On the Use of Test Cases in Model-Based Software Product Line Development

Alexander Knapp
University of Augsburg, Germany
knapp@informatik.uni-augsburg.de

Markus Roggenbach
Swansea University, Wales, United Kingdom
M.Roggenbach@swan.ac.uk

Bernd-Holger Schlingloff
Humboldt University and Fraunhofer FOKUS,
Berlin, Germany
hs@informatik.hu-berlin.de

ABSTRACT

We address the question of how to select test cases in a controlled model-based software product line development process. CVL, the common variability language, provides a framework for materialization of product models from a given variability model and resolution model. Reflecting common practice, we assume that test case development for product line is independent from product development. In such a setting, the question arises which test cases can be applied to which products. To this end, we describe a procedure and tool set for assigning the outcome of a test case on an arbitrary member of a software product line using UML base and CVL variability models. As a case study, we demonstrate our approach with the example of a product line of automatic espresso machines.

Keywords

Software product lines, model-based testing, test colouring, UML.

Categories and Subject Descriptors

Software [**Software Engineering**]: Software/Program Verification—*Formal methods*; Software [**Software Engineering**]: Testing and Debugging

1. INTRODUCTION

Software Product Lines (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. One challenge in the development of these products is that the built-in software is similar, but not identical in all products; there are slight differences according to the features exhibited by a particular product. The main goal of SPL engineering is the strategic re-use of software artefacts. However, re-use increases the probability of errors. Therefore, quality assurance for SPLs is of utmost importance.

In this paper we consider the problem of testing the members of a SPL. A naive approach would be to design a specialized test suite for each member of the SPL. Such "separate test case development", c.f. [7], however, would be wasteful as it ignores that features are shared. Similar as with source code, it is desirable to re-use test cases. Therefore it is current practice to adapt the test suite from one member of a SPL to another. This "opportunistic reuse of existing test cases", c.f. [7], utilizes the fact that there will be much overlap between their test suites. Here, one manually re-works the test suite for each individual product and defines the applicability and expected outcome of each test case on an ad-hoc basis. Clearly, this is a time-consuming and tedious exercise. Therefore, Reuys et al. [7] suggest to "design test cases for reuse". However, this leads to complex test cases which include variability information.

In contrast, we suggest to design a single, "universal" test suite for a SPL. For a given materialization, the universal test suite is filtered by an automated colouring procedure. In a model-based SPL development, the base model contains realizations for all features; similarly, the suggested universal test suite shall contain tests for all features. Given a universal test suite, one then determines for each member of the SPL the subset of "applicable" test cases. Clearly there are syntactic criteria influencing applicability: 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: if the test case checks the correct functionality of a feature which is not present in the product, it is not clear what the expected outcome of this test case for the respective product is.

We study this question in the context of a model-based development process by step-wise refinement, from an abstract function model to a concrete implementation model. Here, we assume that test cases are developed independently in another department or business unit, following the usual code of conduct to separate development and testing. Given a product model and a test case, there are several possibilities, which we capture by a test case colouring scheme: (1) 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*. (2) 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 with respect to the model is *red*. (3) 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.

[Submitted as Research Paper to the 18th International Software Product Line Conference Florence, Italy, September 15-19, 2014.]

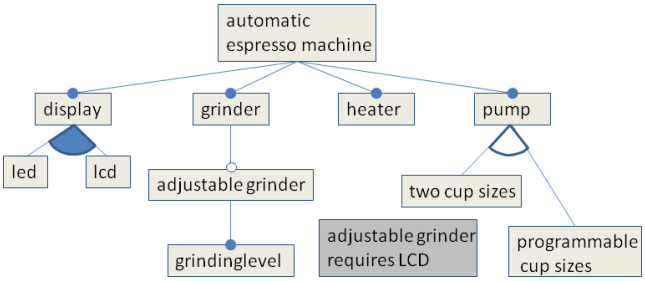

Figure 1: A feature model for automatic espresso machines

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.

The goal of the paper is to describe an automated test selection procedure based on a sound semantical theory. Subsequently, we formalize the requisite notions, describe an algorithm for automated test case colouring and its implementation, and illustrate the ideas by the example of a product line of automatic espresso machines.

2. MODELLING IN SPL ENGINEERING

Product line development involves two engineering processes: domain engineering and application engineering. For model-based SPL engineering, the artefacts produced during domain engineering are mostly models. CVL, the common variability language, is an attempt to define a syntactic framework for such model-based SPL engineering [2]. 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* is an explicit description of commonalities and differences of various products. Feature models are usually organized as and-or trees, where each node is marked with the name of a feature. An example of a feature model is given in Figure 1. It depicts part of the features of a modern automatic espresso machine. There are various manufacturers of these machines, virtually all of which organize their portfolio as a product line. For more information on features of automatic coffee machines, see [9]. In our 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 that variable's value.

