Chapter 7
Model-Based Round-Trip Engineering and Testing of Evolving Software Product Lines

Malte Lochau, Dennis Reuling, Johannes Bürdek, Timo Kehrer, Sascha Lity, Andy Schürr, and Udo Kelter

Modern software systems tend to be more and more long living and, therefore, have to undergo continuous evolution to cope with new, and often initially unforeseen, user requirements, application contexts, and execution platforms. In practice, the necessary changes applied to respective design-, implementation-, and quality-assurance artefacts are often performed in an ad hoc, and mostly manually conducted, manner, thus lacking proper documentation, consistency checks among related artefacts, and systematic quality-assurance strategies.

These issues become even more challenging in case of variant-rich software systems such as software product lines, where even small changes may (intentionally or

M. Lochau (✉) · J. Bürdek · A. Schürr
Technische Universität Darmstadt, Fachbereich Elektrotechnik und Informationstechnik, Fachgebiet Echtzeitsysteme, Darmstadt, Germany
e-mail: malte.lochau@es.tu-darmstadt.de; johannes.buerdek@es.tu-darmstadt.de; andy.schuerr@es.tu-darmstadt.de

D. Reuling
Praktische Informatik/Softwaretechnik, Fachbereich Elektrotechnik und Informatik, Universität - GH - Siegen, Siegen, Germany
e-mail: dreuling@informatik.uni-siegen.de

T. Kehrer
Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany
e-mail: timo.kehrer@informatik.hu-berlin.de

S. Lity
Institut für Softwaretechnik und Fahrzeuginformatik, Technische Universität Braunschweig, Informatikzentrum, Braunschweig, Germany
e-mail: lity@isf.cs.tu-bs.de

U. Kelter
Praktische Informatik/Softwaretechnik, Fachbereich Elektrotechnik und Informatik, Universität - GH - Siegen, Siegen, Germany
e-mail: kelter@informatik.uni-siegen.de

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_7

erroneously) affect a high number of similar product variants simultaneously. Again, the idealistic assumption that a software product line is designed, implemented, and assured in its entirety from scratch prior to the initial delivery any individual product variant to costumers is often unrealistic in practice. In particular, three (potentially concurrently) evolving sets of related product-line artefacts have to be taken into account:

1. A *product-line architecture* typically consists of a configuration model, configurable product-line implementation source code, as well as further design- and quality-assurance artefacts from which respective variants are automatically derivable for a given product configuration.
2. A *product family* consists of materialised software variants corresponding to valid product configurations of the product line as delivered to the customers.
3. A set of *product-specific quality-assurance artefacts* (e.g. test cases) that permit sufficient assurance of every software variants of the product line prior to their delivery and initial execution by the customer.

As a consequence, during product-line *evolution* and *co-evolution* scenarios, developers are faced with multiple diverse yet highly interrelated notions of artefact-consistency preservation, namely consistency between (1) product-line architecture artefacts and (2) respective software variants of the product family, as well as consistency between (3) configuration-specific quality-assurance artefacts and (2) corresponding software variants.

In this chapter, we describe a model-based framework for systematic and (semi-)automatic round-trip engineering of continuously evolving software product lines incorporating all possible evolution and co-evolution scenarios of product-line engineering and quality-assurance artefacts. To this end, we lift the corresponding *forward-* and *re-engineering* scenarios known from classical round-trip engineering to product-line engineering, respectively. In particular, we consider a product-line architecture to consist of a feature diagram serving as a configuration model, a STATECHART model superimposing all product-variant behaviours into one behavioural product-line specification, and a preprocessor-based C-code product-line implementation comprising all software-variant implementations. As quality-assurance methodology, we consider model-based testing, where test suites are automatically generated for product-line implementations with respect to a given set of test goals on the corresponding product-line STATECHART test model, to be covered on all derivable software variants. Our methodology combines two key techniques from model-based software engineering, namely:

- *Model differencing* and *model merging* for automatically comparing and integrating software variants and versions in a systematic way into one unified yet evolving product-line representation, and
- *Knowledge-carrying software* for integrating information about variant- and version-specific software artefacts into engineering and quality-assurance processes at different levels of abstraction

This combination ensures consistency of interrelated engineering- and quality-assurance artefacts throughout the entire life cycle of evolving product lines. In addition, the approach facilitates the application of efficient family-based analysis strategies, initially developed for software variants already organised in product lines, to both variant- and version-rich software systems, as well as arbitrary combinations thereof.

To summarise, the contribution of this chapter consists of an integrated approach that combines different recent techniques and tools from model-based software engineering and software product-line engineering into one novel conceptual framework for product-line round-trip engineering. The methodology is illustrated by a running example by means of an extract from the extended Pick and Place Unit (xPPU) study, and we further describe available tool support for the different techniques.

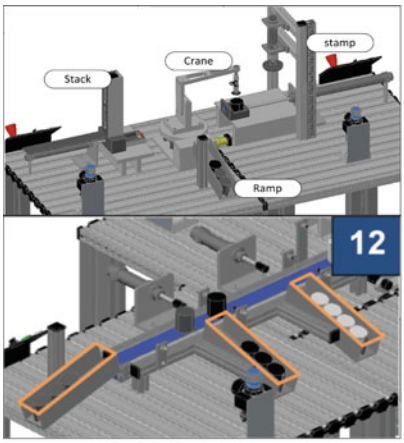
This chapter is organised as follows. In Sect. 7.1, we first describe the necessary background on product-line engineering and model-based testing and introduce a running example by means of an extract from the xPPU case study. Based on these foundations, we summarise the challenges in round-trip engineering and model-based testing for quality assurance of evolving software product lines, as addressed in the remainder of the chapter. The main part of this chapter is separated into two consecutive sub-parts: in Sect. 7.2, we first describe evolution scenarios of the different engineering and quality-assurance artefacts separately and, in Sect. 7.3, we then explain co-evolution scenarios to ensure consistency among concurrently evolved yet interrelated artefacts. Section 7.4 concludes and gives a sketch of a road map for future research. Finally, Sect. 7.5 summarises recent publications describing in detail the different approaches summarised in this chapter.

7.1 Foundations

In this section, we first describe the necessary background and basic notions from the research fields of model-based software engineering and testing, especially in the context of software product lines, as used throughout this chapter. Based on these concepts, we describe the major challenges in handling evolution and co-evolution scenarios in product-line engineering and model-based testing, in order to facilitate a comprehensive methodology to support model-based round-trip engineering and quality assurance of evolving software product lines.

7.1.1 Model-Based Software Development and Testing

As our running example, we consider an excerpt from the extended Pick and Place Unit (xPPU) case study [Vog+14b], which is used in the following to illustrate the proposed methodology. For a detailed description of the xPPU case study, we refer the interested reader to Sect. 4.3.



(a)

	v1	v2	v3
1	Initialize	Initialize	Initialize
2	Add Error Handling	Add Error Handling	-
3	-	-	Add Error Handling
4	-	Remove Error Handling	-
5	-	-	Delete Variant

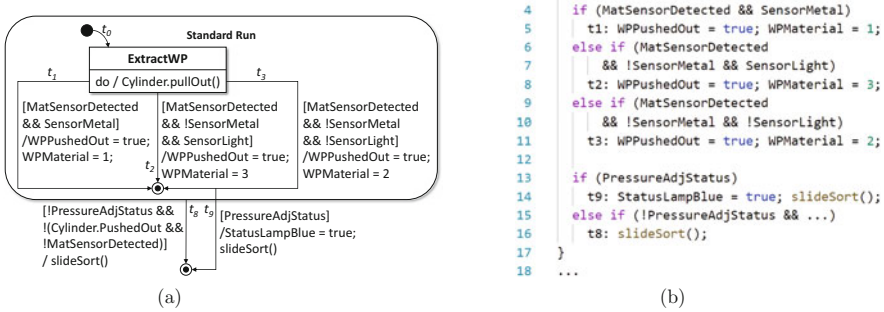
(b)

Fig. 7.1 (**a**) xPPU evolution scenario. (**b**) Overview of xPPU evolution steps

Extended Pick and Place Unit (xPPU) The xPPU is a bench-scale demonstrator for software systems in the automation-engineering domain. As depicted in Fig. 7.1a, the xPPU is a configurable system consisting of several different hardware components for handling and transporting Workpieces (WP) with cylindrical shapes (e.g. bottles). In this way, the xPPU is adaptable to different application scenarios. In particular, the xPPU is able to handle three types of WP: *light plastic*, *dark plastic*, and *metal*. To this end, an xPPU comprises a *Stack* working as WP input storage, a *Ramp* working as a WP output storage, a *Stamp* for labelling WP, and a *Crane* for transporting WP between working positions.

The PLC-based control software of the xPPU has been developed in a model-based way, by employing a combination of structural and behavioural modelling languages as defined by the EN 61131-3 standard for automation-engineering software [Gro11]. Model-based development of automation-control software helps to cope with inherent complexity and mission criticality, as apparent in this and similar application domains, by facilitating automated generation of high-quality and platform-specific implementation code, as well as model-based quality-assurance techniques such as model-based testing.

Model-Based Testing Model-based testing is a widely used black-box testing technique that abstracts from internal implementation details of software components or -systems under test [UL07]. To this end, a *test model* serves as a behavioural specification of the expected behaviour of the (potentially inaccessible) implementation code to be tested. Behavioural conformance of an implementation to a given test model is investigated by experimental execution scenarios (i.e. *test cases*). Hence, *test models* are utilised in two ways during model-based testing:




```
1  ...
2  while (!WPStockEmpty) {
3    Cylinder.pullOut();
4    if (MatSensorDetected && SensorMetal)
5      t1: WPPushedOut = true; WPMaterial = 1;
6    else if (MatSensorDetected
7        && !SensorMetal && SensorLight)
8      t2: WPPushedOut = true; WPMaterial = 3;
9    else if (MatSensorDetected
10       && !SensorMetal && !SensorLight)
11     t3: WPPushedOut = true; WPMaterial = 2;
12
13   if (PressureAdjStatus)
14     t9: StatusLampBlue = true; slideSort();
15   else if (!PressureAdjStatus && ...)
16     t8: slideSort();
17 }
18 ...
```

Fig. 7.2 Extract from a xPPU variant. (**a**) Test model. (**b**) Code

- The test model is used as input for testing tools for automatically *generating* test cases, *executing* those test cases on the system under test, and *evaluating* test-execution results with respect to the expected behaviour (*test oracle*) as stated by the test model.
- The test model is used to measure *adequacy* of an (either already existing or proactively generated) set of test cases (i.e. a *test suite*). For instance, a *coverage criterion* may be applied to identify a set of *test goals* in the test model, each to be satisfied by at least one test case of the test suite.

