Int J Software Informatics, Volume 9, Issue 2 (2015), pp. 153–175
International Journal of Software and Informatics, ISSN 1673-7288
©2015 by ISCAS. All rights reserved.

E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Automating Test Case Selection in Model-Based Software Product Line Development

Alexander Knapp[1], Markus Roggenbach[2], and Bernd-Holger Schlingloff[3]

[1] (Universität Augsburg, Germany)

[2] (Swansea University, Wales, UK)

[3] (Humboldt Universität and Fraunhofer FOKUS, Germany)

Abstract We address the problem of how to select test cases for products in a controlled model-based software product line development process. CVL, the common variability language, gives a framework for materialisation of product models from a given base model, variability model and resolution model. From such product models, software products can be derived. In practise, test case development for the product line often is independent from the product development. Therefore, the problem arises which test cases can be applied to which products. In particular, the question is whether a test case for one specific product can be also used for a "similar" product. In this paper, we show how the expected outcome of a test case to a product in a model-based software product line development can be determined. That is, we give a procedure for assigning the outcome of a given test case on an arbitrary member of a software product line. We recall the relevant definitions for software product line engineering, describe our approach, and demonstrate it with the example of a product line of super-automatic espresso machines.

Key words: software product lines; model-based testing; test colouring; UML

Knapp A, Roggenbach M, Schlingloff BH. Automating test case selection in model-based software product line development. *Int J Software Informatics*, **Vol.9, No.2 (2015): 153–175. http://www.ijsi.org/1673-7288/9/i213.htm**

Dedication

To contribute a paper on testing in a special issue dedicated to our academic mentor and friend Bernd Krieg-Brückner is a great honour. Bernd's vision and energy have been and still are an inspiration for us. In Bernd's research, the topic of formal methods for software quality has always been a first-class citizen. In the line of his research on re-use of software design and validation artifacts, we consider here the question of re-using test cases in the context of software product lines. As former members and friends of his working group, we very well remember Bernd's "intelligent coffee machine", connected to the internet and equipped with a speech interface. Thus, it appeared to be more than fitting to use a coffee machine as master example in this paper.

Corresponding author: Bernd-Holger Schlingloff, Email: hs@informatik.hu-berlin.de
Received 2015-01-11; Revised 2015-04-01; Accepted 2015-04-15.

1 Introduction

The concept of a software product line originates by the work of D. Parnas[21]. It has gained much attention by the research and consultancy of the Carnegie Mellon University Software Engineering Institute[6,17]. According to the CMU-SEI definition, "a software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"[5]. SPLs are abundant in today's software-intensive systems: most electronic control units, e.g., in cars or trains, come in multiple variants, as well as consumer products like coffee machines, dishwashers, mobile phones, etc.

A challenge common to the development of these systems, say, coffee machines, is that their built-in software is similar, but not identical in all products; there are slight differences according to the features exhibited by a particular product. Sources of variability include planned diversity for different user groups, evolution and enhancement of products, and re-use of modules from one product in another one. SPL engineering addresses this challenge. The main goal of SPL development is the strategic re-use of software artifacts. There have been various approaches to re-use: by copy and paste, macros, subroutines, modules, objects, components and services. The common problem in all of these approaches is that re-use increases the probability of errors. Therefore, quality assurance for SPLs is of utmost importance. In this paper we address the research question of how to automatise test case selection for the members of a SPL.

A first, naive approach would be to design a test suite specific to each product, sometimes called "separate test case development" in the literature[23]. However, for product lines with hundreds of features and zillions of possible products, separate test case development might not be feasible. Even if the number of actual products for the market is limited, say, to a few dozens, it might be wasteful, since the individual test suites for these products will include much overlap. The individual products of a product line share common features, described in the requirements. All these features must be properly tested by each of the separate test suites.

A second, more advanced approach re-uses test cases by adapting the test suites from one product to another. This is called "opportunistic re-use of existing test cases" in Ref. [23]. Here, the test suite for each individual product is manually re-worked. The applicability and expected outcome of each test case is defined on an ad-hoc basis. Clearly, this is a time-consuming and tedious task.

As an improvement, Reuys et al.[23] suggest to "design test cases for reuse". Here, variability information about scope and applicability is included in each test case during the test design. That is, in this approach test cases are by themselves variant objects. However, this may lead to complex test cases which are difficult to maintain.

In contrast to these, we suggest to use a single, "universal" test suite for a SPL. This universal test suite is developed according to usual test design methods and can be used for all members of the SPL. The universal test suite shall contain tests for all features. This is similar to model-based SPL development, where the base model — also called a "150% model"[22] — contains realizations for all features. Given now a materialisation of the SPL, our idea is to filter the universal test suite in order to determine the subset of test cases "applicable" to this materialisation. Clearly there are syntactic criteria influencing applicability: for example, the system under test must provide all interfaces (input/output signals) which are used by the test case. However, applicability also depends on dynamic product features: even if all necessary interfaces are present in a particular product, a test case might check the correct functionality of a feature which is not realised in this product. In such a case, the test case is not applicable, since it is not clear what the expected outcome of this test case for the respective product should be. With large product lines, determining which test cases from the universal test suite are applicable for a given product can be time-consuming. In this paper, we automatise this task by defining an algorithm for automated test case colouring.

Throughout the paper, we assume that, given a set of requirements, products are developed in a model-based development process by stepwise refinement, from an abstract function model to a concrete implementation model. Test cases are developed independently in another department or business unit, taking the very same requirements as starting point. This assumption reflects common practice in large or medium-sized companies, and is mandatory for safety-critical systems development.

Our contribution is otherwise open to any method of test case development. We do neither assume nor require that test cases are generated automatically or semi-automatically from some model; that is, we are not concerned with "model-based test generation". The test suite can be obtained in any way. For our example in Section 3, the test cases given in Section 4 were designed manually, matching the requirements. However, we presuppose that test development is independent from product development; in particular, we presume that test cases are *not* generated from the product models. It would be unreasonable to (automatically) derive test cases from the same models from which products are developed; such a procedure would test the derivation process rather than the product. Thus, we assume that test cases are designed separately from the models which are used for product development.

Given a product model and a test case, there are several possibilities:

– The test case describes a behaviour which is expected from this particular product (or product model, respectively). In this case we say that the colour of the test case with respect to the product model is *green*.

– The test case describes a behaviour which is not to be expected from this product model (but maybe from some other product of the product line). In this case, the colour of the test case w.r.t. the model is *red*.

– The product model is at an abstract level such that it can not yet be decided whether the implementation will exhibit the behaviour described by the test case or not. In other words, there are open design decisions such that one valid refinement shows the behaviour, whereas another one does not. In this case, we say that the test case is coloured *yellow*.

With a finished product, only green and red test cases are executed: Green test cases confirm that some desired functionality is present, whereas red test cases check that some undesired functionality is absent in the product.

In this paper, we formalise these notions and illustrate the ideas with the example of a product line of super-automatic espresso machines. This work is based on earlier work on specification-based testing for software product lines, where an algebraic / process algebraic modelling language was used[11]. Here, we transfer this approach to UML models which are materialisations of CVL models. This article is a significant extension of our short paper[12], with a fully worked-out example and a detailed implementation description.

The rest of this paper is organised as follows. We start by presenting our example product line in Sect. 2. Then, in Sect. 3, we give an overview of model-based design for software product-line engineering. Subsequently, in Sect. 4, we describe our approach to test case colouring. We describe tool support for this test case colouring procedure by model checking in Sect. 5. In Sect. 6 we elaborate the example and show the outcome of some simple test cases. We discuss related work in Sect. 7. Finally, in Sect. 8 we conclude and point at some future work.

