Model-Based Testing

Malte Lochau[1], Sven Peldszus[1], Matthias Kowal[2], and Ina Schaefer[2]

[1] TU Darmstadt, Germany
{malte.lochau,sven.peldszus}@es.tu-darmstadt.de
[2] TU Braunschweig, Germany
{kowal,schaefer}@isf.cs.tu-bs.de

Abstract. Software more and more pervades our everyday lives. Hence, we have high requirements towards the trustworthiness of the software. Software testing greatly contributes to the quality assurance of modern software systems. However, as today's software system get more and more complex and exist in many different variants, we need rigorous and systematic approaches towards software testing. In this tutorial, we, first, present model-based testing as an approach for systematic test case generation, test execution and test result evaluation for single system testing. The central idea of model-based testing is to base all testing activities on an executable model-based test specification. Second, we consider model-based testing for variant-rich software systems and review two model-based software product line testing techniques. Sample-based testing generates a set of representative variants for testing, and variability-aware product line testing uses a family-based test model which contains the model-based specification of all considered product variants.

1 Introduction

Software more and more pervades our everyday lives. It controls cars, trains and planes. It manages our bank accounts and collects our personal information for salary or tax purposes. It comes to our homes with smart home technology in our fridges or washing machines which get increasingly connected with the Internet and personal mobile devices. Because of the ubiquity and pervasiveness of modern software systems, we have high requirements towards their trustworthiness.

Software testing greatly contributes to the quality assurance of modern software systems [55,41]. In general, testing is a partial verification technique as it only checks a software system on a selected set of inputs, while formal methods, such as model checking or program verification, allow a complete verification by considering all possible system runs. One major advantage of software testing over formal methods, however, is that testing can be performed in the actually runtime environment of the software, including all hardware and peripheral devices, while formal methods usually abstract from certain details. However, today's software system get more and more complex. They exist in many different variants in order to satisfy changing environment conditions,

M. Bernardo et al. (Eds.): SFM 2014, LNCS 8483, pp. 310–342, 2014.

such as user, technical or legal requirements. Hence, in order to ensure safety-critical, business-critical or mission-critical requirements, we need rigorous and systematic approaches for software testing.

In this tutorial, we, first, present model-based testing as an formal approach for dynamic functional testing of single systems [54]. The central idea of model-based testing is to base all testing activities on an executable formal test model of the expected system behavior. The test model can be used for test case generation, for test case execution and and for test result evaluation. In main advantages of model-based testing over classical manual testing activities is that test cases can be derived in a systematic and automatic fashion from the test models with defined coverage metrics. Furthermore, model-based testing allows the automation of test execution and test result evaluation by comparing the actual test results with the expected results expressed in the test model. When software evolves, the test models allow regression test selection by automatic change impact analysis on the test models. After an introduction of the general notions of model-based testing, we introduce the formal notions of model-based input/output conformance testing.

Second, we consider model-based testing for variant-rich software systems [50], in form of software product lines. Testing software product lines is particularly complex because the number of possible product variants is exponential in the number of product features. Hence, it is generally infeasible to test all product variants exhaustively. We review two model-based software product line testing techniques which can be used in combination to facilitate efficient testing of variant-rich software systems. First, sample-based testing allows to automatically generate a set of representative variants which should be tested instead of testing all possible product variants. Second, we present variability-aware product line testing which uses a family-based test model which contains the model-based specification of all considered product variants and define the notion of product line conformance testing.

This tutorial is structured as follows. In Sect. 2, we introduce the key notions and concepts of (software) testing. In Sect. 3, the principles of model-based testing techniques are described together with a formalization of model-based input/output conformance testing following Tretmans [54]. In Sect. 4, we extend the model-based testing principles to software product lines and describe two recent techniques for variability-aware product line testing. Sect. 5 concludes the tutorial.

2 Foundations of Software Testing

Generally speaking, (software) testing deals with the quality assurance of a (software) product. The IEEE defines the purpose of *testing* as

> [...] an activity performed for evaluating product quality, and for improving it, by identifying defects and problems [17].