Figure 7.2 shows an extract from the test-model specification of the xPPU in terms of a STATECHART model [UL07]. STATECHARTS (and respective dialects) offer a widely used visual modelling language that constitutes a particularly well-established specification formalism for concisely capturing functional specifications of reactive control-software systems at system and component levels. STATECHARTS are also widely applicable as a basis for automated generation of implementation code, as well as for model-based test-case generation and test-coverage measurement [UL07, Rös+14, Loc+14].

The xPPU behaviour, as abstractly specified in the STATECHART model in Fig. 7.2, constitutes handling of three different types of WP: *light plastic*, *dark plastic*, and *metal*. Each of those types of WP are transported from the *Stack* via the *Crane* to the *Stamp*. *Light* WP are stamped using *adjustable pressure*, whereas *dark* WP and *metal* WP are stamped using *standard pressure*. To this end, variable *PressureAdjStatus* determines whether adjustable pressure or standard pressure is used based on the material of the incoming WP. Finally, all WPs are transported to the *Slide* and sorted according to their specific type. The behaviour specified in the test model in Fig. 7.2a corresponds to one particular *implementation variant* of the xPPU, as shown in the (simplified) code-listing excerpt in Fig. 7.2b. Whenever a new WP arrives in the xPPU (see Line 2), the *Cylinder* pulls it from the *Stack* (see Line 3). Lines 4–11 implement the control logic for identifying and handling the three different types of WP, as described above.

When using STATECHARTS as test models, test cases correspond to valid and complete transition paths in the state-transition graph (i.e. paths corresponding to valid executions from the initial state to a final state). A test-case execution thus defines a sequence of input stimuli to be injected into the system under test, together with a corresponding sequence of observable output behaviours expected from the system under test for those inputs as given by the transition labels in the test-model specification. Similar to code-coverage criteria, coverage criteria for STATECHART models aim to investigate different possible control flows (e.g. state and transition coverage), as well as data-flow aspects (e.g. def-use coverage) of the implementation under test [UL07].

For example, applying *transition coverage* to the xPPU test model in Fig. 7.2 ensures that a test suite contains at least one test case for investigating the correct handling of each type of WP. The code parts corresponding to the three test goals $t1$, $t2$, $t3$ correspond to the three transitions in the test model (see Fig. 7.2a) and are marked with respective code labels (see Fig. 7.2b). For instance, a test-case execution examining the handling of *light plastic* WP with *adjustable pressure* requires as expected output the corresponding status lamp to be switched on (test goal $t9$ in Line 14). After that, all types of WP are transported to the slide, where they are finally sorted according to their specific type (test goals $t9$ and $t8$ in Line 14 and 16, respectively).

To summarise, a *test suite* achieving complete transition coverage on the xPPU test model in Fig. 7.2a requires at least three test cases, for instance:

- Test case $tc_1 := (t_0, t_1, t_8)$ for handling *metal* WP
- Test case $tc_2 := (t_0, t_2, t_9)$ for handling *light plastic* WP using *adjustable pressure*, and
- Test case $tc_3 := (t_0, t_3, t_8)$ for handling *dark plastic* WP

Product Families Besides the particular xPPU variant described so far, the modular architecture of the xPPU supports many further variants in order to adapt to different environments, platforms, and customer needs. Such a collection of similar yet well-distinguished variants of the same *core product* is frequently called a *product family* [Ape+13]. For presentation purposes, we limit our considerations in the following to two further variants from the *xPPU product family*, referred to as *v2* and *v3*, and the previously described variant is denoted as *v1*, respectively. In contrast to variant *v1*, variant *v2* has reduced functionality; namely, it cannot handle *light plastic* WP differently and always uses *standard pressure* for stamping. Figure 7.3 shows the corresponding extract from the test model and the respective implementation code of variant *v2*. Here, the handling of *metal* WP is equal to that of variant *v1*, whereas the handling of *light plastic* and *dark plastic* are the same in *v2*, contrary to different behaviours for each *plastic* WP in case of variant *v1*. Hence, a test suite achieving complete transition coverage on the test model of variant *v2* requires at least two test cases, for instance

- test case $tc_1 := (t_0, t_1, t_8)$ may be (re-)used from the test suite of variant *v1*, whereas

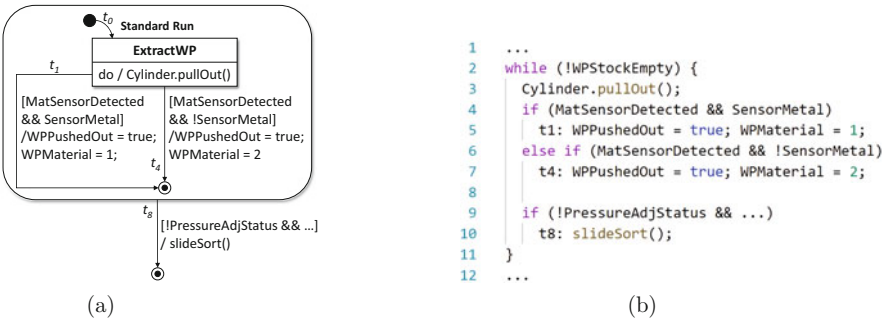


Fig. 7.3 A second xPPU variant. (**a**) Test model. (**b**) Code

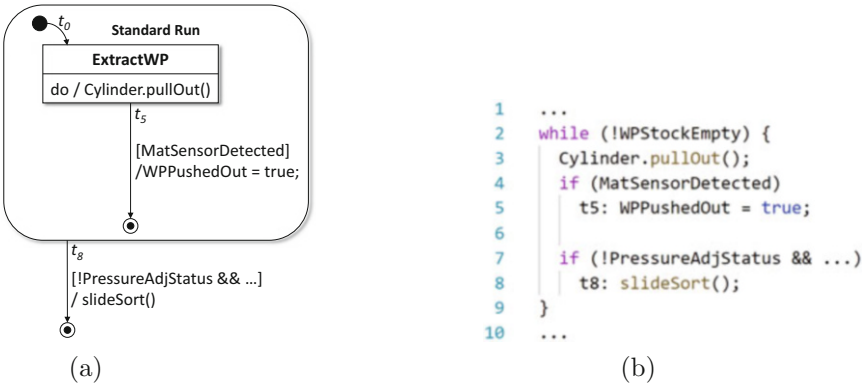


Fig. 7.4 A third xPPU variant. (**a**) Test model. (**b**) Code

- test case $tc_4 := (t_0, t_4, t_8)$ is a new test case, additionally required to examine equal handling of *light plastic* and *dark plastic* WPs in variant *v2*.

In contrast to the reusable test case t_1, the other test cases t_2, t_3 derived for testing variant $v1$ are not applicable for testing variant *v2*.

Finally, variant v_3 constitutes a very basic xPPU, which is only able to handle *metal* WP and which has no *Stamp* (see Fig. 7.4). As a consequence, for testing variant $v3$ (again, aiming at transition coverage of the respective test model of variant *v3*), only one single test case, for instance

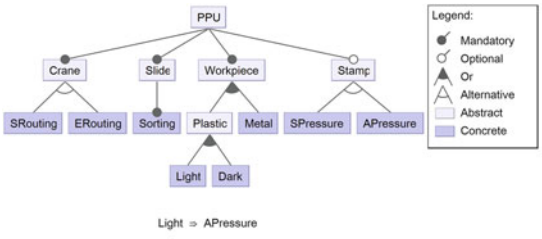
- test case $tc_5 := (t_0, t_5, t_8)$

is required, which differs from all the previously derived test cases due to the essential behavioural differences of variant *v3*, as compared with variant *v1* and *v2*. Next, we describe how principles from product-line engineering can help to systematically exploit commonality among the members of a product family during both software development and quality assurance (e.g. for reasoning about test-case reuse among variants).

7.1.2 Model-Based Product-Line Engineering and Testing

Software product line engineering (SPLE) is an emerging methodology that has been successfully applied in various industrial application domains [Wei08]. SPLE offers a practicable possibility to handle the increasing variability during engineering and quality assurance of automation-control software, as described for the xPPU example. To this end, SPLE aims at systematically exploiting knowledge about commonality and variability among all kinds of engineering artefacts (e.g. design- and test models, implementation code, and test cases) in a family of similar products [PBL05a, CN01]. An explicit specification of common and variable parts among the different variants is based on their supported *features*, denoting configuration parameters (i.e. user-visible characteristics of products) in the *problem space* of a product family. For automated derivation of product variants complying to a given configuration, features are further related to software building blocks by means of reusable engineering artefacts in the *solution space*, being composable into respective implementation variants. In the following, we first describe the idealistic view on product-line engineering based on the assumption that the whole product line is developed from scratch before finally being delivered to the customer.