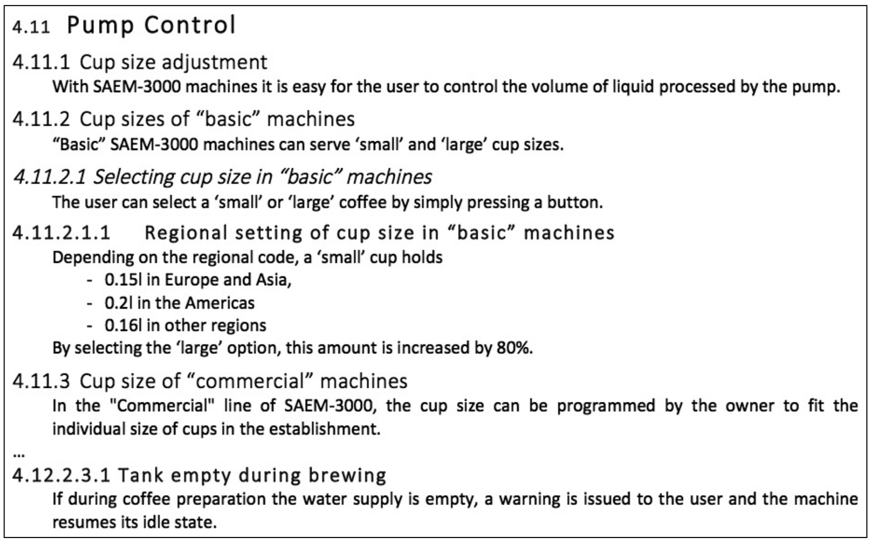
2 An Example Product Line

In this section, we present a product line of "super-automatic espresso machines". Although this example is hypothetic, it is modelled in the spirit of a real industrial example. There are various manufacturers of such machines, virtually all of which organise their portfolio as a product line. Customer prices are ranging from approximately one hundred to several thousand dollars, with dozens of different features to choose from.

Basic components of such a machine are the brew group with heating element(s), water pump, dispensing unit, water tank, bean hopper, grinder, and control unit. For each of these components there are several design variants. For example, the brew group can contain one or two heaters, where one heater is a boiler for brewing, and the optional second is a thermal block used to prepare steam in parallel to the brewing process. The grinder can be made of metallic or ceramic materials, and it can have a mechanical or automatic adjustment of grain size. The main purpose of the control unit is to set the volume of water which the pump presses into the brewing unit. In automatic espresso machines, this task is usually accomplished by an electronic control unit and appropriate software.

If a machine contains electronic components, additional functions can be realised. Similar to other domains, much of the user-visible innovation is implemented by software. In super-automatic espresso machines, more and more sophisticated computing hardware and sensors are integrated, which allow fine control over every aspect of the brewing process. Basic settings include the dosage of the beans, the grind setting or adjustment, and the volume in the cup. As a user interface, simple machines have a few fixed-purpose buttons and indicate their state by a number of LED lights and analog indicators. In contrast, advanced machines have a monochrome or full-colour LCD display, and user programmable capabilities for the button settings. For more information on features of automatic coffee machines, see Ref. [2].

In an industrial systems engineering process, both the system and the test cases for the system are derived from *system requirements*. For a product line, there are generic requirements which should be satisfied by each product, as well as special requirements concerning only certain features. Requirements are often written in natural language and managed in tools like IBM Rational DOORS® or Polarion® Requirements. Figure 1 gives some requirements for our example product line of super-automatic espresso machines.



4.11 **Pump Control**

4.11.1 Cup size adjustment

With SAEM-3000 machines it is easy for the user to control the volume of liquid processed by the pump.

4.11.2 Cup sizes of "basic" machines

"Basic" SAEM-3000 machines can serve 'small' and 'large' cup sizes.

4.11.2.1 Selecting cup size in "basic" machines

The user can select a 'small' or 'large' coffee by simply pressing a button.

4.11.2.1.1 Regional setting of cup size in "basic" machines

Depending on the regional code, a 'small' cup holds

- **0.15l in Europe and Asia,**
- **0.2l in the Americas**
- **0.16l in other regions**

By selecting the 'large' option, this amount is increased by 80%.

4.11.3 Cup size of "commercial" machines

In the "Commercial" line of SAEM-3000, the cup size can be programmed by the owner to fit the individual size of cups in the establishment.

…

4.12.2.3.1 Tank empty during brewing

If during coffee preparation the water supply is empty, a warning is issued to the user and the machine resumes its idle state.

Figure 1. Super-automatic espresso machines product line: Some requirements

From such requirements, test cases are developed (before, during, or after the system development). The test cases should reflect the user's needs and expectations, and should cover all of the requirements. For our example product line, typical testing objectives include:

TO-1 Upon pressing of the 'select/small/large' button, the machine will automatically brew coffee.

TO-2 The user presses the 'select/small/large' button, and the machine indicates the start of the brewing process with an appropriate message on the LCD screen and/or an appropriate pattern of the LEDs, turns on the heater, sets the grinding level, and turns on the grinder. It then advances to dispensing the coffee. There are two variants of this test case: If there is enough water in the tank, the machine pours the coffee; otherwise, it displays a warning and cancels the brewing process.

TO-3 The user selects a 'small' or 'large' coffee, and the amount of water dispensed is according to the regional setting. (This test is only applicable for machines of the "basic" group.)

TO-4 The user adjusts the water amount two steps by pressing 'down' – 'select' – 'up' – 'up' – 'select'; as an effect, the water amount during dispensing is increased by 20%. (This test is for all machines which feature programmable cup sizes.)

TO-5 The user adjusts the grinding level two steps by pressing 'down' – 'select' – 'up' – 'up' – 'select'; as an effect, the grinding level during grinding is increased by

20%. (This test is only for machines which have an adjustable grinder but no programmable cup size. Note that this is the same user input as before.)

TO-6 The heater is turned on only after the user selected a coffee; i.e., if the user presses any sequence of buttons except 'select', 'small' or 'large' and the heater is turned on, then this behaviour is forbidden.

The above list shows that some test cases are applicable to all products, while some are specific to products possessing certain features.

3 Model-based SPL Engineering

Product line development usually involves two engineering processes: domain engineering and application engineering. In domain engineering, reusable components are developed by a domain analysis and domain reference architectures. In application engineering, customer and market needs serve to generate different products by instantiating and composing generic artifacts from the domain engineering process. The instantiation and composition should be largely automatic. This way, many different products can be generated in an efficient way. There should be also feedback loops from application engineering to domain engineering, such that updates from different individual products can be generalised and adapted to the product line.

Model-based design is a particular form of software development, where a system model is continuously used as the central artifact throughout the whole engineering process. Initially, requirements are captured in an abstract model, e.g., in SysML, representing the system specification. This abstract model is refined and transformed into a concrete implementation model, e.g., in UML or Simulink®. From this implementation model, executable code is generated automatically by a suitable model compiler.

For model-based SPL engineering, the artifacts produced during domain engineering are mostly models. However, these models are generic, allowing an instantiation into different product models. During application engineering, the product models are refined and compiled as in "ordinary" model-based design.

Figure 2 depicts the model-based domain and application engineering work flows and their interrelations. The domain engineering process is generic, whereas the application engineering process exists in several instances, namely one for each product. Tasks are denoted by rectangles, and the models which are the results of tasks are given in circles.

We demonstrate the ideas of model-based SPL engineering by elaborating the example product line of super-automatic espresso machines discussed above, providing examples for all five types of models (feature model, resolution model, base model, variability model, and product model) involved. Languages involved are CVL, the common variability language, a recent attempt to define a syntactic framework supporting model based SPL engineering[7], and UML for providing the models.

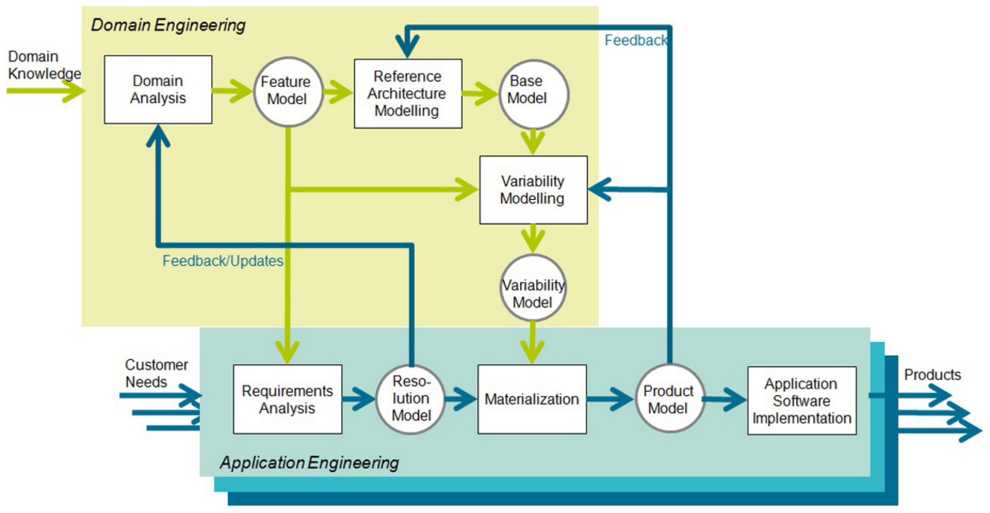


Figure 2. Various models in software product line engineering

3.1 Feature and resolution model

A *feature* (in CVL called a *VSpec*) is the description of a designated functionality. Each feature has a unique name and represents one characteristic of a product which is interesting for some stakeholder, e.g., a special added value for the customer. The *feature model* (in CVL called a *VSpec Tree*) is an explicit description of commonalities and differences of various products. Feature models are usually organised as and-or-trees, where each node is marked with the name of a feature. The root of the feature tree is the name of the product family. Sub-features of a feature may be marked as optional or mandatory. Additionally, it is allowed to attach boolean constraints on features to the feature tree. Tools for maintaining feature trees include pure::variants®, BigLever Gears®, and FeatureIDE[24].