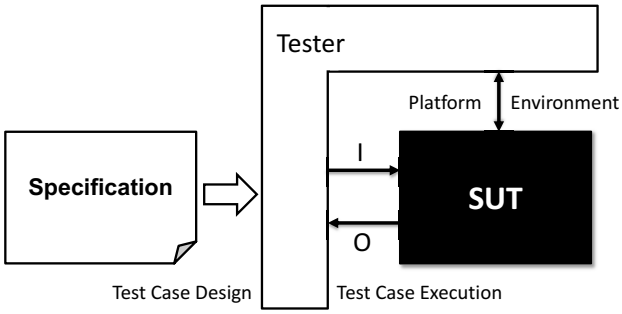


Fig. 1. General Setting of Software Testing

It characterizes the testing activity itself as

> [...] the process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspects of the system or component [17].

Thus, the notion of testing comprises any kind of activity explicitly aiming at ensuring quality requirements a software product must meet. This includes arbitrary activities *and* properties somehow relevant for the product quality goals. Hence, a wide range of testing approaches exists, differing in the methods applied and the test aims pursued. For instance, a testing method may be either *static*, e.g., systematic code inspections, or *dynamic* by means of experimental executions of the *system under test* (SUT).

The general setting for conducting (dynamic) tests on an SUT is illustrated in Fig. 1. In each test case execution, the SUT is run by a *Tester* under controlled environmental/platform conditions by stimulating accessible inputs I and observing the expected output behaviors O of the SUT. The *Tester* might be a real person, a virtual process, e.g., a test script for test automation etc. The category of dynamic testing is often further subdivided into *active* testing, i.e., real executions are enforced under experimental input sequences I and *passive* testing, e.g., by just monitoring output behaviors of the system under operation. According to Tretmans, the actual aspects to be observed as outputs during test case execution depend on the charateristics under considertion.

> (Software) testing is an activity for checking or measuring some quality characteristics of an executing object by performing experiments in a controlled way w.r.t. a specification [54].

Therefore, the *design* principles for appropriate test cases depend on those aspects under consideration. In particular, characteristics to be investigated by testing may be

– *functional*, i.e., related to some *behavioral* aspect expected from the system, e.g., by means of (visible) actions,

- *extra-functional*, i.e., concerning robustness, performance, reliability, availability etc. of (software) functions as well as
- *non-functional*, i.e., massively depending on the (hardware) platform, e.g., energy consumption, resource consumption etc.

We concentrate on test case design for finding *functional* errors. This may include finding errors in all parts of the system potentially interacting with the software. Test case *executions* perform determined sequences of input actions (stimuli) together with sequences of output actions expected from the SUT as defined by the system *specification* The resulting test verdict denotes whether the actual product reaction conforms to this expected behavior. The following notions are used in the literature for situations in which a test case execution fails [55].

- A *failure* is an undesired observable behavior of an SUT.
- A *fault* in an SUT causes a failure, e.g., by reaching a human/software error, hardware defects etc., during test execution.
- A (software) *error* is a logical error in the implementation of a requirement thus potentially leading to a fault.

A software is erroneous if it fails to satisfy its requirements. Hence, an implementation is tested against the requirements, which are either represented in an informal, e.g., textual, or in a formal way, e.g., a formal specification such as a *test model* [55].

Depending on the development phase in which test cases are applied onto an implementation, test cases are to be defined according to the current representation of the implementation available. For instance, test cases may be applied to an abstract implementation model, to implementation code fragments running on a hardware emulator, as well as to the final software fully deployed onto the target platform. Summarizing, we use the following characterization of (dynamic) software testing.

Definition 1 (Software Testing). *Software testing consists of the dynamic validation/verification of the behavior of a program on a finite set of test cases suitably selected from the usually infinite input/execution domain against the expected behavior.*

This definition essentially reflects the notion proposed in [55]. Applying testing as a verification technique is often considered as a counter part to *formal* verification techniques such as model checking. Both approaches can be opposed as follows. *Testing* allows a *partial*, i.e., incomplete verification of the correctness of an implementation with respect to a specification and thus constituting a heuristic verification method. The implementation can be tested at any level of abstraction as long as it is executable. In particular, tests can be applied to the final system implementation including any factor potentially influencing the software such as hardware components etc. Furthermore, testing can be performed by engineers at any skill level in a totally informal and pragmatic way. In contrast, *formal verification* permits a *complete* verification of the correctness of an implementation with respect to a specification. The implementation must be

represented by means of a formal abstraction of the real implementation. Hence, the major challenges when applying formal methods like model checking are (1) to ensure an implementation to be a valid refinement of the verified abstraction, (2) scalability issues, and (3) ensuring correctness of the verification tools. Both methods have advantages and disadvantages, thus complementing rather than excluding each other.