Problem Space For the problem-space specification, SPLE usually employs *feature models* to describe the set of available features, together with constraints among those features to be satisfied by a feature selection to constitute a *valid product configuration*. Figure 7.5a shows the feature model for the xPPU product line using the visual Feature Oriented Domain Analysis (FODA) notation (frequently called *feature diagrams*) [Kan+90a]. A feature model organises the set of supported features as nodes in a tree-like hierarchy, inducing dependencies of child features to its parent features (i.e. the selection of a feature requires the selection of its parent feature in a valid configuration). Singleton child features are either *mandatory* (i.e. they must be selected whenever their parent features are selected in a valid configuration) or *optional*. For instance, a valid xPPU configuration *must* contain a *Crane* device and *at least one Slide* and *must* handle *at least one* type of *Work Piece*, whereas the *Stamp* is *optional*. Besides singleton child features, mutually



(a)

	Metal	Dark	Light	SPressure	APressure	Sorting	SRouting
V1	x	x	x		x	x	x
V2	x	x		x		x	x
V3	x			x		x	x
V4	x				x	x	x
V5		x			x	x	x
V6			x		x	x	x
V7	x	x			x	x	x
V8	x		x		x	x	x
V9		x	x		x	x	x
V10		x		x		x	x

(b)

Fig. 7.5 Extract from the xPPU feature model and valid configurations. (**a**) xPPU feature model. (**b**) xPPU variants

dependent sibling child features may be assembled into *feature groups*, being either *or groups* (i.e. *at least one* of its features must be selected if the parent feature is selected), or *alternative groups* (i.e. *exactly one* feature must be selected). For instance, a *Crane* either uses *Standard Routing*,or *Extended Routing*, whereas the set of types of supported *Plastic* WP may include *Dark*, *Light*, as well as both in combination. Finally, further dependencies between hierarchically unrelated features can be expressed using *cross-tree constraints* (e.g. *Work Pieces* made of *Light Plastic* require a *Stamp* with *Adaptive Pressure*). The set of all valid configurations according to the xPPU feature model is given in Fig. 7.5b. Please note that—due to space limitations—we omitted the second half of configurations, which only differs from the given ones by having *ERouting* selected instead of *SRouting*. Further note that the first three configurations correspond to the xPPU variants $v1$, $v2$, and $v3$, as described above.

In the next step of SPLE, a *mapping* of configuration-specific solution-space artefacts onto corresponding feature selections is defined, in order to relate configurations to respective parts in configurable test models and implementation code of the product line.

Solution Space Features not only denote configuration parameters in the problem space but also refer to *variation points* within engineering artefacts in the solution space, potentially at all levels of abstraction [Ape+13]. Here, we use an annotation-based approach for a product-line representation of a product family, by integrating variability information into solution-space artefacts (i.e. test models, implementation code, and test artefacts).

Presence Conditions for Variant-Knowledge At the level of design- and test models like STATECHARTS, variant-specific model elements (here: transitions) are equipped with annotations over propositional feature expressions, representing *presence conditions* for well-defined variation points in the solution space. Those *model templates* therefore virtually include (or superimpose) any possible model variant of the product line into one model, constituting a so-called *150% model*. Hence, a configuration-specific *model variant* (i.e. a 100% model) can be obtained from a 150% model by projecting only those model elements whose presence conditions are satisfied by the respective feature selection of the configuration [CE00]. Figure 7.6a depicts the 150% test model for the xPPU product line, where the respective test-model variants for the configurations $v1$, $v2$, and $v3$ correspond to the model variants, as described above.

A similar principle is frequently used in practice for integrating variation points into source-code artefacts of product-line implementations: conditional-compilation directives such as `#if` macro, as provided by the C preprocessor, allow for marking variable code parts (variation points), again, by using propositional formulae over (Boolean) feature variables as presence conditions [Käas+11]. Figure 7.6b depicts the variable implementation source code of the xPPU example corresponding to the aforementioned 150% test-model extract.

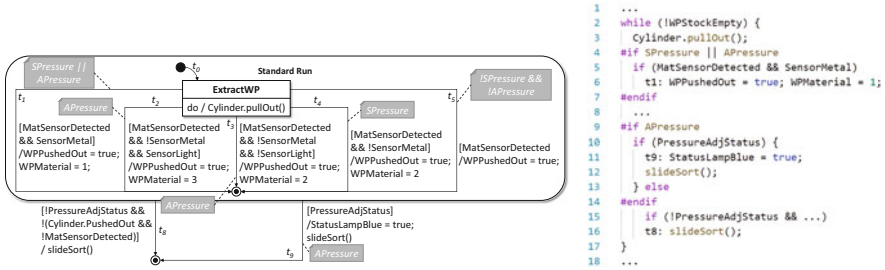


Fig. 7.6 150% xPPU Test model. (**a**) Test model. (**b**) Code

Family-Based Product-Line Testing The additional knowledge in a product-line representation provided by the feature model and corresponding feature mappings onto a 150% test model provides opportunities for improving the efficiency of quality assurance of product families. To this end, *family-based product-line analysis strategies* aim at analysing whole product families at once instead of using a variant-by-variant approach [Thü+14a]. In particular, *family-based test-suite generation* potentially reduces the overall number of test-generator runs and therefore the number of required test cases for covering all members of a product family, as compared to considering every variant one by one, as described above [Bür+15a]. For this, the additional information provided by the presence conditions in 150% test-model specifications supports automated reasoning about (re-)usability of derived test cases among different variants. To do so, the set of presence conditions attached to those transitions located on the path being traversed in the test model by a test case for reaching a particular test goals is conjugated to form a presence condition for that particular test case (i.e. a so-called *Software product line (SPL) test case*). The presence condition of an SPL test case, therefore, characterises exactly the set of configurations for which that test case is applicable. Based on this notion, we call a set of SPL test cases an *SPL test suite*, and an SPL test suite is further called *complete* if for each test goal in the 150% test model (being selected by a given coverage criterion as usual) and for each test-model variant there exists at least one SPL test case covering that test goal and whose presence condition is satisfied by the configuration of that variant (see [Bür+15a, Loc+14] for a precise definition).

As an example, applying family-based SPL test-suite generation to the 150% test model of the xPPU example (see Fig. 7.6) for transition coverage may result in the following *complete SPL test suite*:

- SPL test case $tc_1 := (t_0, t_1, t_8)$; [*SPressure* || *APressure*]
- SPL test case $tc_2 := (t_0, t_2, t_9)$; [*APressure*]
- SPL test case $tc_3 := (t_0, t_3, t_8)$; [*APressure*]
- SPL test case $tc_4 := (t_0, t_4, t_8)$; [*SPressure*], and
- SPL test case $tc_5 := (t_0, t_5, t_8)$ [!*SPressure* && !*APressure*].

Here, the feature expressions given in brackets denote the respective presence conditions (i.e. test case t_1 is applicable to the variants $v1$ and $v2$; test cases t_2, t_3, t_4 are applicable to variant $v1$; and test case t_5 is applicable to variant $v3$). Hence, the resulting test cases exactly correspond to those previously derived by using a variant-by-variant approach but now carry additional information about the respective implementation variants of the xPPU product line to which they are applicable. Hence, test cases being reusable among different product variants are generated only once using a family-based approach, thus reducing the number of (redundant) test-generator calls, as compared to a variant-by-variant approach.

7.1.3 *Product-Line Round-Trip Engineering and Artefact Co-evolution*

In practice, those idealistic 150% product-line representations, on which family-based analysis strategies heavily rely, are usually not—or only partially—available. This is due to the fact that product lines are, in most cases, not developed *pro-actively* from scratch in a *forward-engineering* manner but rather continuously evolve over time and therefore comprise not only variability *in space* (by means of simultaneously existing variants) but also variability *over time* (by means of sequences of subsequent versions). Hence, most product lines are developed *re-actively* (i.e. by starting with an initial minimum product line comprising a small set of core variants, which is then continuously revised throughout their life cycle to adapt to ever-changing needs) or in an *extractive* way (i.e. by *reverse engineering* a product-line representation from an existing product family) or by combining both styles [Ape+13].

For instance, Fig. 7.1a illustrates a possible evolution scenario of the xPPU product line: the core xPPU initially comprises a *Stack* with multiple *Slides* for *Sorting* WP according to their types, as well as a *Crane* and a *Stamp*. Later on, in evolution scenario 12, an alternative *Standard Ramp* without *Sorting* will become available. As a consequence, all product-line artefacts (potentially) affected by those changes have to be adapted to support the new variants, namely the feature model, the 150% design- and test-model specification, the variable implementation code artefacts, the respective model- and implementation variants, and the accompanying model-based SPL testing artefacts.

Figure 7.7a provides an overview of the different model-based product-line engineering and testing artefacts under consideration, together with possible evolution step and resulting co-evolution scenarios (which will be referred to as ①–⑥ in Sect. 7.3) corresponding to respective forward- and re-engineering steps potentially arising during product-line round-trip engineering. To summarise, we consider three different kinds of artefacts and use the following terminology for this different artefacts throughout this article.

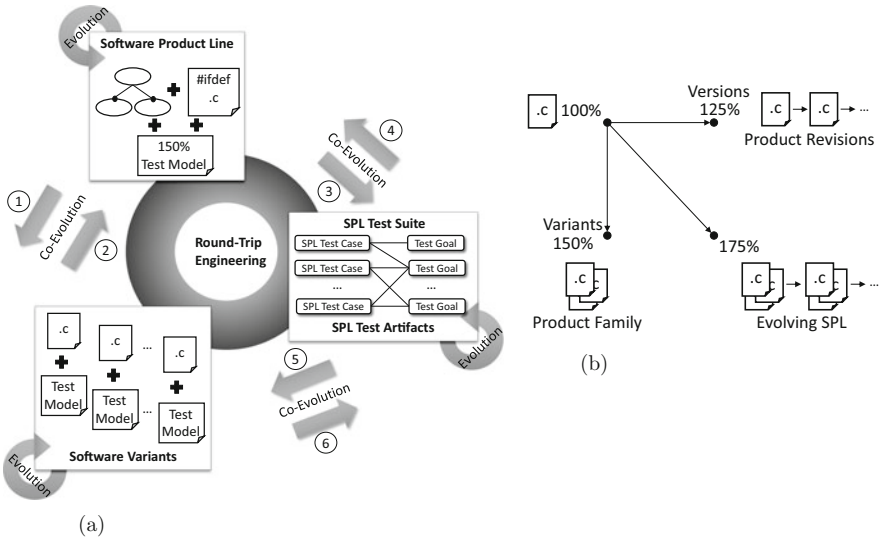


Fig. 7.7 Overview of SPL evolution. (**a**) Product-line round-trip engineering. (**b**) Dimensions of variability

- **Software Product Line Artefacts.** The problem-space artefacts of product lines include the feature model, given as a feature diagram in FODA notation; the solution-space artefacts consists of the 150% implementation, given as C code with preprocessor macros over feature conditions, as well as a 150% test-model specification, given as STATECHART models annotated with feature conditions.
- **Software Variants.** The set of software variants include variant implementations given as (plain) C code, as well as corresponding test-model variants given as (plain) STATECHART models, each of them related to a particular product configuration of the product line.
- **Product-Line Testing Artefacts.** The set of model-based testing artefacts include the set of test goals on the 150% test model, as well as a complete SPL test suite with respect to the set of test goals.