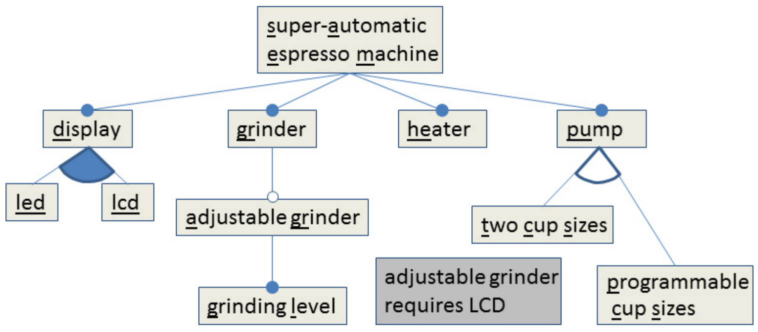


Figure 3. A feature model for super-automatic espresso machines

An example of a feature model is given in Fig. 3. This figure depicts some features of our above example of super-automatic espresso machines. In this feature model, it is determined that each machine has a display, a grinder, a heater, and a pump as mandatory features. The display can consist of a number of LED lights, or an LCD text display (or both). The pump can be adjusted to serve just the two cup sizes 'small' and 'large', or the cup size can be programmable by the user (but not both). An optional feature is the ability to adjust the grinder in order to influence the taste. If the feature is selected, there must be a variable *grindinglevel* which can be set; in this case an LCD is necessary to display the variable value.

For each feature model, one can construct an equivalent boolean formula with feature names as propositions, see, e.g., Ref. [1]. In the above example, this formula is

$$\begin{aligned} &(saem \Rightarrow di) \wedge (saem \Rightarrow gr) \wedge (saem \Rightarrow he) \wedge (saem \Rightarrow pu) \\ \wedge\ &(di \Rightarrow saem) \wedge (gr \Rightarrow saem) \wedge (he \Rightarrow saem) \wedge (pu \Rightarrow saem) \\ \wedge\ &(di \Rightarrow (led \vee lcd)) \wedge (led \Rightarrow di) \wedge (lcd \Rightarrow di) \\ \wedge\ &(agr \Rightarrow gr) \wedge (agr \Rightarrow gl) \wedge (gl \Rightarrow agr) \wedge (agr \Rightarrow lcd) \\ \wedge\ &(pu \Rightarrow (tcs \wedge \neg pcs \vee pcs \wedge \neg tcs)) \wedge (tcs \Rightarrow pu) \wedge (pcs \Rightarrow pu) \end{aligned}$$

Since our feature model involves ten features and different parameters, there is quite a large number different product models. Only a few of these will be materialised as actual products in the market.

Given any feature model, a *resolution model* (or simply *resolution*) is an assignment of truth values to feature names, such that the corresponding boolean formula evaluates to true.[1] In our example, two possible resolutions are given in Fig. 4.

$$\boxed{\{saem, di, gr, he, pu, led, tcs\} \mapsto true, \quad \{lcd, agr, gl, pcs\} \mapsto false} \tag{1}$$

$$\boxed{\{saem, di, gr, he, pu, led, lcd, agr, gl, pcs\} \mapsto true, \quad \{tcs\} \mapsto false} \tag{2}$$

Figure 4. Two resolution models for the super-automatic espresso machines feature model

3.2 Base model

A *base model* is an artifact realising the features of the product line. (The base model model is sometimes called "the 150% model", a terminology which we refrain to adopt.) Formally, a base model can be any model which is an instance of some MOF meta-model. In our work, a base model is a UML model consisting of (restricted) state machine diagrams, class diagrams and (restricted) OCL formulae. The base model describes realisations for *all* features; thus, if the feature model contains conflicting features, then the base model does not represent a possible product.

Part of the base model for our example product line is given in Fig. 5 (static structure), Fig. 6 (user interface), and Fig. 7 (control component). Subsequently, we explain some of the elements relevant for the variability in the product line.

[1]In propositional logic, such a truth assignment is sometimes called a *model* of the formula; we use the CVL terminology here in order to avoid misunderstandings.

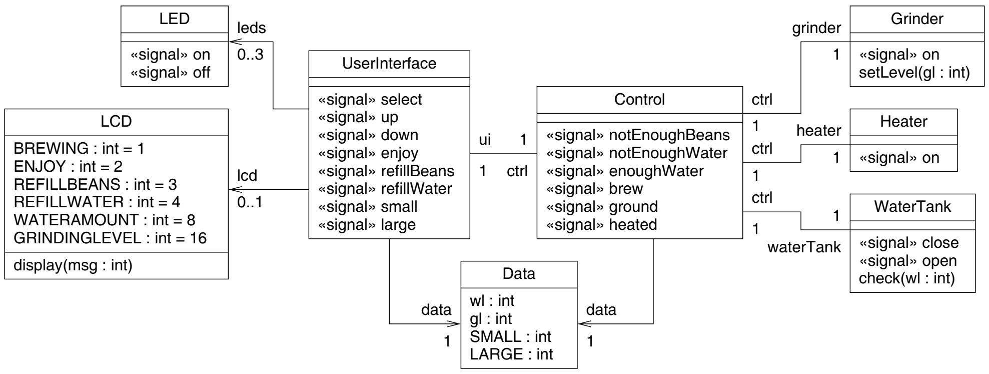


Figure 5. Super-automatic espresso machines product line: Static structure

The UserInterface class in the static structure is associated with up to three LEDs and at most one LCD display. Class Data provides a variable wl for the amount of water, which is used for brewing one cup. In simple machines, it can be set to the two constants SMALL and LARGE for the two cup sizes; in more expensive machines, wl can be adjusted by the user within certain limits. Class Data also provides a variable gl for the grinding level, which is necessary for machines with an adjustable grinder.

