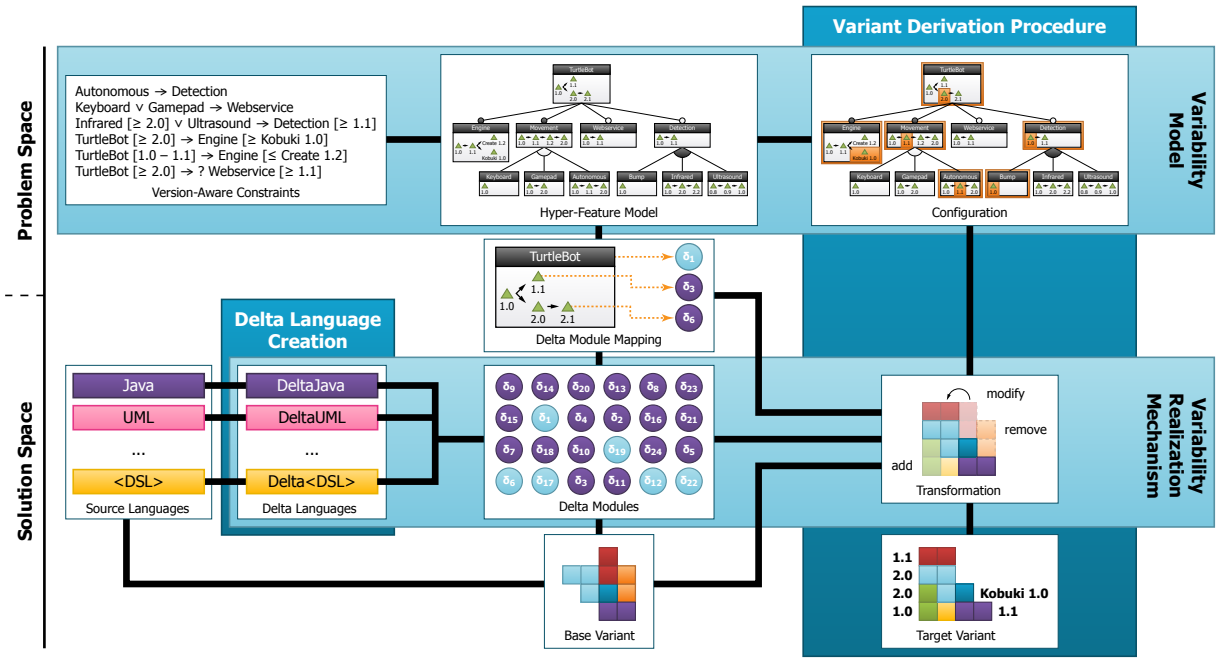

Fig. 6 The Hyper-Feature Model (HFM) and version-aware constraints constitute the variability model on conceptual level. The creation and application of delta languages form the variability realization mechanism on realization level. The variant derivation procedure uses the delta module mapping to translate a conceptual configuration to a set of relevant delta modules. The delta modules are applied in a suitable sequence to transform the base variant of the software family to the intended target variant, which encompasses aspects of variability in space and time according to the selected features and feature versions.

5 Variant Derivation Procedure

Configuration knowledge regarding variability in space and in time is captured conceptually in HFMs and manifested in configuration and evolution delta modules, which transform realization artifacts. A variant of the software family is a software system that encompasses the functionality specified by the selected features at the revisions specified by the selected versions. Fig. 6 illustrates the interconnection of the different artifacts defined in the thesis and the following sections describe the process of deriving a variant with variability in space and time.

5.1 Automatic Version Selection

The first step of variant derivation is the selection of a configuration. This can be performed using the graphical representation of HFMs by manually selecting features and suitable versions. However, the manual selection of versions may be tedious. To remedy this problem, the thesis defines a procedure to automatically select a set of suitable versions for a valid preselection of features. For this purpose, the HFM, the version-aware constraints and the preselection of features are translated into a Constraint Satisfaction Problem (CSP). A CSP solver determines suitable version constellations as possible solutions. The procedure is guided by an objective function, which rates intermediate solutions to only maintain an optimal solution. The thesis defines three criteria that may be combined to form a suitable objective function:

1. **Novelty**: More recent versions towards the end of a branch are assumed to be preferable over less recent ones as they are more current.
2. **Importance**: Features closer to the root of the tree spanned by the HFM are assumed to represent more coarse-grain functionality and, thus, to have a larger effect on the overall system than features further down in the tree.
3. **Inverse Importance**: Features further away from the root of the tree spanned by the HFM are assumed to represent the actual implementation of functionality and, thus, to have a larger effect on the overall system than features further up in the tree that may just serve the purpose of conceptual containers.