Throughout the life cycle of a product-line, all three kinds of artefacts potentially undergo continuous *evolution* in terms of *changes* imposing *revisions* of artefacts and therefore new *versions* of the entire product line. Due to the complex inter-relations between the different kinds of artefacts, an accompanying *co-evolution* of other artefacts is required in order to ensure artefact consistency in handling (potentially concurrent) evolution steps at any level throughout the entire life cycle of the product line. Concerning model-based engineering and quality assurance of evolving software product lines using model-based testing in particular, the major challenge to be solved can be summarised as follows:

> Every (supported) version of all valid software variants of an evolving product line has to be sufficiently (re-)tested (covered) prior to its (re-)delivery to the customer and/or its initial execution or restart.

As illustrated in Fig. 7.7b, we therefore distinguish three *dimensions* of integrated representations of artefact variability in evolving software product lines based on the initial artefact (i.e. 100% representation), namely:

- All existing *versions* of the same artefact in a 125% presentation
- All existing *variants* of the same artefact in a 150% representation, as well as
- All existing *variants and versions* of the same artefact in a 175% representation

In the following, we describe in detail the different possible scenarios of product-line evolution (Sect. 7.2) and co-evolution (Sect. 7.3), as depicted in Fig. 7.7a.

7.2 Evolution

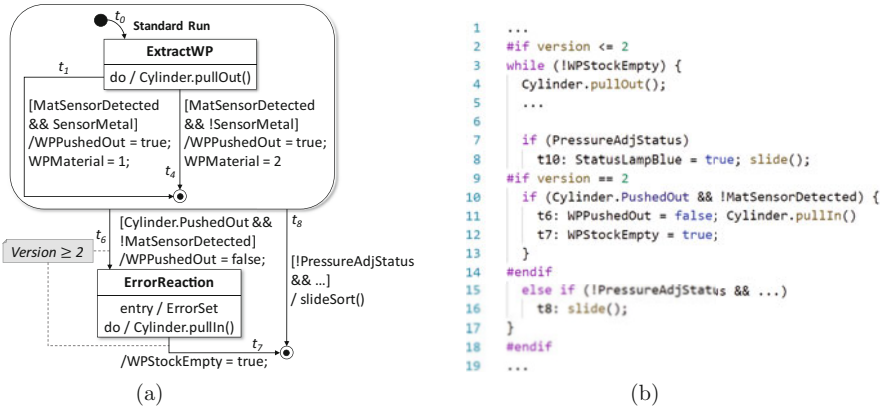
In this section, we discuss different possible *evolution scenarios* of model-based product lines and describe techniques to properly handle the impact of those evolution scenarios on the different kinds of product-line artefacts.

7.2.1 Evolution of Software Variants

Under idealistic circumstances, evolution of software product lines would be conducted in a properly preplanned, offline, and forward manner as follows:

- **Step 1:** updating the feature model
- **Step 2:** adapting the solution-space and model-based testing artefacts and the corresponding feature mappings affected by the update
- **Step 3:** deriving updates of software variants for those product configurations affected by the changes, and
- **Step 4:** (re-)generating and (re-)executing test cases required for ensuring the correctness of the changes on the affected software variants

In practice, evolution usually takes place at the level of particular variants rather than at the level of the whole product-line representation [Nev+15]. For instance, a *clone-and-own approach* is frequently used to make changes to a particular model-/program variant and then to propagate those changes by copying and pasting/replacing the affected model/code parts in other variants for which the change is also relevant [Ape+13]. However, if not conducted carefully, such an ad hoc approach is inherently prone to causing continuous decay of the overall product-line structure (e.g. causing either redundant-code or missing-code anomalies in a



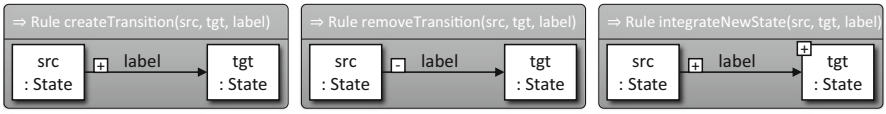

```
...
#if version <= 2
while (!WPStockEmpty) {
  Cylinder.pullOut();
  ...

  if (PressureAdjStatus)
    t10: StatusLampBlue = true; slide();
#if version == 2
  if (Cylinder.PushedOut && !MatSensorDetected) {
    t6: WPPushedOut = false; Cylinder.pullIn()
    t7: WPStockEmpty = true;
  }
#endif
  else if (!PressureAdjStatus && ...)
    t8: slide();
}
#endif
...
```

Fig. 7.8 Test model of variant 2 in version 2. (**a**) Test model. (**b**) Code

particular variant), which, in the worst case, may lead to inconsistent and erroneous variant implementations and/or quality-assurance artefacts.

Figure 7.1b summarises the evolution steps of the xPPU product line considered in the following examples. Consider variants $v1$, $v2$, and $v3$, as described in the previous section, to constitute the *initial version 1* of the xPPU product line. In a first evolution step, leading to *version 2* of the xPPU product line, a *revision* of the xPPU functionality takes place, resulting in adding error-handling capabilities. To this end, a new model fragment, comprising the additional state *ErrorReaction* and corresponding *transitions* for error handling, is added to those test-model variants affected by this change. In particular, the new behaviour is supposed to be added to the existing variants $v1$ and $v2$ of the xPPU product line, whereas variant $v3$ remains without error handling. Figure 7.8 depicts the updated version of the test model of variant $v2$, now containing the newly added model fragment, where a similar change is applied to the respective test model of variant $v1$ (e.g. by applying clone and own of the new fragment from $v2$ to $v1$ or vice versa). In order to master those kinds of product-line evolution scenarios in a model-based setting, we are faced with two major challenges, namely:

- Evolution steps are often conducted in an ad hoc manner and without a proper documentation. Hence, in order to *understand* and *propagate* those changes to other affected variants as well, they have to be properly *represented* in a well-defined way.
- Evolution steps are potentially conducted to all possible artefacts of product-line representations. This may impact the integrity and consistency of further artefacts at the same level, as well as at any other level of representation. Hence, in order to *make explicit* those changes for subsequent engineering steps (e.g. family-based quality assurance), they have to be properly *integrated as additional knowledge* into product-line artefacts.



(a) (b) (c)

Fig. 7.9 Edit operations for statecharts (Abstract syntax). (**a**) Create transition operation. (**b**) Remove transition operation. (**c**) Integrate state operation

To cope with these challenges, we utilise and combine two techniques, namely (1) *model differencing and model patching* from model-based software engineering [Men02] and (2) *annotation of presence conditions* from product-line engineering [CE00] (Fig. 7.9).

Model Differencing and Model Patching Model-differencing approaches are used for deriving and representing common and differing parts between model versions/variants [Men02]. Here, we employ model differencing techniques for handling variants and revisions of product-line modelling artefacts. To this end, *state-based differencing* of two given versions/variants, $v1$ and $v2$, of a model aims at identifying similar parts within $v1$ and $v2$ on the basis of the current states of both models. We refer to Sect. 10.1.1 for an in-depth description of model-differencing and patching techniques and will only briefly describe the corresponding notions and concepts in the following.

There are various different techniques to decide whether element a of model $v1$ and element b of model $v2$ are considered *similar*. For instance, equality of (unique) *identifiers* or *names* of elements are frequently used criteria for comparing model elements. Based on those criteria, a pair (a, b) of model elements considered similar is called a *correspondence*, where a and b are said to correspond to each other. A *matching* between models $v1$ and $v2$ is a set of (all) correspondences between the elements of $v1$ and $v2$. Given such a matching, a *directed delta* (*difference*) comprising a set of change actions from model $v1$ to model $v2$ can be derived as follows:

- Each model element of $v1$ (or $v2$, respectively) not matched to any other model element leads to a change action that deletes (or creates, respectively) this element.
- Each non-identical property (e.g. a name) of two corresponding elements yields a change action overwriting this property with the value apparent in model $v2$.

Each change action derived this way into a directed delta corresponds to a low-level change being observable between both models, where, however, the actual modification may have been applied in a different way in case of ambiguity (see Sect. 10.1.1 for details). In addition, those corresponding low-level changes consider both models simply as plain directed graphs without considering any further well-formedness rules or necessary abstractions needed for understanding the impact of evolution steps. Instead, model differences should be represented in

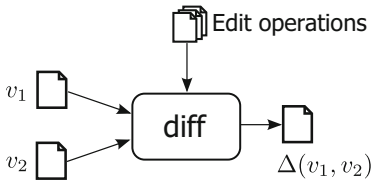


Fig. 7.10 High-level model differencing

a structured and preferably human-readable way (e.g. in terms of *edit operations* corresponding to editing commands in a visual modelling environment). To this end, we further consider *high-level differencing* based on such edit operations for a suitable representation of model differences [KKT11] (see Fig. 7.10). An edit operation groups (several) change actions into one *change set* leading to a so-called *lifting* of differences to a higher abstraction level. Hence, each edit operation obeys an interface consisting of two parts:

- A *difference* $\Delta(v_1, v_2)$ consists of a sequence of *edit steps* $s_1 \ldots s_n$ that when applied to model variant/version $v1$ in exactly this order will yield model variant/version $v2$.
- An *edit step* invokes an *edit operation* and supplies appropriate actual parameters for applying the respective changes to a given model.

Edit operations may be defined and implemented using recent techniques, for instance, declarative graph transformation rules [KKT11]. Simplified rules for edit operations on STATECHARTS are presented in Fig. 7.9, being depicted in their abstract syntax. The first two *atomic* operations in Fig. 7.9a and b specify how to create (delete) a given transition, labelled by *label*, between a source state *src* and a target state *tgt*. Based on these atomic operations, a sample *complex* edit operation for creating a new state and connecting this state by a new transition to an existing one is presented in Fig. 7.9c. This complex operation therefore allows to integrate and connect a new state into an existing model by one edit single operation.