Given any feature model, a *resolution model* (or simply *resolution*) is an assignment of truth values to feature names, such that all constraints are satisfied. A *base model* for the CVL is any model which is an instance of some MOF (meta-object facility) metamodel. In our work, a base model is a UML model consisting of (restricted) state machine diagrams and class diagrams. The base model contains realizations for all features; thus, if the feature model contains conflicting features, then the base model does not represent a possible product. Consequently, the base model consists not necessarily of well-formed UML diagrams. A *variability model* is a feature tree with variation points linking into the base model. In CVL, there are several kinds of variation points: existence, value assignment, substitution, and others. *Materialization* 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. In a model-based development process, the resulting product models are further refined to different products by model transformation and code generation.

There are various papers dealing with the question of how to derive test suites from a base model, a variability model, and a resolution (see, e.g., [8]). In contrast to these approaches, we do not deal with the question of test generation, and assume that test case development is independent from the product development. This reflects common practice in large companies, where there are separate departments for software development and software testing, and it also reflects situations where software development is done at a supplier's side, whereas testing is done by the vendor. That is, we regard base model or product models exclusively for the sake of product development. Test cases and test suites are crafted and maintained in a different process. Under this assumption, the problem arises which test cases can be applied to which products.

3. A TEST COLOURING PROCEDURE

Subsequently, we show how the base model, the variability model and the resolution model can be used to determine the expected outcome of a test case to a product. Our contribution is based on our previous work [3], where we here use standard UML instead of CSP-CASL. Furthermore, the developed technology is automatized to a higher degree, as it is based upon model checking rather than interactive theorem proving.

Test cases reflect intentions described in the specification. We use a three-valued colouring scheme: A test case it coloured green if it reflects a behaviour that is expected from this particular variant. 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.

To define the notion of a test case, we need to fix a *test signature* Σ. In our approach, we assume that Σ is a subset of the events which are contained in the base model. 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 events from the test signature Σ. In order to fix the semantics of a test case, we further assume that there exists a function $enabled$ assigning to each configuration state of a UML model the set of output events 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 is either an input or $e_i \notin \Sigma$. 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.

The *colour* of a test case $T = \langle e_1, \ldots, e_n \rangle$ 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 the initial configuration, and c_i is reached from c_{i-1} by e_i for all $1 \leq i \leq k$ 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 the initial configuration, and c_i is reached from c_{i-1} by t_i for all $1 \leq i \leq n$; 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.

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 the initial configuration; it is red, if the event is initially not enabled; and yellow, if it is enabled 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 $enabled(s) = \{a, b\}$, 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 $/object.op(arg)$, 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.

For automating the above defined test colouring procedure for a given materialization of a SPL and a test case, we use the tool HUGO/RT, which is a UML model translator for model checking [4]. HUGO/RT translates both the materialization and a test case into Promela, which is the input language of the model checker SPIN. The variability model and resolution are used in this translation such that the Promela code reflects both the product model and the test case. Using SPIN, we now check on the one hand whether the observer 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 failure state is reachable. If the failure state cannot be reached, the test case is coloured green, otherwise yellow.

4. EXAMPLE

We demonstrate our ideas with an example product line of automatic espresso machines using the features described in Fig. 1. Since the feature model involves ten features and different parameters, there is quite a large number of different product models. Only a few of these will be materialized as actual products in the market. We consider three elementary test cases to be evaluated on the selected product models. Their colour varies, depending on the resolution and level of abstraction of the respective product model.

4.1 Base Model

We first give a base model for the realisation of *all* features. It contains a class model (due to space limitations not shown here[1]) with variability in the setting of variables and associations. The state machine diagram in Fig. 2 describes the user interface of the espresso machines. 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.

The diagram in Fig. 3 describes the internal control structure of the espresso machines. Within the control state 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 run-time system.

4.2 Variability Model

The variability model for the product line is depicted in Fig. 4. 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. 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.

[1]The full case study with all diagrams can be found at `http://www.cs.swan.ac.uk/~csmarkus/caseStudy.pdf`.