Each of these criteria is scored with a value between 0 (bad) and 1 (good) by analyzing the structure of the HFM. Obviously, these factors may be contradictory in the sense that improving the value for one automatically leads to deterioration of the other (e.g., with Importance and Inverse Importance). However, each of these values

may be valid in its own right depending on the nature of the concrete software family. Furthermore, additional criteria may be defined that can be used as part of the objective function for automatic version selection.

5.2 Delta Module Mapping

To derive a variant for a conceptual configuration, the configuration needs to be resolved to the set of relevant delta modules. For this purpose, an explicit mapping from (combinations of) features and feature versions of the HFM to (sets of) configuration and evolution delta modules is defined as visualized in Fig. 7.

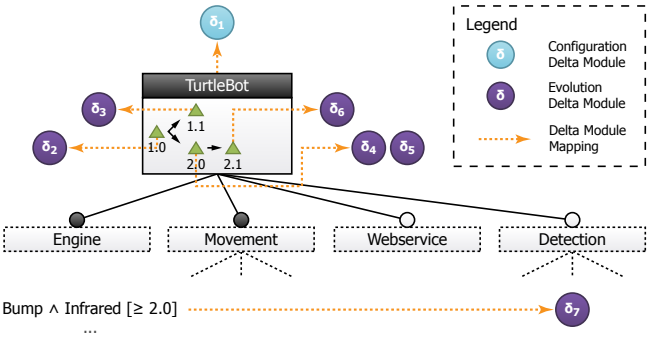

Fig. 7 Features are mapped to configuration delta modules and versions are mapped to evolution delta modules but more complex mappings are possible.

In the general case, each feature is mapped to a configuration delta module and each feature version is mapped to an evolution delta module. However, more complex mappings may be specified if more sophisticated logical expressions are required as condition or multiple delta modules are used as target. The language to specify conditions of mappings uses the same constructs as the version-aware constraint language. The delta modules of a mapping are relevant for variant derivation if the respective condition is satisfied by the conceptual configuration.

5.3 Application Order and Variant Derivation

Using the mapping model, it is possible to resolve a conceptual configuration to a set of required delta modules. As feature versions of HFMs are incremental, the predecessors of explicitly selected versions are implicitly included when evaluating conditions of the mapping. To determine all relevant versions for a particular configuration, the HFM is pruned of all irrelevant versions using two automated steps:

1. Prune all branches of the development line that do not contain the selected version.
2. Prune all versions superseding the selected version.

This procedure is illustrated in Fig. 8 where the irrelevant versions of feature `TurtleBot` are pruned for a selection of version 2.0. The remaining versions are used as basis for evaluating the conditions of the delta module mapping.

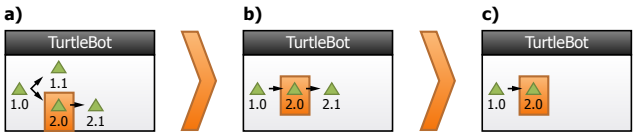

Fig. 8 Irrelevant versions are pruned from the HFM as preparation for determining an application sequence: a) initial version constellation, b) pruned irrelevant branches, c) pruned superseding versions.

It is further necessary to establish an application sequence of the relevant delta modules to ensure deterministic variant derivation. With HFMs, a large part of the application order constraints is implicitly imposed by the structure of the HFM:

1. Delta modules of an initial version require the delta modules of their defining feature.
2. Delta modules of a version require the delta modules of their predecessor version.

These two rules are employed to introduce implicit application order constraints upon the delta modules associated with individual features and versions as demonstrated on the HFM in Fig. 9.