For instance, regarding our xPPU example, the difference $\Delta(PPU2_{v1}, PPU2_{v2})$ describing the evolution from version 1 of the test-model variant $v2$ (see Fig. 7.3a) to version 2 (see Fig. 7.8) may be given as follows:

- *IntegrateNewState(*S_0*,ErrorReaction,*t_6*)*: A new state *ErrorReaction* is added and integrated via the (new) transition t_6.
- *CreateTransition(ErrorReaction,Final,*t_7*)*: A new transition t_7 is created, from the previously created state *ErrorReaction* to the existing final state.
- *AddAnnotations(*t_6*,*t_7*, Version* $\geq$ *2)*: Both new transitions t_6 and t_7 are annotated with version information as the new error functionality is only available in version 2 and subsequent versions (see below for more details).

Hence, a high-level difference allows for a proper representation of evolution steps. Furthermore, such a representation can be used for propagating (parts of) changes between different versions/variants, denoted as *model patching* [KKT13]. To this end, we utilise difference $\Delta(v1, v2)$ between two models $v1$ and $v2$ as a *patch* (or edit script) on a third model $v3$ as follows:

- Actual parameters for each edit step $s_k \in \Delta(v1, v2)$ are to be adapted to model $v3$ as elements and/or properties available in $v1$ may not be (identically) available in model $v3$. To do so, a matching between models $v1$ and $v3$ is computed for finding corresponding (and thus appropriate) parameter values, as described earlier.
- Sequential dependencies between edit steps $s_i, s_k \in \Delta(v1, v2)$ are to be derived for computing a (partial) ordering among patch operations. For instance, in $\Delta(PPU2_{v1}, PPU2_{v2})$, the creation of state *ErrorReaction* has to precede the creations of transition t_7 requiring this state as a source state.

Based on this construction, we can apply an (adapted) patch to other models for propagating changes among variants and/or versions [KKR14]. In case of the xPPU example, we may apply patch $\Delta(PPU2_{v1}, PPU2_{v2})$ to xPPU variants $v1$ and $v3$ for introducing error handling (see evolution steps in Fig. 7.1b), instead of manually (re-)creating these changes for all variants [KKR14].

Presence Conditions for Version-Knowledge In the previous section, we already explained the idea of using presence-condition annotations to represent variation points as additional knowledge within solution-space artefacts of software product lines. Based on this concept, a so-called 150% model (e.g. a STATECHART test model for the whole product line) can be defined that superimposes all model variants (i.e. all 100% test models of any derivable software variant) of the product line. In this regard, presence conditions annotate variable model parts with *variant-information* (i.e. propositional conditions over feature-selections), for which they are relevant. In a similar way, presence conditions may be employed to denote *version-information* and to propagate this information among engineering- and quality-assurance artefacts throughout the whole life cycle of an evolving product line. To this end, we introduce (atomic) presence conditions of the form

$$\textit{Version}\ \texttt{relop}\ k,$$

where $\texttt{relop} \in \{<, \leq, \geq, >\}$ as usual, to denote ranges of version numbers (revisions), in which an annotated artefact is—or has been—present in a model- or code fragment of the product line. In order to keep the following presentation graspable, we limit our considerations to a globally consistent and linearly increasing version-history, represented by a single (Integer-valued) meta-variable *Version*. Starting at initial version 1, *Version* is constantly increased by the value 1 after every new revision. We further assume that each revision may include multiple, yet non-conflicting, changes to the same and to different artefacts. Based on the notion of atomic presence conditions, arbitrary *version-history intervals* can be expressed using logical connectives $\wedge$ and $\vee$ as usual (please note that we will use $\wedge$ and $\vee$ in models and `&&` and `||` in code interchangeably in the following). For instance, an artefact annotated with the presence condition

$$(\textit{Version} \geq 2 \wedge \textit{Version} < 6) \vee \textit{Version} \geq 7$$

was not part of the initial version 1 but has been newly added to a model/code artefact in *version 2* but was later (temporarily) removed again in *version 6* and is, from *version 7* on up to the current version, again part of the model/code artefact. As a consequence, artefacts without version annotations are implicitly annotated with the presence condition $Version \geq 1$ (i.e. the artefact existed from version 1 until the current version).

Similar to the integration of all 100% model/code *variants* of a product line into one 150% model/code representation using presence conditions over feature-selections, all 100% model/code *versions* of one single variant can be integrated into one superimposed model using presence conditions over version-intervals. For convenience, we will call the latter representation a *125% model/code* artefact in the following (assuming that differences among different versions are considerably smaller than those between variants). Reconsidering the example in Fig. 7.8a, this model constitutes the 125% test model of xPPU variant *v2*, including both initial version 1 without error handling and version 2 (and all later versions up to the current version) with error-handling capabilities. The model fragment for error handling, consisting of the transitions t_6 and t_7, as well as the state *ErrorReaction*, is therefore annotated with presence condition $Version \geq 2$, whereas all other model elements are not annotated and thus are present in all versions since the initial version. The corresponding 125% code fragment of variant $v2$ is depicted in Fig. 7.8b, where the `#if` block (Lines 9–14) marks the code parts for error handling added during revision 2 of the implementation. Similar updates have been likewise applied to the STATECHART model and respective implementation code of variant $v1$, whereas variant $v3$ has not been affected by this revision.

Concerning the next evolution step, assume the new error handling later to be considered useful also for variant *v3* and therefore added to the respective STATECHART model and implementation code of variant *v3* during *revision 3* of the xPPU product line. As a consequence, the 125% test model of variant *v3* now also contains the model fragment for error handling, as previously added to variants *v1* and *v2*, whereas this fragment is now annotated with the presence condition $Version \geq 3$ and likewise for the implementation code of *v3*. In contrast, variants *v1* and *v2* remain unchanged during revision 3.

In revision 4 of the xPPU product line, however, error handling is removed, again, but only from variant $v2$ as it has been shown to be inappropriate for this particular xPPU configuration, whereas it remains in variants $v1$ and $v3$. Figure 7.11a shows the 125% test model of variant *v2* after revision 4, in which the presence conditions of the transitions have been updated, accordingly, to

$$Version \geq 2 \wedge Version < 4,$$

and, similarly, for the 125% implementation code of variant $v2$.

Finally, let us consider a special case of product-line revision in which the presence/absence of entire variants changes as part of an evolution step. For instance, as part of revision 5, it has been decided that variant $v3$ is no more supported by the xPPU product line. Hence, *all* solution-space artefacts related to

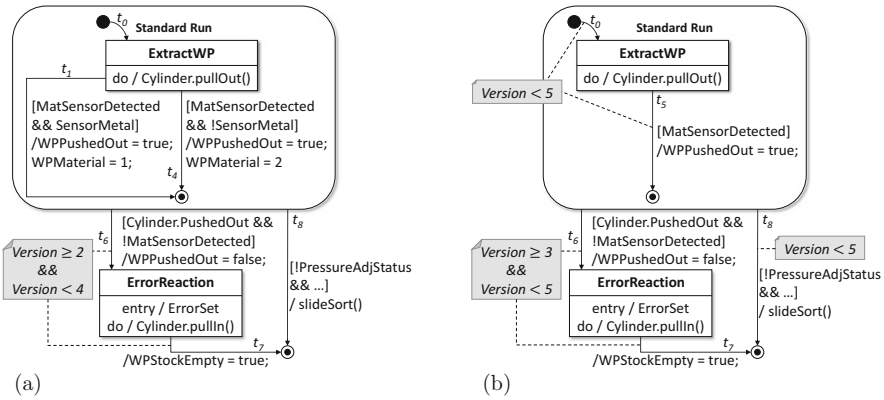


Fig. 7.11 Further evolution steps of variants 2 and 3. (**a**) Test model of variant 2 in version 4. (**b**) Test model of variant 3 in version 5

$v3$ are disabled from version 5 on for variant $v3$, as illustrated in the corresponding 125% model in Fig. 7.11b (and similarly, for the implementation code of variant $v3$). In contrast, variants $v1$ and $v2$ are unaffected by revision 5.

To generalise, updating a presence condition φ of a product-line artefact of a 125% representation to presence condition φ' as a result of a revision k consists of three possible cases:

- $\varphi' := \varphi \vee \mathit{Version} \geq k$ if the artefact is *added* during revision k
- $\varphi' := \varphi \wedge \mathit{Version} < k$ if the artefact is removed during revision k, and
- $\varphi' := \varphi$ if the artefact remains unchanged during revision k

which can be automatically derived from respective model/code difference-rule applications, as described above.

7.2.2 Evolution of Software Product Lines

As described before, an idealistic view on product-line evolution should always start with the evolution of the problem-space specification (i.e. the feature model), followed by necessary adaptations to solution-space engineering artefacts (i.e. 150% models and code).

Evolution of Problem-Space Artefacts Based on the *syntactic differences* between a feature model and its revised version due to a feature-diagram edit applied during product-line evolution, the *semantic impact* may be classified in terms of the potential changes of those edits caused on the set of valid configurations (i.e. depending on whether valid configurations may become valid and/or vice versa) [Bür+15b, TBK09].