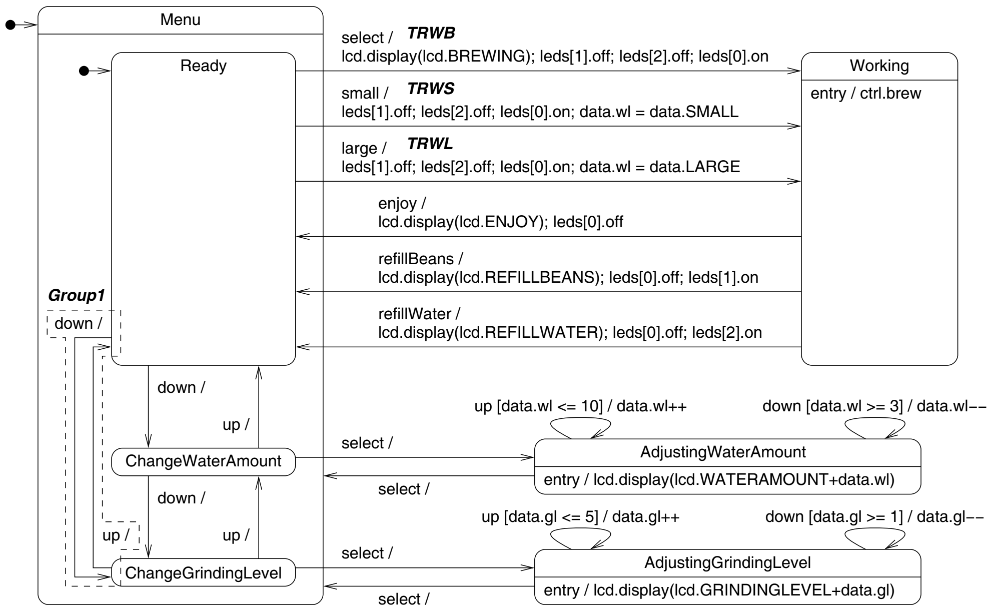

Menu
Ready
select / TRWB
lcd.display(lcd.BREWING); leds[1].off; leds[2].off; leds[0].on
small / TRWS
leds[1].off; leds[2].off; leds[0].on; data.wl = data.SMALL
large / TRWL
leds[1].off; leds[2].off; leds[0].on; data.wl = data.LARGE
enjoy /
lcd.display(lcd.ENJOY); leds[0].off
refillBeans /
lcd.display(lcd.REFILLBEANS); leds[0].off; leds[1].on
refillWater /
lcd.display(lcd.REFILLWATER); leds[0].off; leds[2].on
Working
entry / ctrl.brew
Group1
down /
down /
up /
ChangeWaterAmount
down /
up /
up /
ChangeGrindingLevel
select /
select /
up [data.wl <= 10] / data.wl++
down [data.wl >= 3] / data.wl−−
AdjustingWaterAmount
entry / lcd.display(lcd.WATERAMOUNT+data.wl)
select /
select /
up [data.gl <= 5] / data.gl++
down [data.gl >= 1] / data.gl−−
AdjustingGrindingLevel
entry / lcd.display(lcd.GRINDINGLEVEL+data.gl)

Figure 6. Super-automatic espresso machines product line: User interface

The state machine diagram in Fig. 6 describes the user interface of the espresso machines. Basically, there are two states: Ready and Working. TRWB, TRWS, and TRWL are transitions from Ready to Working; they are triggered by pressing the select button, and, if present, the small or large button. When the brewing is finished, the machine returns to the ready state with an appropriate message. With some machines, there is the possibility to use the down and up buttons to access menus for adjusting the amount of water for a cup and the grinding level. Note that there are several possible transitions from Ready triggered by down; the variability model resolves this apparent non-determinism by selecting the appropriate ones according to the features materialised in the product model.

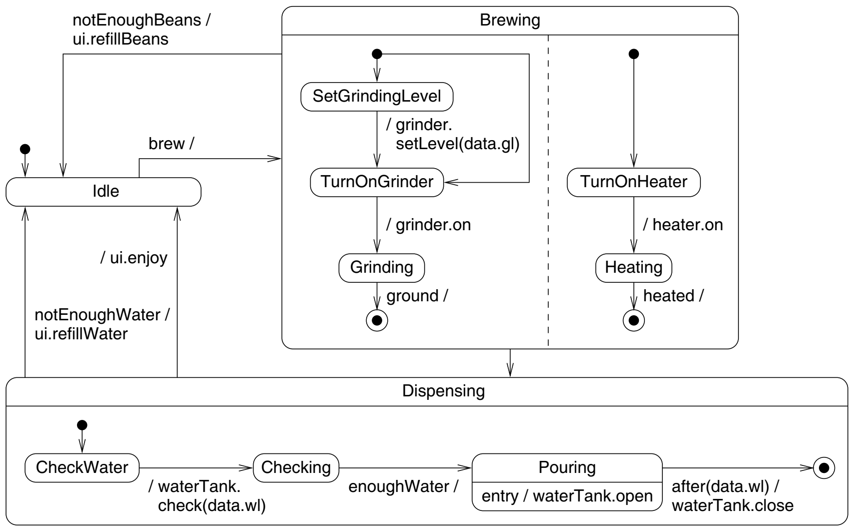


Figure 7. Super-automatic espresso machines product line: Control component

The diagram in Fig. 7 describes the internal control structure of the espresso machines. Basically, there are three control states: Idle, Brewing and Dispensing. Within Brewing, there are two parallel regions, for grinding and heating. UML does not put any restrictions on the interleaving of transitions between these two regions; this non-determinism is resolved by the programmer, code generator or runtime system. For the filling level of a cup, this state machine does not contain any variability; the amount of water is determined by the variable wl, defined in the class diagram in Fig. 5 and set by the user interface in Fig. 6.

3.3 Variability model

A *variability model* is a feature tree with variation points linking into the base model. In CVL, there are several kinds of variation points: object existence, variable assignment, object substitution, and others.

For our example product line, a variability model is depicted in Fig. 8. It states that the class LED from the base model is present in a product model if and only if the feature *led* is true in the resolution of the feature model. Likewise, if the feature *lcd* is true in a resolution, then the class LCD is present in the resolved product model, otherwise it is absent.

The feature *adjustable grinder* determines that the variable gl (for the grinding level) in class Data and the method setLevel in class Grinder are present, as well as the states ChangeGrindingLevel in the Menu state of UserInterface and SetGrindingLevel in the Brewing state of Control. Note that CVL leaves it specific to the tool and/or the base language what cascading effects the removal of a given object has. In our case, all transitions entering and leaving these two states are also left out when resolving a product model where *adjustable grinder* is false. The feature *adjustable grinder* has as mandatory subfeature, the *grindinglevel*, which initializes the respective variable in the resolution.

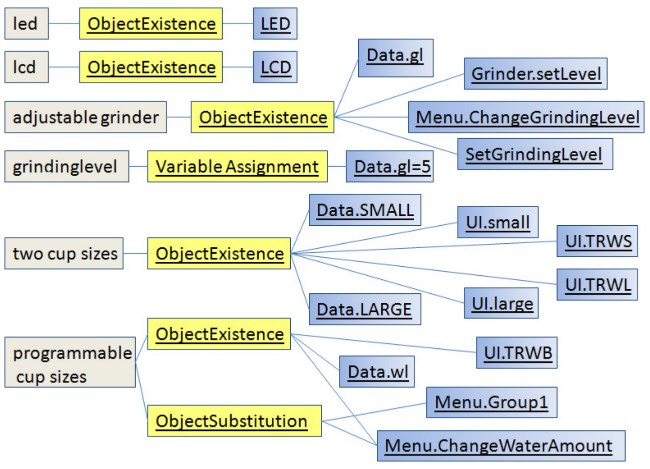


Figure 8. The variability model for the product line

Features *two cup sizes* and *programmable cup sizes* are mutually exclusive; resolving *two cup sizes* to true gives the constants SMALL and LARGE as well as the signals small and large which trigger transitions from Ready to Working, whereas resolving *programmable cup sizes* to true gives the variable wl and the transition via select. Additionally, if *programmable cup sizes* is set to true, then the transitions down and up from Ready to ChangeGrindingLevel in the Menu state (shown in bold) must be replaced by transitions leading in and out of the state ChangeWaterAmount.

3.4 Product model

From the variability model for a given base model, for each resolution a *product model* can be generated. In CVL, this process is called *materialisation*. It is done by applying the variation points according to a given resolution. This means deleting model elements which are bound to an existence variation point, assigning a value to a variable bound to a value-assignment variation point, etc. As an example, in Fig. 9 (part of) the product model for the first resolution from Fig. 4 is given. This product model describes a low-end machine with only two buttons (‘small’ and ‘large’), and three LEDs (indexed by 0 to 2), which signal brewing, empty bean hopper, and empty water tank, respectively.

The product models resulting from materialisations are plain UML models with the usual semantics. In a model-based development process, product models are further refined to the software for different products.