The goal of software testing is to design and apply *test suites* for a software product under test. Test suites contain a set of *test case specifications*. In practice, the design of a test suite consists of selecting sample input data for the SUT, where the concrete test derivation techniques and test case representation depend on several factors, e.g., the *test method*, e.g., static, dynamic testing, the *test aim*, e.g., functional, non-functional tests, the *test scale*, e.g., unit, component, integration, system tests, and the *information base*, e.g., black box, white box, gray box tests. The test scale corresponds to the level of abstraction considered in particular development phases where the test cases are applied. In addition, the actual testing technique heavily depends on the information base available for the system under test, e.g., accessibility to the implementation source code and platform details. Thus, black box tests comprise, e.g., combinatorial testing strategies and model-based testing as both solely consider the I/O interface of the system under test. In white box testing, the source code and further implementation details are fully accessible, whereas in gray box testing only some of those details, e.g., an architectural description, are available.

The *quality* of a test suite, e.g., with respect to the reliability of the verification results obtainable from a test suite execution is estimated by means of *adequacy criteria*. Those criteria define metrics not only to measure the suitability of a test suite, but also to guide the test case selection process, e.g., by constituting test end criteria for test case generation algorithms. For instance, *structural coverage criteria* constitute the most widespread notion for measuring test suite adequacy. Those criteria require test cases of a test suite to sufficiently traverse structural elements, i.e., *test goals*, either located in a (test) model representation, or in the code under test. Therefore, either an explicit coverage of control flow constructs like statements, decision structures, loops, and entire paths, or implicit coverage of data flows, e.g., by means of *def-use* coverage is enforced. Closely related to structural coverage criteria are data coverage criteria requiring appropriate coverage of the input data space, e.g., one-value, boundaries, equivalence classes, random-value, and all-values. Furthermore, combinatorial coverage criteria over input value domains are frequently used, e.g., pairwise, T-wise for a constant T and N-wise coverage for a variable N. Further adequacy and test selection criteria are based, e.g., on fault-models capturing well-known typical implementation faults, on mappings of test cases to requirements and scenarios, on explicit test case specification languages, and statistical methods for random generation of test data.

For further details on principles and practices of (software testing), we refer to interested reader, amongst others, to the classical text books on software testing of Myers [41] and Beizer [6], to the Dagstuhl Tutorial on formal

foundations of model-based testing by Broy et al. [9] and recent standardizations from industries [17,18,56]. Here, we limit our considerations to dynamic testing of functional characteristics at component-level based on model-based black-box knowledge in the following. This testing discipline is usually referred to as *model-based input/output conformance testing*.

3 Model-Based Testing

In this section, we provide an introduction into the fundamental concepts of model-based testing and review a formal approach to model-based input/output conformance testing initially introduced by Tretmans [54].

3.1 Fundamentals and Concepts

In model-based testing, a *test model* serves as a specification of the implementation under test (cf. Fig. 1). Depending on the model-based testing practices applied, test models may provide a comprehensive basis for any activity during the testing processes including test case derivation, test coverage measurement, test case execution, test result evaluation, and test reporting [55]. In combination with appropriate test interfaces and tool support, model-based software testing campaigns are executable in a more or less fully automated way once a (validated) test model specification, as well as a well-defined testing interface are available.

Definition 2 (Model-Based Testing [55]). *Model-based testing is the automation of black box tests.*

The implementation under test to constitutes a *black box* solely offering predefined input/output interfaces for the tester to interact with the system during testing. Beyond that, nothing is known about the internal implementation details and the computational states, data structures, hardware usage etc., during (test) executions.