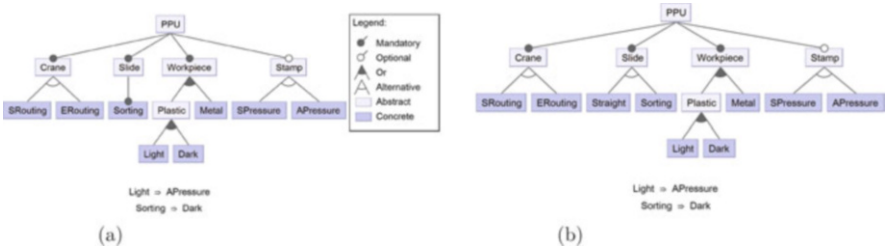


Fig. 7.12 Feature-model evolution scenarios. (**a**) Feature model version 2. (**b**) Feature model version 3

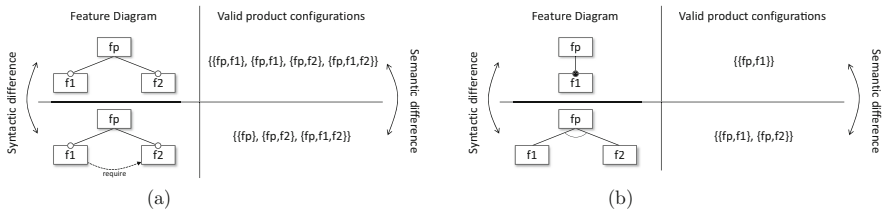


Fig. 7.13 Feature-model edit operations. (**a**) Operation 1. (**b**) Operation 2

As a first example of feature-model evolution, consider the feature-diagram edit from the initial model version in Fig. 7.5a to the new version in Fig. 7.12a. Here, the additional cross-tree constraint *Sorting* $\Rightarrow$ *Dark* has been added to restrict the set of valid configurations of the xPPU product line. Semantically, this edit removes variant *v3* from the set of valid configurations, which has been referred to as *revision 4* from the perspective of software-variant evolution in the previous subsection. A corresponding model-differencing rule for this kind of (atomic) edit operation (see Fig. 7.13a) is therefore classified as *specialisation* step.

As a second example, consider the feature-diagram revision from the model version in Fig. 7.12a to the new model version in Fig. 7.12b. This change consists of a *complex* edit operation involving two *atomic* edits: (1) adding a new feature node *Straight* to parent feature *Slide* and (2) converting the two sibling singleton feature node *Straight* and *Sorting* into an *alternative group*. This edit now enables customers, in addition to the previous variants, to further configure xPPU variants having a *Standard Ramp* with only one *Slide* (i.e. without *Sorting* of WP). The corresponding model-differencing rule for this kind of (complex) edit operation (see Fig. 7.13b) is therefore classified as *generalisation* step.

In addition to the classification of the semantic impact of feature-model edits, the differencing information can, again, be used to annotate model parts with version-information in a similar way, as already described above for STATECHART models and implementation code. The resulting feature model, unifying variant, and version information at the same level of abstraction are also referred to as Hyper-Feature-Models [SSA14].

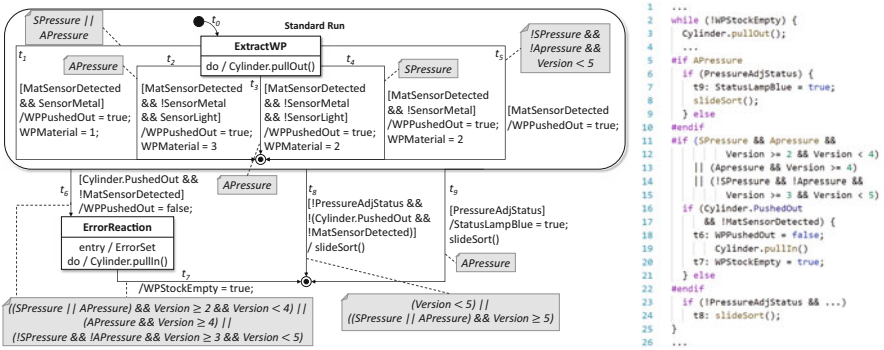


Fig. 7.14 175% test model. (**a**) Test model. (**b**) Code

Evolution of Solution-Space Artefacts The evolution of solution-space artefacts can be handled with similar techniques, as already described for software-variant evolution (i.e. by combining model differencing and presence-condition annotations). However, during the evolution of entire product lines, solution-space representations following the idea of 150% models/code now have to integrate all versions of all model variants by superimposing the 125% model-/code-parts of all variants. In those models, presence conditions have to relate variant- and version-information in a consistent way, in order to express which model-/code parts are (or have been) present in which model-/code variant in which version of the product line. Consequently, we call this kind of representation 175% model/code. We, again, refer to Fig. 7.7b for an overview of the terminology for the different kinds of representations described so far.

As an example, reconsider the five revisions of the xPPU product line, as described previously at the level of software variants, now being applied at the level of the product-line representation. The resulting 175% STATECHART model, including all five revisions of all three variants, is depicted in Fig. 7.14. Most remarkably, the presence conditions

$$((SPressure \vee APressure) \wedge Version \geq 2 \wedge Version < 4) \vee$$
$$(APressure \wedge Version \geq 4) \vee$$
$$(!SPressure \ \wedge \ !APressure \wedge Version \geq 3 \wedge Version < 5)$$

of the transitions $t6$ and $t7$ precisely reflect the version-history of error handling in the xPPU product line from version 1 to version 5 as follows:

- The clause in row (1) states that error handling is available in product configurations corresponding to variants $v1$ and $v2$ from version 2 to version 3.

- The clause in row (2) states that error handling is no more available in the product configuration corresponding to variant $v2$ from version 4 but remains available in variant 1.
- The clause in row (3) states that error handling is available in the product configuration corresponding to variant $v3$ from version 3 to version 5 (in which the entire variant is finally removed from the xPPU product line).

Similarly, transition *t5* is annotated with the presence condition

$$(!SPressure \ \wedge \ !Apressure) \wedge Version < 5$$

to denote that this transition is present in variant $v3$ from the initial version up to version 4 as it is removed during revision 5. Finally, the annotation

$$(Version < 5) \vee ((SPressure \vee APressure) \wedge Version \geq 5)$$

ensures that transition *t8* will be removed from variant 3 in version 5 but will remain in variants 1 and 2.

The 175% implementation code in Fig. 7.14b shows the corresponding code parts of transitions t6, t7, t8, and t9. Here, the code parts nested in the `#if` block in Lines 5–10 are present in all versions of all variants having feature *APressure* selected, whereas the `#if` block in Lines 11–22 conditionally adds code for error handling, depending on the particular variant and version under consideration.

7.2.3 Evolution of Model-Based Testing Artefacts

Concerning model-based testing artefacts of evolving software product lines, we have to adapt the notions of SPL test case and (complete) SPL test suite [Bür+15a], accordingly, to also take version-information into account, as provided by a 175% test model. To this end, the presence condition of an SPL test case now incorporates both variant- and version-information, thus denoting the set of variants together with a sub-range of their versions required for the test case to be applicable.

As an example, instead of using an automated test-generation tool, consider a tester to manually add a test case to a test suite for the xPPU product line. Based on the 175% test model, the corresponding presence condition for that test case can be derived by conjugating the corresponding presence conditions of those transitions traversed by this test case. For instance, the test case

$$tc3 := (t0, t3, t8)$$

corresponding to the path $t0, t3, t8$ with presence condition *true* from transition $t0$, *APressure* from transition $t3$, and

$$(Version < 5) \vee ((SPressure \vee APressure) \wedge Version \geq 5)$$

from transition $t8$ results in the conjugated presence condition:

$$(t0)\ (true) \wedge$$
$$(t3)\ (APressure) \wedge$$
$$(t8)\ ((Version < 5) \vee ((SPressure \vee APressure) \wedge Version \geq 5)).$$

In addition, the notion of complete SPL test suite has to be likewise enhanced, now requiring that every test goal is covered on every *variant* and *version*, including this test goal, by at least one SPL test case being applicable to this particular *version* of that *variant*. Table 7.1 shows a minimal set of test cases required for complete test coverage of the 175% test model, as shown in Fig. 7.14b. Each row corresponds to a test case, represented by a path through the test model, together with the presence condition and the set of test goals covered by that test case in the respective variants and versions. For example, test case *tc1* covers the test goals *t0*, *t1*, and *t8* on variants *v1* and *v2* in all their versions. Hence, test goal *tc1*, which is only present in variant *v1* and *v2*, is completely covered by this test case on all versions in which it occurs. In contrast, test goals *t0* and *t8* are also present in version *v*3, thus requiring a further test case *tc6*, covering test goals *t0* and *t8* on variant *v3* in all of its versions. In addition, the test case also covers test goal *t5*. The further test cases of the given test suite can be derived accordingly.

As illustrated by this example, the derivation and evolution of model-based testing artefacts (i.e. test goals and corresponding SPL test suites) requires additional knowledge as provided by the feature model and the 175% test model, which will be described in the following section about *co-evolution*.

7.3 Co-evolution

In this section, we discuss the co-evolution scenarios ①–⑥ of model-based product lines, as depicted in Fig. 7.7a, and describe how to ensure consistency among the different product-line engineering- and quality-assurance artefacts involved.

7.3.1 Co-evolution of Software Product Lines and Product Variants

Co-evolution scenario ① is concerned with the evolution of software variants due to changes in the software product line. Following a brute-force approach, all existing model/code variants might be simply re-generated by deriving from the respective 175% model/code the corresponding 100% representations according to

Table 7.1 175% test suite

Test case	Presence condition	Variants	Versions	Goals
$tc_1 = (t_0, t_1, t_8)$	(SPressure $\vee$ APressure) $\wedge$ ((Version $<$ 5) $\vee$ ((SPressure $\vee$ APressure) $\wedge$ Version $\geq$ 5))	v1, v2	1, 2, 3, 4, 5	t0, t1, t8
$tc_2 = (t_0, t_2, t_9)$	APressure	v1	1, 2, 3, 4, 5	t0, t2, t9
$tc_3 = (t_0, t_3, t_8)$	APressure $\wedge$ ((Version $<$ 5) $\vee$ ((SPressure $\vee$ APressure) $\wedge$ Version $\geq$ 5))	v1	1, 2, 3, 4, 5	t0, t3, t8
$tc_4 = (t_0, t_4, t_8)$	SPressure $\wedge$ ((Version $<$ 5) $\vee$ ((SPressure $\vee$ APressure) $\wedge$ Version $\geq$ 5))	v2	1, 2, 3, 4, 5	t0, t4, t8
$tc_5 = (t_0, t_1, t_6, t_7)$	(SPressure $\vee$ APressure) $\wedge$	v1	2, 3, 4, 5	t0, t1, t6, t7
	(((SPressure $\vee$ APressure) $\wedge$ Version $\geq$ 2 $\wedge$ Version $<$ 4) $\vee$(APressure $\wedge$ Version $\geq$ 4) $\vee$ (!SPressure $\wedge$!APressure $\wedge$ Version $\geq$ 3 $\wedge$ Version $<$ 5))	v2	2, 3	t0, t1, t6, t7
$tc_6 = (t_0, t_5, t_8)$	(!SPressure $\wedge$!APressure) $\wedge$ Version $<$ 5) $\wedge$ ((Version $<$ 5) $\vee$ ((SPressure $\vee$ APressure) $\wedge$ Version $\geq$ 5))	v3	1, 2, 3, 4	t0, t5, t8
$tc_7 = (t_0, t_5, t_6, t_7)$	(!SPressure $\wedge$!APressure) $\wedge$ Version $<$ 5) $\wedge$ (((SPressure $\vee$ APressure) $\wedge$ Version $\geq$ 2 $\wedge$ Version $<$ 4) $\vee$(APressure $\wedge$ Version $\geq$ 4) $\vee$ (!SPressure $\wedge$!APressure $\wedge$ Version $\geq$ 3 $\wedge$ Version $<$ 5))	v3	3, 4	t0, t5, t6, t7

the corresponding product configuration and the new version number of the evolved product line.