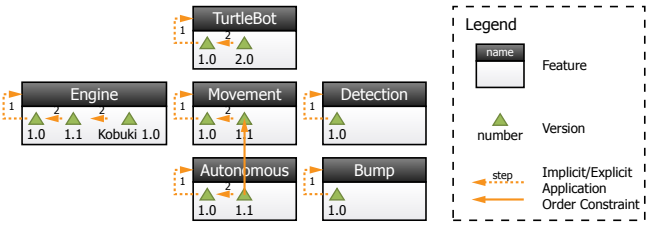

Fig. 9 In addition to explicit application order constraints, implicit application order constraints are established to represent 1) the versions' dependence on the defining feature and 2) the incremental nature of versions.

Additionally, explicitly specified application order constraints are collected. Using these order constraints, a topological sorting is performed to determine a partial order that respects the interdependencies of all relevant delta modules. From this partial order one concrete sequence is chosen. The relevant delta modules and their

delta operations are applied to a copy of the realization artifacts of the software family's base variant. The target variant resulting from this procedure contains functionality of the selected features at the revisions specified by the selected feature versions and, thus, encompasses aspects of both variability in space and time.

6 Evaluation

To examine the suitability of the concepts presented in the thesis, an evaluation was performed. For this purpose, all concepts of the thesis were realized using model-based development to create the tool suite *DeltaEcore*[4]. Within the evaluation, three case studies were inspected:

1. **Configurable TurtleBot driver**: The TurtleBot is a small domestic service robot. This case study presents the robot's driver software, which can be custom tailored to different configurations of the robot to account for limited resources, such as CPU.
2. **Metamodel Family for Role Modeling**: Role-based modeling captures both context-dependent and collaborative behavior of objects within various notations. This case study presents a family of metamodels that can be utilized to derive different concrete notations as defined in the literature.
3. **Family of Feature Modeling Notations**: Many different, yet similar, feature modeling notations with accompanying constraint languages exist. This case study presents a software family consisting of metamodels and a syntax for textual languages to derive a multitude of feature modeling notations and constraint languages of different expressive power.

	Case Study 1	Case Study 2	Case Study 3
Monitored Period	1.5 years	1.5 months	1.0 months
Features	11	48	57
Versions	29	106	64
Constraints	6	5	10
Delta Languages	7	1	2
Delta Modules	46	52	65

Table 1 The inspected case studies utilize various combinations of nine different languages, devise delta dialects for them and specify HFMs with features and feature versions as well as configuration and evolution delta modules to manage variability in space and time.

The software systems of the case studies utilize different combinations of a total of nine individual source languages (see Table 1). For each of the source languages, a delta dialect was devised using the provided language creation infrastructure. Changes associated with variability in space and time could be captured as features and feature versions in HFMs. Utilizing the created delta languages, configuration and evolution delta modules were devised to manifest the changes associated with features and feature versions in realization artifacts. In order to derive variants, configurations were created manually and semi-automatically using the automatic version selection procedure. The resulting variants were inspected for validity through manual inspection as well as automated tests.

On a conceptual level, it was possible to specify all changes as features and feature versions along with version-aware constraints. On a realization level, changes associated with variability in space and time could be manifested within configuration and evolution delta modules. Created variants contained the changes according to the selected features and feature versions and exposed no defects. Emerging problems in the process could be circumvented using the means provided by DeltaEcore. Hence, apart from minor caveats, the results of the evaluation support the claim that the concepts of the thesis constitute a suitable solution for an integrated management of variability in space and time in software families.

7 Related Work

Work related to the thesis addresses subsets of variability in space and time on the conceptual or realization level. However, no integrated approach is presented. Fig. 10 illustrates the addressed areas of related work and the following sections briefly discuss each approach.

Feature models [15,6] may not be used as adequate substitute for HFMs when representing variability in time as their workaround solutions to representing feature versions do not capture the intent of variability in time and development lines of versions cannot be represented.

Mitschke and Eichberg [21] introduce a versioning scheme for feature models of SPLs they call feature-driven versioning but their approach allows only one version per feature at a time and, thus, does not support configuration of versions.

Ducasse et al. [8] model software evolution by treating history as first class entity. Their model is used primarily for analyses determining certain evolution patterns but cannot be employed for creating actual variants of a software family.