In model-based testing, the test model constitutes an explicit, but usually highly abstracted representation of all behavioral aspects being relevant for the system implementation to behave correctly. Therefore, formalisms used for test modeling should offer natural notions and artifacts apparent in the testing method, e.g., a concept for test case specifications that makes distinctions between input and output behaviors and that allows the identification of test goals covered by test executions. Concerning functional testing, we are, in particular, interested in capturing the dynamics of a system, i.e., in test models that define the *behavior* of a system under test. Therefore, a test model should provide a finite representation of the potentially infinite execution domain of a software system under test, e.g., by means of high-level modeling languages such as state machines and other behavioral models as, e.g., defined by the UML [55]. Thereupon, the modeling language under consideration must incorporate a rigorous, accurate formalization together with a precise operational semantics that allows

for a clear definition of *behavioral conformance* of experimental executions of an SUT with respect to the expected behavior.

Testing in general constitutes a semi-formal, pragmatic approach for software verification/validation. The representative executions performed on the SUT may be designed ad hoc. In contrast, model-based conformance testing relies on formal specifications by means of formalized test models. Hence, the purpose of behavioral conformance testing is to compare the intended and the actually implemented behaviors and to decide whether they differ only up to some degree of confidence [55]. Recent literature on the formal foundations of conformance testing provides corresponding conceptual frameworks to denote testing principles by means of notions known from formal operational semantics and behavioral equivalences [54]. In general, verifying the correctness of a (software) system implementation i with respect to a formal behavioral specification s requires to verify an *implementation relation*

$$i \simeq s$$

to hold between both, where $\simeq$ denotes the particular equivalence relation under consideration for behavioral conformance [16]. Intuitively, this notation denotes the implementation i to be correct if it shows the same set of behaviors as permitted by the specification s. The formal semantics $[\![\cdot]\!]$ for characterizing those *sets of behaviors* depends on the representation and comparability of the specification s and the implementation i as well as the relation $\simeq$ under consideration. In many cases, it is sufficient, or even only possible, to establish a *preorder* relation

$$i \sqsubseteq s$$

to hold between an implementation and a specification, i.e., requiring the set of behaviors of the implementation to be included in the set of specified behaviors. In that sense, implementation i is *correct* if it shows *at most* the sets of (visible) behaviors as specified in s. When applying model-based testing as a verification technique, the specification s is given as a test model, e.g., represented as a state machine model. Correspondingly, a *conformance relation*

$$i \textbf{ conforms } s \;:\Leftrightarrow\; [\![i]\!] \subseteq [\![s]\!]$$

is established between the implementation and test model specification in the context of model-based testing.

Considering model-based testing, i constitutes a black box, i.e., the internal structure of the implementation under test is unknown to the tester. Thus, the verification of the behavioral conformance requires to relate a black box, i.e., a monolithic object solely offering an I/O interface that hides any internal details of the system under test, with a formal test model represented by abstract modeling entities, e.g., in terms of algebraic objects. In addition, even if an exhaustive testing campaign has been successfully executed on the implementation under test, no guaranteed statements about the correctness of the implementation can be stated as test result confidence and reproducibility depends on the internal properties of the implementation, e.g., whether non-deterministic behaviors

are potentially apparent. To overcome this mismatch, Bernot was the first to propose i to represent an (imaginary) *implementation model* i to be assumed for establishing a conformance relation **conforms** between a test model and a black box SUT [7]. This way, both the implementation i and the specification s share the same semantic domain defined by $[\![i]\!]$ and $[\![s]\!]$, respectively. Based on Bernot's abstract framework, this idea was later adopted, amongst others, by Tretmans for formalizing model-based testing frameworks with concrete test modeling formalisms under consideration [54].

The definition of behavioral conformance by means of (implicit) behavioral inclusion relation **conforms** between SUT i and specification s constitutes an *intentional* characterization of model-based testing. In contrast, *extensional* descriptions make use of the class $\mathcal{U}$ of all possible external observers (tester) explicitly comparing particular observable behaviors of i with those of s, i.e.,

$$i \textbf{ conforms } s \;:\Leftrightarrow\; \forall u \in \mathcal{U} : obs(u,i) \approx obs(u,s).$$

We now give instantiations of both kinds of characterizations of behavioral conformance in terms of the **ioco** relation and a respective test case derivation algorithm as proposed by Tretmans [54].