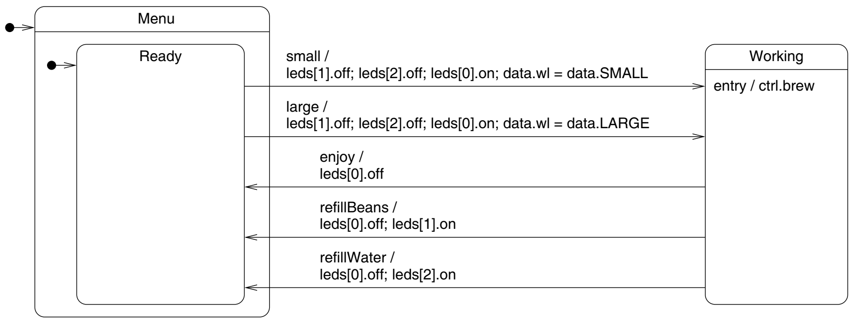


Figure 9. Super-automatic espresso machines product line: User interface for resolution (1)

4 Colouring Product Line Test Cases

In this section, we show how the product model can be used to determine the expected outcome of a test case to a product. Our contribution is inspired by our previous work[11] in the context of the specification language Csp-Casl. The definition presented here differs as we wanted to cater better for the mechanics of the UML language. While in the context of Csp-Casl we worked with refusals in order to deal with internal non-determinism, for testing from UML models we prefer to speak about enabled and obliged events. Furthermore, the developed technology is automatised to a higher degree, as it is based upon model checking (rather than interactive theorem proving).

We use a three-valued colouring scheme to capture what design decisions have been made in the product model with regards to the product: A test case is coloured green if it reflects a behaviour that is expected from this particular variant of a SPL. It is coloured red if the variant should not allow the described behaviour. Finally, a test case is coloured yellow if the respective behaviour is neither required nor disallowed by the specification of the variant. This can happen, e.g., if the specification is non-deterministic or incomplete.

Intuitively, green test cases reflect *required* and red test cases *forbidden* behavioural properties of the specification. Yellow tests mirror *open design decisions*, i.e., properties which are not (yet) decided in the specification. Since the colour of a test case depends on the base model as well as the variability model and its resolution for a particular variant, the same test case can be green for one product, but red or yellow for another one.

In order to make these notions more precise, we briefly recall the UML stipulations on the execution of a model: In UML state machines, a transition $e[g]/a$ may have a *trigger* e, can be restricted by a *constraint* g, and can invoke a *behaviour* a. The UML superstructure explains: "A trigger specifies an event that may cause the execution of an associated behaviour. An event is often ultimately caused by the execution of an action, but need not be. [...] Upon their occurrence, events are placed into the input pool of the object where they occurred [...]. An event is dispatched when it is taken from the input pool and is processed by the classifier. At this point, the event is considered consumed and referred to as the current event."[19, p. 471sq.]. The constraint language is not specified in UML; "a constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element"[19, p. 57]. A behaviour is a consequence of the execution of an action by some related object. The behaviour invoked as the effect of a transition may contain several actions, e.g., calling an operation, changing variable values, or causing the occurrence of some event.

To define the notion of a test case, we fix a *test signature* Σ. In our approach, we assume that Σ is a subset of the occurrences and dispatches of events which are contained in the product model. In this case, we say that the test case is applicable to the product model.

Additionally, we require that stimuli can be sent to the SUT, for example pressing the button small, from the outside. We represent this as the artificial entity *tester*. Intuitively, elements of the signature are the only events which can be "noticed" by the test case; events not in the signature are "invisible". A *test case* is a finite sequence of elements from the test signature Σ.

Example 1. Consider the product model for resolution (1), see Fig. 9 for its user interface and Fig. 10 for its initial configuration.

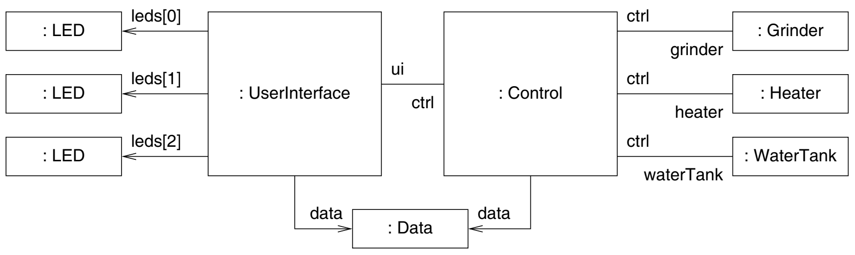


Figure 10. Super-automatic espresso machines product line: Initial configuration for resolution (1)

For testing an implementation against this product model, one could for instance choose the signature

$$\begin{aligned}\Sigma_{small} = \{&\mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(tester, \mathsf{small}, ui),\\ &\mathsf{occ}(ui, \mathsf{on}, leds[0]), \mathsf{occ}(ui, \mathsf{brew}, ctrl),\\ &\mathsf{disp}(ctrl, \mathsf{enjoy}, ui), \mathsf{disp}(ctrl, \mathsf{refillBeans}, ui)\}\ .\end{aligned}$$

Here, occ and disp indicate event occurrences and event dispatches, respectively. Event occurrences and dispatches have a sender and a receiver. For instance, $\mathsf{occ}(tester, \mathsf{small}, ui)$ stands for the occurrence of small, sent by the tester to the user interface object ui, and $\mathsf{disp}(tester, \mathsf{small}, ui)$ stands for the dispatch of this event by the interface object ui.

Σ_{small} allows, e.g., to check if the choice of a small cup leads to turning on an LED as well as starting the brewing process. This expectation can be expressed by the test case

$$\begin{aligned}T_1 = \langle&\mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(tester, \mathsf{small}, ui),\\ &\mathsf{occ}(ui, \mathsf{on}, leds[0]), \mathsf{occ}(ui, \mathsf{brew}, ctrl)\rangle\ .\end{aligned}$$

In order to fix the colour of a test case, we assume that there exists a function *enabled* assigning to each configuration of a UML model the set of elements from Σ which may occur next. That is, an event $e \in \Sigma$ is in $enabled(c)$, if upon its occurrence there is a sequence $c_0 \xrightarrow{e_1} c_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} c_n$ of transitions such that $c_0 = c$ and $e_n = e$, and for all $i < n$ it holds that $e_i \notin \Sigma$. In this case, we say that c_n is *reached* from c by e. For an event $e \in enabled(c)$, we say that it is *obliged* at c, if it is not the case that some $e' \in \Sigma$ different from e is enabled in c. Intuitively, if e is obliged at c, it is the event from Σ which must occur next, if any.

Since UML contains semantic variation points, the function *enabled* is tool-dependent. In particular, UML does not impose an ordering on events in the event pool; furthermore, the mechanism for determining the behaviour to be invoked as a result of a call operation is unspecified, and it is a semantic variation point whether one or more behaviours are triggered when an event satisfies multiple triggers. The UML allows an event to be dispatched in a configuration even if there is no transition taking this event as a trigger; in such a situation, this event is discarded.

The *colour* of a test case $T = \langle e_1, \ldots, e_n \rangle$ in the signature Σ with respect to a product model is a value from $\{green, red, yellow\}$, such that

– $colour(T) = green$ iff for all $k < n$ and every sequence $\langle c_0, c_1, \ldots, c_k \rangle$ of configurations such that c_0 is an initial configuration, and c_i is reached from c_{i-1} by e_i for all $1 \leqslant i \leqslant k$ it holds that e_{k+1} is obliged at c_k;

– $colour(T) = red$ if there is no sequence $\langle c_0, c_1, \ldots, c_n \rangle$ of configurations such that c_0 is an initial configuration, and c_i is reached from c_{i-1} by e_i for all $1 \leqslant i \leqslant n$; and

– $colour(T) = yellow$, otherwise.

In other words, a test case is green if it can be observed in all possible executions of the model triggered by this test case. It is red if there is no possible execution where it can be observed. It is yellow if some executions show the behaviour and others do not.

Note that our definition enforces that for each test case $T = \langle e_1, \ldots, e_n \rangle$ for which $colour(T) = green$ there is at least one sequence $\langle c_0, c_1, \ldots, c_n \rangle$ such that c_0 is an initial configuration, and for all $1 \leqslant i \leqslant n$, configuration c_i is reached from c_{i-1} by e_i. That is, green test cases must indeed be observable in the system's executions.

Example 2. Consider the product model for resolution (1). We claim that test case T_1 w.r.t. signature Σ_{small}, see Ex. 1, has the colour *green*.

Initially, the tester presses the button small. The user interface in Fig. 9 shows the transition TRWS labelled small / leds[1].off; leds[2].off; leds[0].on; data.wl = data.SMALL. This mirrors the dispatch of small and the occurrence of on in test case T_1. The events for turning off LED 1 and LED 2 as well as setting the water level to SMALL are ignored as they are not part of Σ_{small}. Upon entry of state Working, brew occurs.