Zschaler et al. [31,25] introduce VML* as a family of variability modeling languages created by bootstrap-

[4] `http://deltaecore.org`

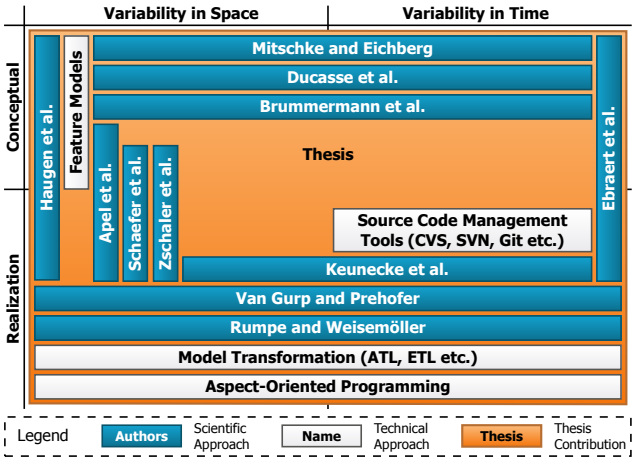

Fig. 10 Related work only addresses subsets of variability in space and time on the conceptual or realization level where the thesis presents an integrated approach.

ping SPL techniques. These languages handle variability in space but not variability in time.

Haugen et al. [14] introduce the Common Variability Language (CVL) with the intent of adding standardized variability in space to arbitrary base models. However, at its current level of development, CVL does not aim at being applicable for variability in time.

Apel et al. [2] define FeatureHouse to create languages for capturing changes related to variability. However, they cannot model arbitrary evolutionary changes due to the compositional nature of the operations.

Schaefer et al. [27] introduce delta modeling to apply changes associated with variability in space in realization assets through transformation. Its concepts are used as basis for the work of the thesis.

Ebraert et al. [10,9] define Change-Oriented Programming (ChOP), which uses change objects to represent evolutionary modifications on realization assets in order to capture variability in time. Even though attempts exist for extracting information regarding variability in space [9], the created feature models do not represent the intentions of configurable features.

Keunecke et al. [16] define feature packs to handle variability of SECOs as components accompanied by variability information. Even though this approach may, in part, be used to address concerns of variability in time, it imposes a component-based architecture on the solution space and is, thus, not applicable in general.

Brummermann et al. [4] introduce a strictly formal definition of distributed variability evolution in SECOs making the variability model itself subject to variability. However, the concrete handling of variant derivation is outside the scope of their approach.

Rumpe and Weisemöller [24] introduce domain-specific model transformation using custom model-modification operations for individual source languages provided as metamodel. While these operations are general enough to perform all modifications on realization artifacts, there is no distinction between the intentions of configuration and evolution, which bears the risk of unintentionally damaging a variant.

Van Gurp and Prehofer [30] augment artifacts under version control with properties representing information from the variability model, which reduces conceptual information to the realization level. Hence, versions for features are not explicit on a conceptual level and logical constraints on versions cannot be specified.

On a technical level, **source code management tools** such as CVS[5], SVN[6] or Git[7] may address handling of versions. However, these tools do not offer information on a conceptual level.

General purpose **model transformation** languages, such as ATL[8] or ETL[9], may be used to manifest changes for both variability in space and time within realization assets. However, the lack of a distinction of both types of changes and their intentions leads to a risk of accidentally harming consistency of an SPL or a SECO as part of modeling variability in space due to too potent modification operations.

Aspect-Oriented Programming (AOP) [17] can be employed to handle configuration and evolution of software families [1,12,11]. However, the compositional nature of the approach makes removal of realization assets complicated so that arbitrary evolutionary changes cannot be performed.

8 Conclusion

This thesis presented an integrated approach for managing variability in space and time in software families. It introduced Hyper-Feature Models (HFMs) with explicit elements for feature versions and defined a version-aware constraint language to formulate cross-tree constraints on versions and version ranges. Furthermore, it distinguished configuration and evolution delta modules due to their different characteristics and introduced a language creation infrastructure to devise suitable delta languages for arbitrary source languages. A mapping from (logical expressions over) features and feature versions of HFMs to sets of delta modules allows resolution

[5] http://cvs.nongnu.org
[6] http://subversion.apache.org
[7] http://git-scm.com
[8] http://eclipse.org/atl
[9] http://eclipse.org/epsilon/doc/etl

of conceptual configurations to delta modules, which are relevant for creating the associated variant. For these delta modules, a suitable application order is determined by respecting the explicitly specified order constraints as well as those implicitly imposed by the structure of the HFM. Applying delta modules in the determined sequence transforms a base variant into a target variant for the selected configuration that contains aspects of variability in space and time.