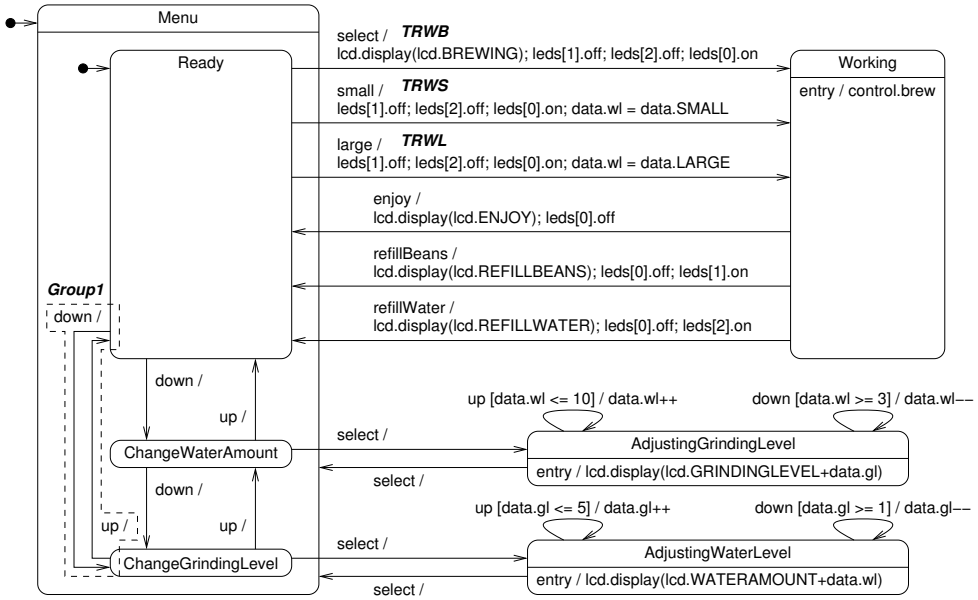

Figure 2: Espresso machines product line: user interface

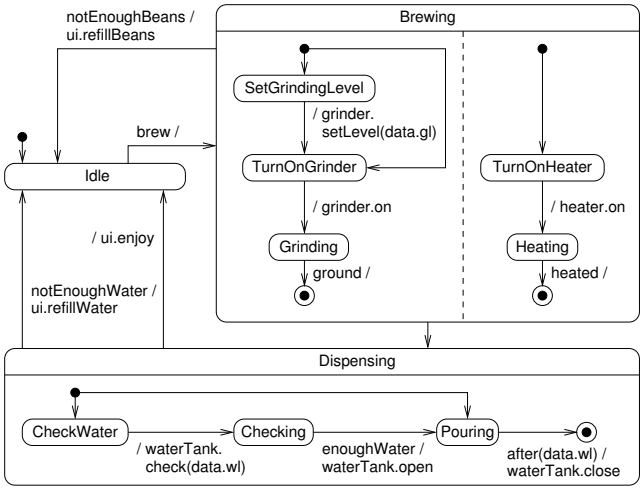

Figure 3: Espresso machines product line: control component

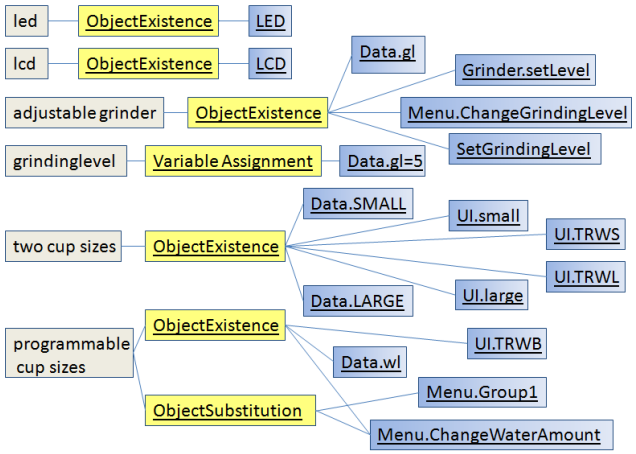

Figure 4: The variability model for the product line

4.3 Some Sample Test Cases

In order to illustrate our colouring approach, we present some sample test cases, for the sake of readability depicted as UML sequence diagrams.

The first test case, see Fig. 5, checks a functionality realised in all materializations, namely that it is possible to brew coffee. Here we use a user object for controlling the environment's input to the system and require that after a select from the user to the user interface ui the system issues a message brew to ctrl representing the machine's controller. However, by using a consider-fragment we disregard all other messages that could be produced and exchanged in the espresso machine, i.e., the observables of this test case are select and brew.

The second test case, see Fig. 6, checks if the features LCD and LED are correctly implemented. To this end, we study the internals of the reaction of the user interface to select more closely and thus take not only select and brew, but also display, off, and on to be observable.

As the above examples indicate, it is possible and desirable to organize the suggested universal test suite of a SPL around its feature model, i.e., its structure can be a consequence of the Domain Engineering process. Naturally, a universal test suite for our SPL will include far more test cases than the two presented examples.