There are several configuration sequences matching this test case. For instance, the order in which LED 1 and LED 2 dispatch off has not been specified. In a concurrent system, they may happen in any order. However, for all these sequences it holds that if the next event is from the test signature, then it is the only one from the test signature, i.e., the event is obliged. For instance, in an initial configuration, no event can be dispatched as all event queues are empty. All coffee machine components start with an event dispatch, thus the only component able to act is the tester. The tester starts by choosing a small coffee.

An important safety property of espresso machines is that they never start brewing coffee without tester interaction, see test objective TO-6 from Sect. 2. A test case for this property should obtain the colour *red*.

Example 3. To encode this test objective for the product model for resolution (1), we define the signature

$$\Sigma_{noSelfActivation} = \{\mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{occ}(tester, \mathsf{large}, ui), \mathsf{occ}(ui, \mathsf{brew}, ctrl)\}\ .$$

We can observe the possible tester inputs to the machine and the start of the brewing process. We claim that with this signature the test case

$$T_2 = \langle \mathsf{occ}(ui, \mathsf{brew}, ctrl)\rangle$$

obtains the colour *red*. We argue again that in the initial configuration, no event can be dispatched as all event queues are empty, all coffee machine components start with an event dispatch, and in T_2 the tester does not act.

Sometimes, the colouring of test cases might yield "surprising results". One would hope, for instance, that choosing a cup of coffee always results in obtaining a coffee. However, as daily experience tells, this is not always the case: there might not be enough coffee beans, or the water tank might be empty. Such possible behaviour, which however won't necessarily happen, should be coloured *yellow*:

Example 4. To encode the test objective "choosing a cup of coffee results in obtaining a coffee" for the product model for resolution (1), we define the signature

$$\Sigma_{dailyExperience} = \{\mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(ctrl, \mathsf{enjoy}, ui)\}\ .$$

We can observe one possible tester input and the controller signalling that the brewing process was successful. We claim that with this signature test case

$$T_3 = \langle \mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(ctrl, \mathsf{enjoy}, ui)\rangle$$

obtains the colour *yellow*.

While we actually find configuration sequences in which small and disp are enabled (thus, this test case is not red), it is not the case that after reaching a configuration with small we will reach a configuration with disp(ctrl, enjoy, ui). For instance, after taking the transition labelled with brew / in the control component, see Fig. 7, it is possible to follow the transition labelled with notEnoughBeans / ui.refillBeans and the event $\mathsf{occ}(ctrl, \mathsf{enjoy}, ui)$ will never happen (thus, this test case is not green).

Here are some simple properties of our colouring.

- An empty test case (consisting of no events at all) is always green.
- A one-element test case is green if its event is enabled and obliged in all initial configurations; it is red, if the event is initially not enabled; and yellow, if it is enabled in some initial configuration but not obliged.

– Any initial fragment of a green test case is green; any extension of a red test case is red.

– If a state is non-deterministic, e.g., from state s there are transitions $/a$ and $/b$, then the test cases $\langle a\rangle$ and $\langle b\rangle$ are yellow, since $\mathit{enabled}(s) = \{a, b\}$, but a is not obliged at s. Assuming that the test signature is $\{a, b, c\}$, the test case $\langle c\rangle$ is red, since neither $/a$ nor $/b$ produce c and thus c is not enabled in s.

– Consider a situation where the effect of a transition invokes a behaviour expression including an operation for which only its signature is known (e.g., a transition $/obj.op(arg)$, where the operation op is declared in the class diagram, but the return value of op for a given argument arg is not specified). Then test cases using such a transition will be yellow, as all possible return values are enabled in the state machine; however, the test case contains only a specific one.

The test verdict (pass or fail) for a test is assigned by executing a green or red test case with a concrete product. A product passes a test suite, if it behaves as expected, i.e., if it exhibits the behaviour described in all green test cases and deviates from the behaviour described in all red test cases. Yellow test cases do not contribute to the detection of faults, thus we do not execute them.

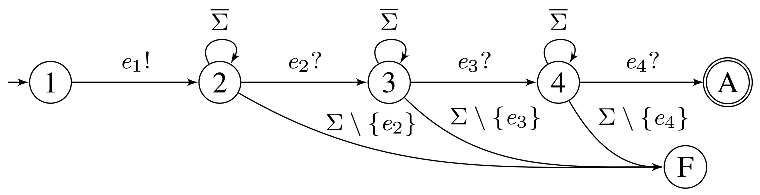


Figure 11. Structure of testing automata

5 Automated Test Colouring via Model Checking

For automating the above defined test colouring procedure for a given materialisation of a SPL and a test case, we use the tool Hugo/RT, which is a UML model translator for model checking[13]. In particular, Hugo/RT resolves the UML's semantic variation points mentioned above in a particular way thus also fixing the *enabled* function: The event pool is implemented as bounded event queues; since inheritance is not supported, the dispatching algorithm becomes straightforward, as no overloading has to be considered; only a single, non-deterministically chosen behaviour can be triggered by a given event; and events which trigger no outgoing transition in a state configuration are silently consumed.

Hugo/RT translates both the materialisation and a test case over a test signature into Promela, which is the input language of the model checker SPIN[10]; syntactically, it is first ensured that the test signature indeed is a subset of the possible event occurrences and dispatches of the materialisation. The resulting encoded product model shows instrumentation for observing all events: Whenever an event occurs or is dispatched in the product model, a notification is sent out which can be used by an observer. The test case results in an automaton process sending those events to the system which occur at the *tester* and also reacting to those produced by the system: If an event of the test case is observed, the test case automaton proceeds; if any other event which is part of the test signature happens, the automaton goes to a dedicated *failure state*; events not present in the test signature are ignored. After successful observation of the last entry of the test case sequence the automaton enters a dedicated *final state*.

The structure of such a testing automaton is shown in Fig. 11 using the test sequence $\langle e_1, e_2, e_3, e_4 \rangle$ over the test signature Σ as an example, where e_1 originates with the artificial *tester* (indicated by a !) and e_2, e_3, and e_4 are produced by the system product model (indicated by a ?) and thus have to be observed. $\overline{\Sigma}$ denotes the unobservable events outside the test signature. The accepting final state is shown as a double-outlined circle labelled A, the failure state as a circled F.

Using SPIN, we now check on the one hand whether the testing automaton can proceed to its final state which is reached when the last event of the test case has happened. If this final state cannot be reached, the test case is coloured red. On the other hand, if the final state is reachable, we additionally check whether the dedicated failure state is reachable. If the failure state cannot be reached, the test case is coloured green, otherwise yellow.

Technically, the testing automaton is notified by the system, i.e., the encoded product model instrumented for observation, of event occurrences and event dispatches by

```
observer!SEND,sender,receiver,behavioral,arguments
observer!RECEIVE,sender,receiver,behavioral,arguments
```

where *behavioral* denotes either a signal or an operation. The testing automaton raises an event occurrence by

```
event_queues[receiver]!signal/operation,empty,arguments
```

where `empty` represents an unknown sender, i.e., the testing automaton as originator. The failure state is implemented as an `assert(false)`, since assertion violations can be checked conveniently in SPIN. The final state is represented by a cycle through a statement labelled `acceptAll` because SPIN offers special support for checking for such "acceptance cycles". However, SPIN does not offer support for real-time (as used for exiting Pouring in Fig. 7); HUGO/RT also provides a translation of UML state machine systems into the real-time model checker UPPAAL[25], but we currently have not included a translation of the test cases to UPPAAL.

Example 5. Let us consider the test case

$$T_1 = \langle \mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(tester, \mathsf{small}, ui), \\ \mathsf{occ}(ui, \mathsf{on}, leds[0]), \mathsf{occ}(ui, \mathsf{brew}, ctrl) \rangle$$

for the test signature Σ_{small} of Ex. 1. This test case structurally conforms to the example shown in Fig. 11.