For instance, in the first evolution step applied to the 150% xPPU test model shown in Fig. 7.6a, error handling has been added to the variants *v1* and *v2* (see Fig. 7.1b). As a consequence, one may simply re-generate the corresponding 100% model variants of all possible configurations to ensure consistency with the product line. However, in this way, also those model variants not affected by any changes would be re-generated, which becomes highly inefficient in case of larger product lines with hundreds or even thousands of possible configurations. To avoid this, the additional information gained from model differences and respective presence conditions in 175% representations allow for a more fine-grained change-impact analysis, as will be described in the following.

Problem-Space Co-evolution Scenario ① As described in the previous section, a semantic classification of syntactic feature-model edits can be helpful in proving the potential impact of problem-space evolution on the validity of existing software variants:

- *Generalisation* indicates that (1) all existing variants still correspond to a valid configuration and (2) new variants corresponding to previously invalid configurations may be derivable after the feature-model update.
- *Specialisation* indicates that (1) some existing variants may become invalid and (2) no new variants are derivable after the feature-model update.
- *Refactoring* indicates that the set of valid variants does not change after the feature-model update.
- *Arbitrary edit* indicates that (1) some existing variants may become invalid and (2) new variants may be derivable after the feature-model update.

Based on this information, further investigations on the change impact with respect to the validity or invalidity of particular configurations can be conducted in a systematic and automated way (e.g. using constraint solvers [TBK09]). For instance, the edit applied to the initial version of the xPPU feature diagram in Fig. 7.5a, leading to the new version in Fig. 7.12a, constitutes *specialisation* as variant *v3* becomes invalid. In contrast, the second feature-diagram evolution, leading to the version in Fig. 7.12b, is *generalisation* as we add the new optional kind of *Straight* slide, which leads to a new set of variants having this slide, optionally in addition to the old ones. In these cases, where new variants arise, the 175% test model can be used to derive additional test cases for specifically assuring the corresponding implementation variants. Otherwise, in cases of variants becoming invalid, the presence-condition information attached to existing test cases can be used to remove invalid test cases from SPL test suites.

Solution-Space Co-evolution Scenario ① As described in the previous section, evolution of solution-space artefacts potentially causes changes in parts of 175% model/code representations. Hence, assuming that an evolution scenario yields a new *version k* of the product line, a closer investigation of the presence conditions after updating 175% models/code to version *k* provides information about affected

software variants. In particular, for an artefact annotated with a presence condition having a newly added sub-clause of the form

$$(\varphi \wedge \mathit{Version}\ \texttt{relop}\ k),$$

with φ being a propositional formula over features as described previously, two cases arise:

- If `relop` is equal to $<$, then the artefact has been removed during revision k from all variants satisfying φ
- If `relop` is equal to $\geq$, then the artefact has been added during revision k to all variants satisfying φ, respectively.

Based on this information, the overall subset of variants affected by changes on solution-space artefacts performed in revision k can be obtained without additional effort. In addition, the corresponding updates to 100% model/code representations of the affected variants can be conducted automatically (e.g. by means of patches derived from this information).

For instance, consider the transitions $t6$ and $t7$ added for error handling to the 150% test model in Fig. 7.14. For variant $v3$, these transitions become present in versions 3 and 4 due to the sub-clause

$$(!\mathit{SPressure}\ \wedge\ !\mathit{APressure} \wedge \mathit{Version} \geq 3 \wedge \mathit{Version} < 5)$$

in the presence condition of $t6$ and $t7$ in the updated 175% model.

In contrast to co-evolution scenario ①, scenario ② is concerned with the evolution of software product lines due to changes directly applied to individual software variants. Again, we consider co-evolution of both problem-space and solution-space artefacts.

Problem-Space Co-evolution Scenario ② Given an (evolving) set of software variants corresponding to a set of all valid configurations of a product family, the problem of deriving a corresponding configuration model (e.g. a feature diagram) that precisely captures this set of valid configurations is frequently known as *feature-model mining* or *product-line extraction*. We will not go into detail about this particular evolution scenario but rather refer to recent literature about different techniques addressing this problem [Alv+08, MBB16].

Solution-Space Co-evolution Scenario ② Given a set of N software artefacts (e.g. models or code) corresponding to a set of valid software variants of a product family, the problem of deriving an integrated representation superimposing similarities among those representations is frequently referred to as N-way merging [RC13].

N-way Model Merging and Model Integration An overview of the three steps performed during N-way merging in general is depicted in Fig. 7.15 and can be described as follows.

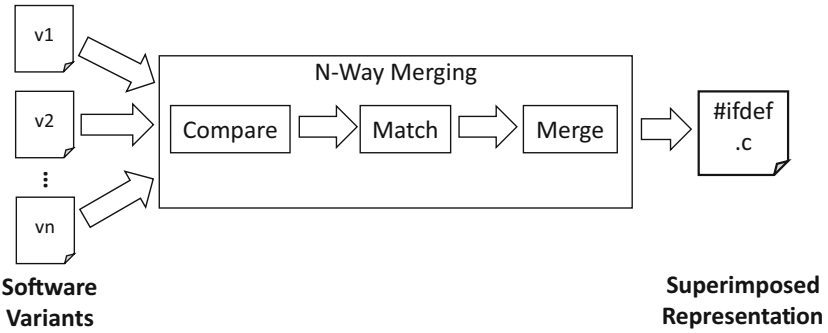


Fig. 7.15 N-way merging

- **Compare.** In this step, elements (e.g. code lines or model parts) of the different models are compared and their similarity is measured with respect to a given *similarity criterion*. Thus, for each possible set of presumably similar elements originating from different models, a similarity value between 0 and 1 is computed. To this end, the same element properties may be used, as already previously described for model differencing (e.g. the types and names of elements).
- **Match.** Based on the compare values, those subsets of elements are being matched (i.e. considered to be same) that constitute the (presumably) most similar elements among the different models. As a result, a complete match contains a complete partitioning of all model elements from all N models. Although various different matching algorithms can be used in this step, a frequently applied greedy-based heuristic incrementally selects further subsets of unmatched elements having the best remaining similarity value, until all elements are finally matched. Similar to the notions already described in the previous section about model differencing, elements matched for merging are referred to as corresponding (see Sect. 7.2).
- **Merge.** In the merge step, all previously matched elements are integrated into the resulting merged model. To this end, the *union-merge operator* is frequently used in practice, which is based on the assumption that all matched elements are complementary (i.e. being literally the same element appearing in different variants and/or versions) and should therefore be *unified* into one element within the superimposed representation. In contrast, unmatched elements (i.e. residing in singleton subsets after matching) are inserted as singleton elements without any unification with other elements.

As described previously, one key aspect of our approach is to use presence conditions for representing variant- and version- information in a uniform and declarative way. In order to facilitate consistency-preserving artefact co-evolution, we automatically integrate presence conditions during the merging step of N-way merging. In particular, we integrate variability information using *variation*

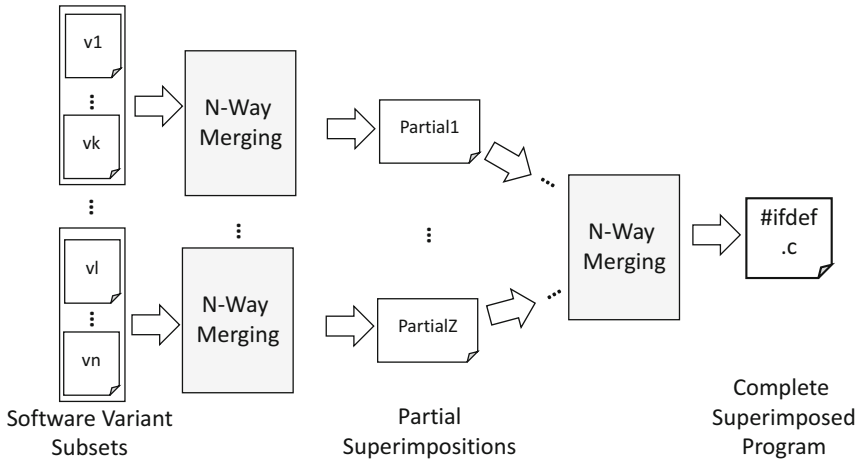


Fig. 7.16 Incremental N-way merging of software variants

points/revision points in terms of conditional model/code fragments over presence conditions rather than (meta-)annotations, as described previously. This alternative representation enables a seamless application of many recent family-based analysis techniques and tools, which are mostly based on this so-called *variability encoding* [Ape+13].

Based on the technique of N-way merging, the following basic co-evolution scenarios can be handled in an automated way:

- Given a set of N 100% models/code artefacts corresponding to the different versions of the same model/code variant, N-way merging results in a 125% representation.
- Given a set of N 100% models/code corresponding to the different variants of the same model/code version, N-way merging results in a 150% representation.