4.4 Colouring Test Cases

Colouring "Brewing coffee" First, we consider a materialisation containing both the features *lcd* and *led*. Inspecting the behaviour of *UserInterface* in Fig. 2, we would deem this test case to be green for this particular materialisation. In fact, HUGO/RT confirms that on the one hand, brew is produced after select when ignoring the display message to the LCD and the offs and on to the LEDs; on the other hand, trivially, no other message than brew can be produced

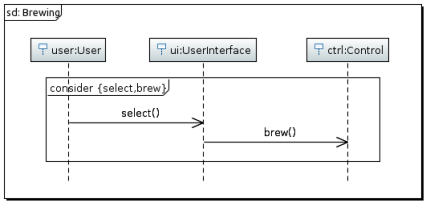

Figure 5: Test case "Brewing coffee"

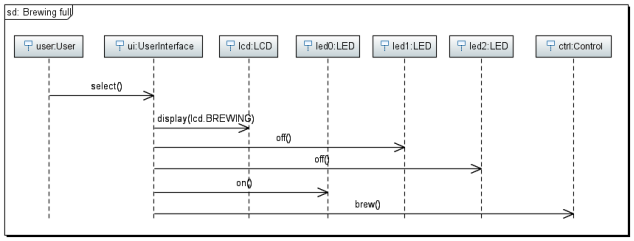

Figure 6: Test case "Brewing coffee in detail"

after select. When moving to further materialisations, where only one of the features *led* or *lcd* is present, this test case keeps its green colour. In both cases, the superfluous messages on the select-transition from Ready to Working are discarded, which has the same effect as ignoring them in the test case.

Colouring "Brewing coffee in detail" For the second test case, we move back to the materialisation showing both features *led* and *lcd*. Although the sequence of messages can be executed by the materialised espresso machine, the test case has to be coloured yellow: the sequence of consumption of the off and on messages to the LEDs is not determined. Conversely, when leaving out some off or on message from the test case in Fig. 6, the test case colour changes to red for the materialisation with both *led* and *lcd*.
Interestingly, the situation changes when we transfer this test case to the materialisation showing only the feature *lcd*; we discard the off and on messages both from the UserInterface state machine in Fig. 2 and from the sequence diagram in Fig. 6. It may seem that again the test case colour is yellow, since the ordering of consuming the display message by lcd and the sending of brew by ui is non-deterministic. However, display is declared to be an operation of LCD in the class diagram. Thus, after sending out the display call to lcd, the user interface ui has to wait for a receipt acknowledgement before being able to send out brew to ctrl, and HUGO/RT confirms that the correct test case colour is green.

5. RELATED AND FUTURE WORK

We have presented a testing theory for model-based assessment and execution of multi-variant systems. To our knowledge, this is the first theoretical treatment of the subject in the context of UML models. Our theory is well-suited for testing deterministic reactive systems, where the response functionally depends on the provided stimuli. In the UML specification, it can deal with non-determinism caused by semantic variation points, under-specification and open design decisions, by assigning the respective test cases the colour yellow, which means that it is not necessary to execute them with the product.

(Re-)using test cases in software product line testing is a long standing topic. We already discussed the relation to [7] in the introduction. In [5], the authors propose to construct test artefacts 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 system and variability models. Oster [6] 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 thesis 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 modelling of features and resolutions. However, the representative set of products could serve as a basis for an initial colouring of test cases. Bertolino et al. [1] 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 applications can be derived from it. This work could be a nice addition to our method, since we assume that the test suite is designed separately.

Our 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 as test cases and the relation to formal testing theories.

Our future plans include to apply the theory to actual industrial problems in safety-critical systems. We are looking at case studies of voltage stabilizers in wind energy plants 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.

6. REFERENCES

[1] A. Bertolino and S. Gnesi. Use Case-based Testing of Product Lines. In *Proc. ESEC/FSE'03*, pages 355–358. ACM, 2003.

[2] CVL Revised Submission. `http://www.omgwiki.org/variability/doku.php`.

[3] T. Kahsai, M. Roggenbach, and B.-H. Schlingloff. Specification-based Testing for Software Product Lines. In *Proc. SEFM'08*, pages 149–159. IEEE, 2008.

[4] A. Knapp and J. Wuttke. Model Checking of UML 2.0 Interactions. In *Proc. Models'06*, LNCS 4364, pages 42–51. Springer, 2007.

[5] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-based Testing of Delta-oriented Software Product Lines. In *Proc. TAP'12*, LNCS 7305, pages 67–82. Springer, 2012.

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

[7] A. Reuys, S. Reis, E. Kamsties, and K. Pohl. The ScenTED Method for Testing Software Product Lines. In *Software Product Lines*, pages 479–520. Springer, 2006.

[8] S. Weißleder and H. Lackner. Top-Down and Bottom-Up Approach for Model-Based Testing of Product Lines. In *Proc. MBT'13*, EPTCS 111, pages 82–94, 2013.

[9] `http://www.wholelattelove.com/articles/automatic_espresso_machines.cfm`.