The Promela-representation of this test case produced by HUGO/RT takes the following form:

```
proctype Tester() {
  bit direction; byte sender; int behavioral; byte receiver;
  int arguments[1];
  nc_t_1:
```

```
 5  event_queues[obj_ui-1]!send_small,empty,empty;
    nc_t_2:
    observer?direction,sender,receiver,behavioral,arguments[0];
    if
    :: behavioral == send_off -> goto nc_t_2
10  :: ... /* analogously for all other events that are not considered */
    :: direction == RECEIVE && sender == empty && behavioral == send_small
         && receiver == obj_ui -> goto nc_t_3
    :: else -> assert(false)
    fi;
    nc_t_3:
15  observer?direction,sender,receiver,behavioral,arguments[0];
    if
    :: behavioral == send_off -> goto nc_t_3
    :: ... /* analogously for all other events that are not considered */
    :: direction == SEND && sender == obj_I && behavioral == send_on &&
         receiver == obj_led0 -> goto nc_t_4
20  :: else -> assert(false)
    fi;
    nc_t_4:
    observer?direction,sender,receiver,behavioral,arguments[0];
    if
25  :: behavioral == send_off -> goto nc_t_4
    :: ... /* analogously for all other events that are not considered */
    :: direction == SEND && sender == obj_ui && behavioral == send_brew &&
         receiver == obj_ctrl -> goto acceptAll
    :: else -> assert(false)
    fi;
30  acceptAll:
    if
    :: 0 == 0 -> goto acceptAll
    fi
  }
```

In each step, delimited by the labels `nc_t_`x, one event is either generated or checked for occurrence; for instance, in `nc_t_1` the sending of small from *tester* to *ui* is generated. All messages which are not under consideration are ignored; see, e.g., lines 9 and 10. If a message occurs which is neither ignored nor is the event we are waiting for, an assertion violation is raised. Finally, in `acceptAll`, we have passed through all desired events successfully, and an infinite loop is entered which can be detected by using SPIN's check for "acceptance cycles" (indicated by the label's name `acceptAll` starting with `accept`).

First, SPIN reports that an acceptance cycle is reachable; this check is instantaneous on a Intel® Core 2 Quad CPU with 2.33 GHz and 4 GB RAM. Then SPIN reports that no assertion violation can be reached, this time taking 3.15 s. Thus, this test case is indeed coloured green.

6 Colouring Example Test Cases

We now discuss the colouring of some test cases w.r.t. to different materialisations of the variability model. We first give a universal test suite, discuss the syntactic applicability of its test cases to product models, and finally present their colouring.

6.1 A universal test suite

A universal test suite addressing the six testing testing objectives TO-1 to TO-6 stated in Sect. 2 could consist of the following test cases:

Concerning **TO-1**:

$$\begin{aligned}
\text{TC-1}_{small} &= \langle \mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(tester, \mathsf{small}, ui), \mathsf{occ}(ui, \mathsf{brew}, ctrl)\rangle \\
\text{TC-1}_{large} &= \langle \mathsf{occ}(tester, \mathsf{large}, ui), \mathsf{disp}(tester, \mathsf{large}, ui), \mathsf{occ}(ui, \mathsf{brew}, ctrl)\rangle \\
\text{TC-1}_{select} &= \langle \mathsf{occ}(tester, \mathsf{select}, ui), \mathsf{disp}(tester, \mathsf{select}, ui), \mathsf{occ}(ui, \mathsf{brew}, ctrl)\rangle
\end{aligned}$$

Concerning **TO-2**:

$$\begin{aligned}
\text{TC-2}_{enoughWater} = \langle &\mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(tester, \mathsf{small}, ui), \mathsf{occ}(ui, \mathsf{off}, leds[1]), \\
&\mathsf{occ}(ui, \mathsf{off}, leds[2]), \mathsf{occ}(ui, \mathsf{on}, leds[0]), \mathsf{occ}(ctrl, \mathsf{on}, heater), \\
&\mathsf{occ}(ctrl, \mathsf{setLevel(data.gl)}, grinder), \mathsf{occ}(ctrl, \mathsf{on}, grinder), \\
&\mathsf{disp}(ctrl, \mathsf{enoughwater}, waterTank), \mathsf{occ}(ctrl, \mathsf{enjoy}, ui)\rangle \\
\text{TC-2}_{notEnoughWater} = \langle &\mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{disp}(tester, \mathsf{small}, ui), \mathsf{occ}(ui, \mathsf{off}, leds[1]), \\
&\mathsf{occ}(ui, \mathsf{off}, leds[2]), \mathsf{occ}(ui, \mathsf{on}, leds[0]), \mathsf{occ}(ctrl, \mathsf{on}, heater), \\
&\mathsf{occ}(ctrl, \mathsf{setLevel(data.gl)}, grinder), \mathsf{occ}(ctrl, \mathsf{on}, grinder), \\
&\mathsf{disp}(ctrl, \mathsf{notEnoughWater}, waterTank), \mathsf{occ}(ctrl, \mathsf{refillWater}, ui)\rangle
\end{aligned}$$

Concerning **TO-3**:

$$\begin{aligned}
\text{TC-3}_{small} &= \langle \mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{occ}(ctrl, \mathsf{check(15)}, waterTank)\rangle \\
\text{TC-3}_{large} &= \langle \mathsf{occ}(tester, \mathsf{large}, ui), \mathsf{occ}(ctrl, \mathsf{check(27)}, waterTank)\rangle
\end{aligned}$$

Concerning **TO-4**:

$$\begin{aligned}
\text{TC-4} = \langle &\mathsf{occ}(tester, \mathsf{down}, ui), \mathsf{occ}(tester, \mathsf{select}, ui), \mathsf{occ}(tester, \mathsf{up}, ui), \\
&\mathsf{occ}(tester, \mathsf{up}, ui), \mathsf{occ}(tester, \mathsf{select}, ui), \mathsf{occ}(waterTank, \mathsf{check(7)}, ctrl)\rangle
\end{aligned}$$

Concerning **TO-5**:

$$\begin{aligned}
\text{TC-5} = \langle &\mathsf{occ}(tester, \mathsf{down}, ui), \mathsf{occ}(tester, \mathsf{select}, ui), \mathsf{occ}(tester, \mathsf{up}, ui), \\
&\mathsf{occ}(tester, \mathsf{up}, ui), \mathsf{occ}(tester, \mathsf{select}, ui), \mathsf{occ}(ctrl, \mathsf{setLevel(7)}, grinder)\rangle
\end{aligned}$$

Concerning **TO-6**:

$$\text{TC-6} = T_2 = \langle \mathsf{occ}(ui, \mathsf{brew}, ctrl)\rangle$$

w.r.t.
$\Sigma_{noSelfActivation} = \{\mathsf{occ}(tester, \mathsf{small}, ui), \mathsf{occ}(tester, \mathsf{large}, ui), \mathsf{occ}(ui, \mathsf{brew}, ctrl)\}$.

In TC-1 to TC-5, the test signature is identical to the events in the test cases. Naturally, more test cases can be added to the universal test suite in order to realise the testing objectives.

6.2 Applicability of test cases to product models

A test case is applicable to a product model if its test signature – up to the stimuli sent by the tester – are occurrences and dispatches of events contained in the model. Thus, we obtain for the product model for resolution (1) that TC-1_{small} and TC-1_{large} are applicable, while TC-1_{select} is not applicable. For the product model for resolution (2), the result is the opposite: TC-1_{small} and TC-1_{large} are not applicable, while TC-1_{select} is applicable. In the following, we will discuss the colouring of applicable test cases only.

6.3 Colouring of test cases

In the following we discuss for each of the above test cases how their colour depends on the selected features and other properties of the product model.

Required behaviour: If applicable to a product model, TC-1_{small}, TC-1_{large}, and TC-1_{select} are coloured green. Example 2 provides an argument that can easily be adjusted to suit these test cases.

Non-determinism in a model: The test cases TC-$2_{enoughWater}$ and TC-$2_{notEnoughWater}$ are coloured yellow in all applicable product models. The reason is that the base model contains some elements of abstraction, leaving room for different implementations. In the example of our test cases, the interleaving of different actions in the two parallel regions of state *Brewing* in the Control Component, see Fig. 7, is not fixed; this is determined, e.g., by the programmer or automated code generator. Should the model be refined to a deterministic one such that, e.g., turning on the heater happens before adjusting the grinding level and turning on the grinder, and additionally the signals enoughWater, notEnoughWater are made predictable in the model, then these test cases should become green or red. This shows that refinement can turn a yellow test case into a red or a green one.