In the case of multiple subsequent versions of either a software variant or an entire product line, the set of N representations is usually not available all at once but rather emerge over time due to evolution scenarios. Hence, merging has to be applied incrementally and/or on subsets (see Fig. 7.16). To this end, the availability of integrated variability-information in terms of variation points/revision points within (partially merged) models allows for incrementally matching and merging further variants and/or versions into product-line representations throughout the entire life cycle of evolving product lines. Based on the technique of incremental N-way merging, advanced co-evolution scenarios can be handled, such as the following:

- Given a 125% representation comprising the different versions of one particular variant up to revision $k-1$ and a 100% representation as a result of revision k of

that variant, their merging yields a 125% representation comprising all versions of that variant up to revision k.

- Given a set of N 125% representations comprising the different versions of a set of N variants, their N-way merging yields a 175% representation comprising all variants with all their versions.
- Given a set N 150% representations comprising the different versions of a product line, their N-way merging yields a 175% representation comprising all these variants with all their versions.

As an example, recall the evolution scenario of version 1 of the 150% test model in Fig. 7.6, leading to version 2, which shall be now conducted at the level of variants. During the revision leading to version 2, the test models of the variants $v1$ and $v2$ (see Fig. 7.8) are evolved to now contain error handling, whereas the test model of variant $v3$ (see Fig. 7.4) remains unchanged. We now consider this evolution scenario at the level of the implementation code, and we focus on the code parts implementing the transitions below the so-called *standard-run* state (see Fig. 7.14). To this end, we consider the representation of source code in terms of CFA, constituting a program abstraction frequently used by many program-analysis and testing tools [Bür+15a]. States (or nodes) of a CFA correspond to control-flow locations (i.e. lines of source code) in a given program, whereas edges denote different kinds of basic imperative control flows (i.e. control-flow sequences, control-flow branches, and control-flow loops) as usual, being either labelled with (basic blocks of) program statements or expressions, respectively. This representation allows us to apply principles from model differencing and model merging, as described above, to STATECHART models, as well as to implementation code in a similar way. Figure 7.17d shows the corresponding extract from the CFA of the 175% code of the product line in version 1, whereas Fig. 7.17a–c shows similar

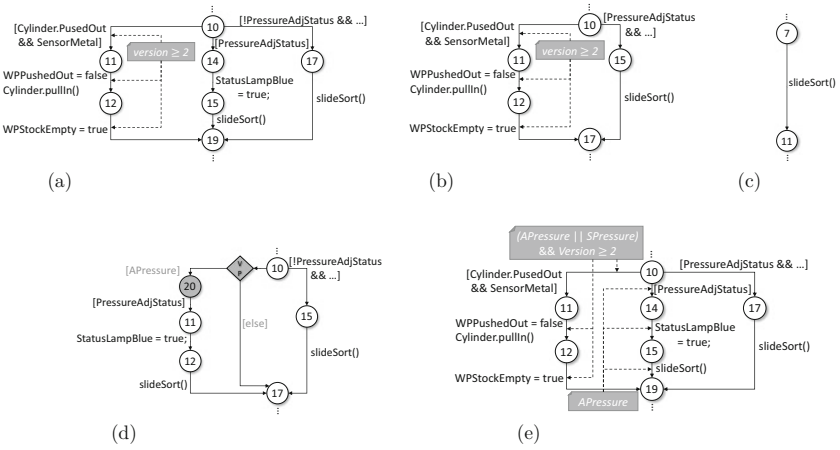


Fig. 7.17 Incremental N-way CFA merging. (**a**) CFA of v1 in Version 2. (**b**) CFA of v2 in version 2. (**c**) CFA of v3 in version 2. (**d**) 175% CFA in version 1. (**e**) 175% CFA in version 2

extracts from the 100% CFA representations of variants $v1$, $v2$, and $v3$ after revision 2. Figure 7.17e therefore depicts the 175% CFA resulting from merging the 175% CFA representation and the 100% CFA representation, thus yielding the 175% CFA representation after revision 2. Hence, by (incrementally) applying N-way merging in this way, similarities among variants and/or versions are reflected in the resulting 175% CFA. For instance, path 10-17-19 is present in all variants and all versions, whereas path 10-14-15-19 is present in all versions of variant $v1$ (having feature *APressure*), and path 10-11-12-19 (for error handling) is only present in version 2 of variants $v1$ and $v2$.

7.3.2 *Co-evolution of Software Product Lines and Model-Based Testing Artefacts*

Concerning scenario ③, the co-evolution of model-based testing artefacts according to evolving product-line representations can be conducted in a straightforward manner. Based on the combined variant/version-information in the updated 175% test-model specification, family-based test generation can be applied for updating the SPL test suite in order to become consistent with the latest revision. Concerning the application of test cases selected for retesting variant implementations being potentially affected by the changes, again, the additional information in the updated 175% implementation code can be used for change-impact analysis similar to principles known from regression testing [Loc+12].

For instance, concerning the SPL test suite, as shown in Table 7.1, the additional test case *tc5* has to be added to the test suite after adding error handling to variants *v1* and *v2* during revision 2. In addition, after removing variant *v3* during revision 5, test cases *tc6* and *tc7* both become invalid as they are only executable on that variant.

Concerning scenario ④, co-evolution of manual updates of SPL test suites and corresponding product-line representations can be conducted by deriving variant/version-information for newly added test cases from the the 175% test model.

For instance, a tester may decide to add the additional test case $tc8 = (t0, t1, t9)$ into the SPL test suite, as shown in Table 7.1, to test the correct interplay between transitions *t1* and *t9* in variant *v1*. The corresponding presence condition obtained from the respective path in the 175% test model is given as

$$(SPressure \vee APressure) \wedge APressure,$$

thus being valid for any version of all variants having feature *APressure* selected. In contrast, test case $tc9 = (t0, t4, t9)$ is invalid as the presence condition ($SPressure \wedge APressure$) contradicts the feature model in all versions of the xPPU product line.

Similarly, the impact of manual removals of test cases from SPL test suites on the test coverage can be investigated on the 175% test model. For instance, if test case *tc2* is removed from the SPL test suite, as shown in Table 7.1, test goal *t0* is no longer covered in any version of all variants containing this goal.

7.3.3 Co-evolution of Product Variants and Test Artefacts

Finally, co-evolution scenarios ⑤ and ⑥ can be handled by sequentially composing the different scenarios for co-evolving product-line representations, as described above, namely:

- Scenario ⑤ can be handled by first conducting scenario ④ and then scenario ①.
- Scenario ⑥ can be handled by first conducting scenario ② and then scenario ③.

7.4 Conclusion

In this chapter, we described a model-based framework for systematic round-trip engineering and quality assurance of continuously evolving software product lines. The presented methodology utilises two major techniques from model-based software engineering, namely:

- *Model differencing* and *model merging* for automated comparison and integration of software variants and versions of an evolving software product line, and
- *Knowledge-carrying software* for the integration of additional information about variant- and version-specific software artefacts into engineering and quality-assurance processes.

This combination ensures consistency among engineering and quality-assurance artefacts throughout the entire life cycle of evolving product lines and facilitates the application of efficient *family-based* product-line analysis strategies to both variant- and version-rich software systems, as well as arbitrary combinations thereof.

To conclude this chapter, we briefly outline a road map for possible future research directions based on the proposed framework.

Besides the model/code artefacts and the corresponding knowledge on product-line representations, as discussed throughout this chapter, further types of artefacts and meta-information annotations might be considered in a similar way due to the generality and generic nature of the presented approach and tools.

In addition to model-based testing, further quality-assurance techniques (e.g. model checking, NFP analysis, etc.) might be lifted in a similar way to become applicable for family-based analyses of both variants and versions in a unified way.

Finally, other kinds of evolution scenarios and co-evolution scenarios might be taken into account. For instance, in practice, a large repository of continuously

evolved legacy test cases exists for which corresponding variant-/version-knowledge is often not available, incomplete, and error prone. Hence, precise techniques for reverse-engineering (or learning) variant-/version-information from those existing artefacts is a crucial open issue for future research.

7.5 Further Reading

Further details about tool support and experiences gained from experimental evaluation results obtained for the different techniques can be found in recent publications summarised in the following. In addition to the references already provided in the different subsections of this chapter, the following references also contain further information about related work on the different approaches considered in this chapter.

A survey about different product-line analysis techniques, including family-based Analysis, can be found in [Thü+14a]. In particular, a tool implementation of the family-based test-suite generation approach based on the software model checker CPACHECKER [Bey+04, D B+13] can be found in Bürdek et al. [Bür+15a], and evaluation results for applying the approach to the PPU case study can be found in Lochau et al. [Loc+14]. The evaluation results show remarkable gains in efficiency under stable effectiveness of applying family-based test generation, as compared to a variant-by-variant approach. This tool can be extended, accordingly, to handle combinations of variant- and version-knowledge, as described in this chapter.

The representation of variability information by means of presence-condition annotations has been initially proposed by Czarnecki et al. as part of their template-based approach for product-line modelling [CE00].

An alternative approach for conceptually integrating variant- and version-information into one representation based on the delta-modelling approach has been proposed by Lity et al. in [Lit+18]. A detailed description of re-engineering the xPPU case study as a product line for model-based testing can be found in [Lit+15]. A general description of challenges in testing product lines can be found in [McG01].

An overview on model-versioning techniques and tools may be found in [ASW09]. Concerning model differencing techniques, in particular as described in this chapter, a dedicated overview can be found in [Kol+09]. Among others, the SILIFT framework allows for a rule-based specification of corresponding model-transformation operators, being applicable to arbitrary input models in a generic way [KWN05]. Among others, this tool has been successfully applied to efficiently and effectively compute and classify differences between FODA feature diagrams, as described in this chapter [Bür+15b]. This approach is, in general, adaptable to any Eclipse Modeling Framework (EMOF)-based modelling language, such as STATECHART test models, as considered in this chapter. This tool can be extended, accordingly, to also compute model differences and N-way model merges of other

product-line artefacts like STATECHART test models and CFA-based representations of implementation code, as described above. Finally, further reading on model-merging notions and techniques can be found, among others, in [Men02] as well as in [RC13].