References

1. V. Alves, P. M. Jr, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming. In *Transactions on Aspect-Oriented Software Development IV*, pages 117–142. Springer, 2007.
2. S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013.
3. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Pearson, 2000.
4. H. Brummermann, M. Keunecke, and K. Schmid. Formalizing Distributed Evolution of Variability in Information System Ecosystems. In *Proceedings of the 6th Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, 2012.
5. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, 2010.
6. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-Based Feature Models and their Specialization. *Software Process: Improvement and Practice*, 2005.
7. F. Damiani and I. Schaefer. Dynamic Delta-Oriented Programming. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, 2011.
8. S. Ducasse, T. Gîrba, and J. Favre. Modeling Software Evolution by Treating History as a First Class Entity. *Electronic Notes in Theoretical Computer Science*, 2005.
9. P. Ebraert, A. Classen, P. Heymans, and T. D'Hondt. Feature Diagrams for Change-Oriented Programming. In *ICFI*, 2009.
10. P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, and T. D'Hondt. Change-Oriented Software Engineering. In *Proceedings of the International Conference on Dynamic Languages*. ACM, 2007.
11. E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Dantas, et al. Evolving Software Product Lines with Aspects. In *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE'08.*, pages 261–270. IEEE, 2008.
12. I. Groher and M. Voelter. Aspect-Oriented Model-Driven Software Product Line Engineering. In *Transactions on Aspect-Oriented Software Development VI*, pages 111–152. Springer, 2009.
13. A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS'13. ACM, 2013.
14. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *12th International Software Product Line Conference (SPLC'08)*. IEEE, 2008.
15. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, DTIC Document, 1990.
16. M. Keunecke, H. Brummermann, and K. Schmid. The Feature Pack Approach: Systematically Managing Implementations in Software Ecosystems. In *Proceedings of the 8th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, 2014.
17. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-Oriented Programming*. Springer, 1997.
18. J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 63–74, New York, NY, USA, 2014. ACM.
19. M. Kowal, C. Legat, D. Lorefice, C. Prehofer, I. Schaefer, and B. Vogel-Heuser. Delta Modeling for Variant-Rich and Evolving Manufacturing Systems. In *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, MoSEMInA, 2014.
20. B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6):466–471, 1978.
21. R. Mitschke and M. Eichberg. Supporting the Evolution of Software Product Lines. In *ECMDA Traceability Workshop*, ECMA-TW, 2008.
22. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques*. Springer Berlin/Heidelberg, 2005.
23. R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.
24. B. Rumpe and I. Weisemöller. A Domain Specific Transformation Language. In *Proceedings of the Workshop on Models and Evolution (ME)*, 2011.
25. P. Sánchez, N. Loughran, L. Fuentes, and A. Garcia. Engineering Languages for Specifying Product-Derivation Processes in Software Product Lines. In *Software Language Engineering*, pages 188–207. Springer, 2009.
26. I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, VaMoS'10, 2010.
27. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*, pages 77–91. Springer, 2010.
28. C. Seidl, I. Schaefer, and U. Aßmann. Variability-Aware Safety Analysis using Delta Component Fault Diagrams. In *Proceedings of the 4th International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, FMSPLE'13, 2013.
29. I. Sommerville. *Software Engineering*. Pearson, 2001.
30. J. van Gurp and C. Prehofer. Version Management Tools as a Basis for Integrating Product Derivation and Software Product Families. In *Proceedings of the Workshop on Variability Management-Working with Variability Mechanisms at SPLC*, 2006.
31. S. Zschaler, P. Sánchez, J. Santos, M. Alférez, A. Rashid, L. Fuentes, A. Moreira, J. Araújo, and U. Kulesza. VML*–A Family of Languages for Variability Management in Software Product Lines. In *Software Language Engineering*. Springer, 2010.