Underspecification of data: TC-3_{small} and TC-3_{large} are both yellow, as the regional value settings of Data.SMALL and Data.LARGE are not defined in the UML model. In a resolution where these constants are assigned to the values 15 and 27, respectively, these test cases are green.

Colour of test case depends on feature selection: TC-4 and TC-5 are applicable only if a select button is available. Similar to the previous discussion of underspecification of data, TC-4 is coloured yellow, since data.wl is not initialised. If the feature adjustable grinder is present, the value of data.gl is initialised with the value 5. Assuming data.wl has the value 5 initially, we can see the following effect w.r.t. different resolutions:

– TC-4 is green in case the feature programmable cup-size is present, however, it is red in case this feature is absent.

– TC-5 is green in case the feature adjustable grinder is present and the feature programmable cup-size is absent, however it is red if both these features are present.

Safety: TC-6 illustrates how to encode a safety property as a negative test case. It has been discussed in Ex. 3.

7 Related Work

Testing is an important topic in the software product line literature. Systematic reviews can be found in Refs. [9,16,14]. These surveys show that most of the work on SPL testing is concerned with the question of selecting products for testing. That is, the authors want to identify a representative set P of products such that the quality of the base model can be assured by testing (only) the products in P. Strategies include selecting products with minimal and maximal features, pair-wise testing, incremental or regression-based SPL testing, etc. In particular, Mota deals with maintaining correctness of the base model after modifications[18]. Baller et al. focus on heuristics for minimization of the test suite for the base model[4].

Also the problem of re-using SPL test cases for the testing of different products has been considered before. In Ref. [23] the authors suggest to tackle the problem by "preserving variability throughout generic test artifacts in domain engineering, and by reusing these generic test artifacts in application engineering to derive product-specific test case scenarios". This approach requires that the development of test cases is lifted from the application level, where it is common practice, to the domain engineering level. Furthermore, we see the test development as an independent competitive process, which leads to test suites out of which test cases are selected according to the specific product needs.

Oster[20] uses a combinatorial strategy for combining features to form a representative set of products. Test cases are then generated automatically from a reusable test model. The main focus of this approach is on the selection of resolution models such that the selected set of product models gives a feasible survey of the product line. For our approach, we are not concerned with the modelling of features and resolutions. However, the representative set of products could serve as a basis for an initial colouring of test cases.

In Ref. [15], the authors propose to construct test artifacts incrementally for every product variant by explicitly considering commonality and variability between two consecutive products under test. This approach is closely related to our work; however, we use a three-valued test evaluation scheme. Moreover, their paper uses a dedicated test model, whereas in our work test cases are evaluated with the base model and variability models.

Bertillon et al.[3] use a notation based on natural language descriptions of requirements to define test cases for product lines. The resulting test specification is generic in the product, and a set of relevant test scenarios for a customer specific application can be derived from it. This work complements our colouring method, since we assume that the test suite is designed separately.

8 Conclusions and Future Work

We have presented a theory and prototypical implementation for test case assessment in the model-based development of multi-variant systems. To our knowledge, this is the first treatment of the subject in the context of UML-based software development.

We deal with both positive (green) and negative (red) test cases, and introduce a third colour (yellow) for test cases whose outcome is not determined with a given product model. This means that it is needless to execute them with products based on this model. Our approach thus allows to assess and select those test cases from a universal test suite which are relevant for a given product. It would be a straightforward extension to define the notion of a "partial resolution" of a base model which yields a *set* of product models as materialisation. Additionally, lifting our approach to logical, abstract test specifications in the universal test suite would be of interest. This would have to include different colourings for the different concretisations. For conciseness, we did not pursue these extensions further.

Our theory is well-suited for testing deterministic reactive systems under test, where the response functionally depends on the provided stimuli. In the UML specification, it can deal with indeterminacy caused by semantic variation points and nondeterminism by under-specification and open design decisions, by assigning the respective test cases the colour yellow. The theory excludes to formulate test cases for systems which are inherently non-deterministic. This can be the case, e.g., for a network of cooperating devices with unpredictable message delays. To deal with such a situation, we are investigating trees and UML interactions as test cases and the relation to the testing theory of de Nicola and Hennessy[8].

Our future plans include to apply the theory to actual industrial problems in safety-critical systems. We are looking at case studies of train control systems and flexible automation modules for engine test beds. To this end, we have to extend our current prototypical implementation such that all steps are fully automatic. Furthermore, all steps in the tool chain, including the model transformation from UML into Promela, need to be certifiable. Therefore, we are looking at verification techniques for model transformation tools in order to allow the use of UML also in safety-critical systems development.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful suggestions and comments. Furthermore, we thank Erwin R. Catesbeiana (Jr.) for describing inconsistent features of super-automatic espresso machines in the brightest green.

References

[1] Apel S, Batory D, Kästner C, Saak Ge. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, 2013.

[2] `http://www.wholelattelove.com/articles/automatic_espresso_machines.cfm`. 2015-01-11.

[3] Antonia Bertolino and Stefania Gnesi. Use case-based testing of product lines. Proc. ESEC/FSE 2003. ACM. 2003. 8. 355–35.

[4] Baller H, Lity S, Lochau M, Schaefer I. Multi-objective test suite optimization for incremental product family testing. Proc. ICST 2014. 2014. 303–312.

[5] CMU Software Engineering Institute: Product Line Web Page. `http://www.sei.cmu.edu/productlines/`. 2015-01-11.

[6] Clements P, Northrop L. Software Product Lines: Practices and Patterns. Addison-Wesley, 2001.

[7] CVL Revised Submission. `http://www.omgwiki.org/variability/doku.php`. 2015-01-11.

[8] de Nicola R, Hennessy M. Testing equivalences for processes. Theo. Comp. Sci., 1984, 34: 83–133.

[9] Engström E, Runeson P. Software product line testing — a systematic mapping study. Inf. Softw. Techn.3, 2011, 53(1): 2–1.

[10] Holzmann GJ. The SPIN Model Checker. Addison-Wesley, 2003.

[11] Kahsai T, Roggenbach M, Schlingloff B-H. Specification-based testing for software product lines. Proc. SEFM 2008. IEEE. 2008. 149–159.

[12] Knapp A, Roggenbach M, Schlingloff B-H. On the use of test cases in model-based software product line development. Proc. SPLC 2014. ACM. 2014. 247–251.

[13] Knapp A, Wuttke J. Model checking of UML 2.0 interactions. Proc. MoDELS 2006 Wsh.s.

Springer. 2007. LNCS 4364. 42–51.

[14] Pérez Lamancha B, Polo Mi, Piattini M. Systematic review on software product line testing. Comm. Comp. Inf. Sci., 2013, 170: 58–71.

[15] Lochau M, Schaefer I, Kamischke J, Lity S. Incremental model-based testing of delta-oriented software product lines. Proc. TAP 2012. Springer. 2012. LNCS 7305. 67–82.

[16] Mota Silveira Neto PA da, Carmo Machado I do, McGregor JD, Almeida ES de, Lemos Meira SR de. A systematic mapping study of software product lines testing. Inf. Softw. Techn., 2011, 53(5): 407–423.

[17] McGregor JD, Northrop LM, Jarrad S, Pohl K. Initiating software product lines. IEEE Softw., 2002, 19(4): 24–27.

[18] Mota Silveira Neto PA da. A Regression Testing Approach for Software Product Lines Architectures: Selecting an Effcient and Effective Set of Test Cases. [MSc thesis], Universidade Federal de Pernambuco. Lambert Academic Publishing, 2010.

[19] Object Management Group. Unified Modeling Language Superstructure. Version 2.4.1. Specification, OMG, 2011. `http://www.omg.org/spec/UML/`.

[20] Oster S. Feature Model-based Software Product Line Testing[PhD thesis]. Technische Universität Darmstadt, 2012.

[21] Parnas DL. On the design and development of program families. IEEE Trans. Softw. Eng., 1976, 2(1): 1–9.

[22] Pohl K, Böckle G, van der Linden FJ. Software product line engineering: foundations, principles and techniques. Springer, 2005.

[23] Reuys A, Reis S, Kamsties E, Pohl K. The ScenTED method for testing software product lines. Software Product Lines. Springer. 2006. 479–520.

[24] Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. Sci. Comp. Prog., 2014, 79: 70–85.

[25] `http://www.uppaal.org`. 2015-01-11.