3.2 A Formal Approach to Model-Based Testing

Formal approaches to I/O conformance testing abstract from the concrete syntax of (high-level) test modeling languages. Instead, labeled transition systems (LTS) are used constituting a well-established semantic model for discrete, event-driven reactive control systems. To serve as test model specification for I/O conformance testing, the special sub class of *input/output labeled transition systems* is considered in the following [9]. A labeled transition system specifies system behaviors by means of a transition relation $\rightarrow \subseteq Q \times act \times Q$ defined over a set Q of states and a label alphabet act of actions. In case of I/O labeled transition systems, the set $act = I \cup U \cup \{\tau\}$ of actions is subdivided into disjoint subsets of controllable input actions I, observable output actions U and internal actions summarized under the special symbol $\tau \notin (I \cup U)$.

Definition 3 (I/O Labeled Transition System). *An* I/O labeled transition system *is a tuple* $(Q, q_0, I, U, \longrightarrow)$*, where* Q *is a countable set of* states*,* $q_0 \in Q$ *is the* initial state*,* I *and* U *are disjoint sets of* input actions *and* output actions*, respectively, and* $\longrightarrow \subseteq Q \times act \times Q$ *is a* labeled transition relation*.*

By $\mathcal{LTS}(act)$ we denote the set of LTS defined over label alphabet act. Each *computation* of a system specified by an LTS refers to some *path*

$$q_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \xrightarrow{\mu_3} \cdots \xrightarrow{\mu_{n-1}} s_{n-1} \xrightarrow{\mu_n} s_n$$

of the state-transition graph starting from the initial state q_0. Please note that we often identify an LTS s with its initial state q_0 in the following. The *behavior* of a computation is defined by the *trace* $\sigma = \mu_1\mu_2 \cdots \mu_n \in act^*$, i.e.,

the respective sequence of actions occurring as transition labels in the computation. The following notations for LTS trace semantics are frequently used in the literature [54].

Definition 4 (LTS Trace Semantics). *Let s be an I/O LTS, $\mu_i \in I \cup U \cup \{\tau\}$ and $a_i \in I \cup U$.*

$$
\begin{aligned}
s \xrightarrow{\mu_1 \cdots \mu_n} s' &:= \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdots \mu_n} &:= \exists s' : s \xrightarrow{\mu_1 \cdots \mu_n} s' \\
\neg s \xrightarrow{\mu_1 \cdots \mu_n} &:= \nexists s' : s \xrightarrow{\mu_1 \cdots \mu_n} s' \\
s \overset{\epsilon}{\Longrightarrow} s' &:= s = s' \text{ or } s \xrightarrow{\tau \cdots \tau} s' \\
s \overset{a}{\Longrightarrow} s' &:= \exists s_1, s_2 : s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Longrightarrow} s' \\
s \overset{a_1 \cdots a_n}{\Longrightarrow} s' &:= \exists s_0, \ldots, s_n : s = s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \ldots \overset{a_n}{\Longrightarrow} s_n = s' \\
s \overset{\sigma}{\Longrightarrow} &:= \exists s' : s \overset{\sigma}{\Longrightarrow} s' \\
\neg s \overset{\sigma}{\Longrightarrow} &:= \nexists s' : s \overset{\sigma}{\Longrightarrow} s'
\end{aligned}
$$

The set of traces of an LTS s is defined as

$$Tr(s) := \{\sigma \in (I \cup U)^* \mid \exists s' \in Q : q_0 \overset{\sigma}{\Longrightarrow} s'\}.$$

To illustrate the notions and concepts of I/O conformance testing based on LTS, we consider as our running example a simple *vending machine*.

Example 1. The graphical representation of an LTS is illustrated in Fig. 2 denoting different behavioral specifications of a vending machine for beverages, where $I = \{1€, 2€\}$ and $U = \{coffee, tea\}$. By convention, transition labels referring to input actions are prefixed by "?" and outputs actions by "!". Each vending machine accepts different types of coins as inputs and (optionally) returns a cup of coffee and/or tea. The trace semantics of the different specifications are given as

- $Tr(q_1) = \{?1€, ?1€{\cdot}!coffee, ?1€{\cdot}!tea\}$,
- $Tr(q_2) = \{?1€, ?2€, ?1€{\cdot}!coffee, ?1€{\cdot}!tea, ?2€{\cdot}!coffee, ?2€{\cdot}!tea\}$,
- $Tr(q_3) = \{?1€, ?2€, ?1€{\cdot}!coffee, ?2€{\cdot}!coffee\}$,
- $Tr(q_4) = \{?1€, ?2€, ?1€{\cdot}!coffee, ?2€{\cdot}!tea\}$,
- $Tr(q_5) = \{?1€, ?2€\}$,
- $Tr(q_6) = \{?1€, ?1€{\cdot}!coffee\}$,
- $Tr(q_7) = \{?1€, ?1€{\cdot}!coffee\}$,
- $Tr(q_8) = \{\}$,

where $\cdot$ denotes concatenation as usual.

According to the test assumption formulated by Bernot [7], we require both a test model specification s as well as an implementation i, i.e., the SUT, to be represented by LTS models, i.e., $s, i \in \mathcal{LTS}(act)$. We further restrict our considerations to LTS being image finite and with finite τ-sequences [54].

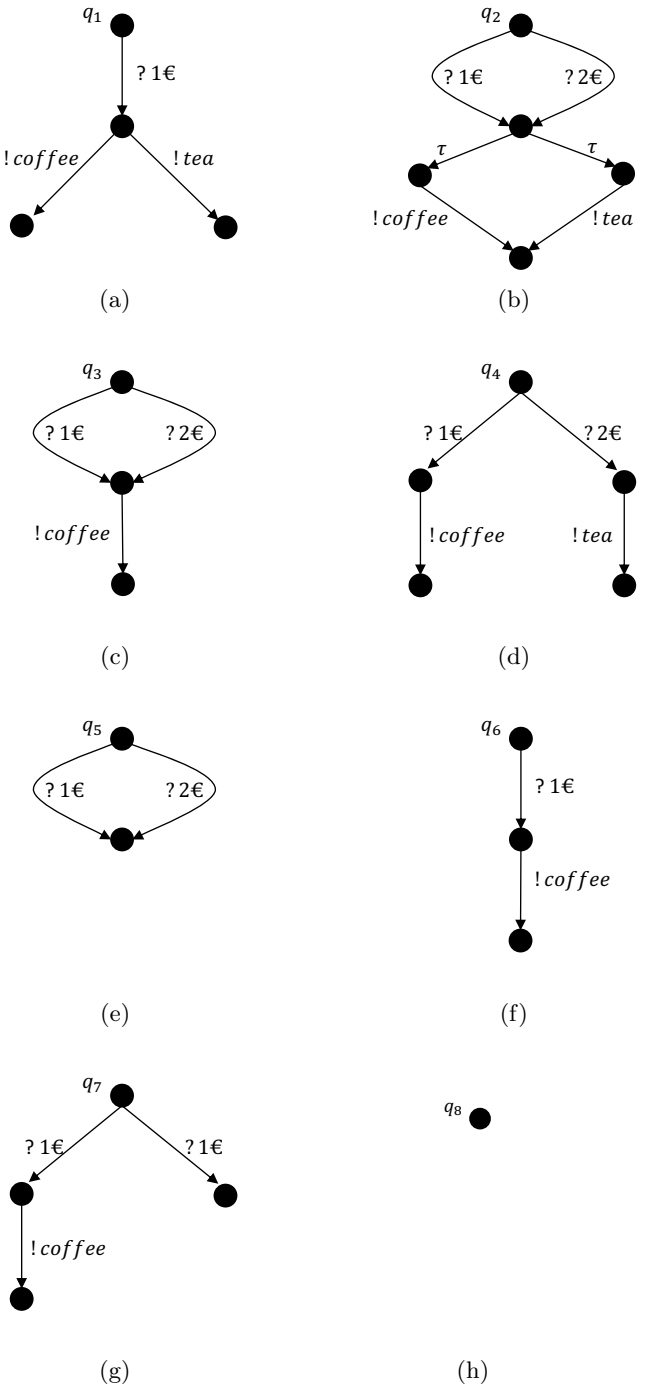


Fig. 2. Sample LTS Vending Machine Specifications

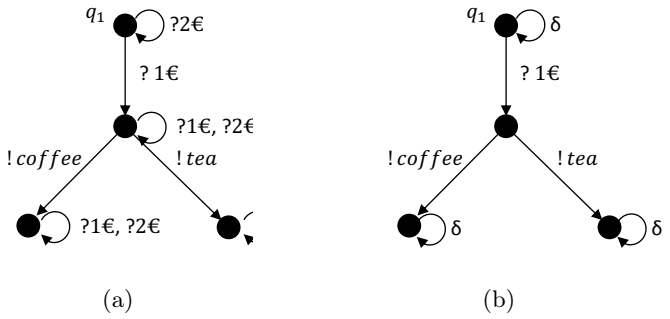


Fig. 3. Adapted Sample LTS Vending Machine Specifications

Due to the black-box setting of model-based testing, the internal structure of LTS i is unknown. However, as the SUT is assumed to never reject any inputs from the environment/tester, we require i to be at least *input-enabled* thus constituting a so-called *I/O transition system* as follows.

Definition 5 (I/O Transition System). *An LTS is* input-enabled *iff for every state $s \in Q$ with $q_0 \Longrightarrow^* s$ and for all $a \in I$ it holds that $s \overset{a}{\Longrightarrow}$.*

This property is usually referred to as *weak* input-enabledness as it only requires a system s to *eventually* react on inputs $a \in I$ in every reachable state s' after potentially performing arbitrary many internal τ-steps. By

$$\mathcal{IOTS}(I, U) \subseteq \mathcal{LTS}(I \cup U)$$

we denote the sub class if (weak) input-enabled LTS over alphabet $act = (I \cup U)$.

Example 2. None of the sample LTS shown in Fig. 2 is (weak) input enabled. A canonical construction to achieve input-enabledness for a given specification is illustrated in Fig. 3(a) adapting the sample LTS in Fig. 2(a).

For every input action $a \in I$, additional transitions $s \xrightarrow{a} s$ are introduced for states $s \in Q$ if $\nexists s' : s \overset{a}{\Longrightarrow} s'$. This way, every input is accepted in every possible state without causing any additional behavior.

Based on LTS trace semantics, a simple conformance relation might be formulated in terms of traces inclusion, i.e.,

$$i \textbf{ conforms } s \;:\Leftrightarrow\; Tr(i) \subseteq Tr(s).$$

However, this definition fails (1) to refuse trivial implementations showing no behaviors (cf. Fig. 2(h) and Fig. 2(e)) and (2) to take the asymmetric nature of LTS traces with input/output actions into account. Both aspects are explicitly addressed by the concept of observational *input/output conformance* (ioco).

The intentional characterization of observational conformance is based on the notion of *suspension traces* [54]. For an implementation i to conform to a specification s, the observable output behaviors of i after any possible sequence of inputs must be permitted by s. For this to hold, the set $out(P)$ of output actions enabled in any possible state $p' \in P$ of i reachable via a sequence σ, denoted $P = p$ **after** σ, must be included in the corresponding set of s. To further rule out trivial implementations i never showing any outputs, the concept of *quiescence* by means of a special output action δ is introduced to explicitly permit the absence (suspension) of any outputs after an input.

Definition 6. *Let s be an LTS, $p \in Q$, $P \subseteq Q$ and $\sigma \in (I \cup U)^*$.*

- $init(p) := \{\mu \in (I \cup U) \mid p \xrightarrow{\mu}\}$,
- p *is* quiescent*, denoted $\delta(p)$, iff $init(p) \subseteq I$,*
- p **after** $\sigma := \{q \in Q \mid p \stackrel{\sigma}{\Longrightarrow} q\}$,
- $out(P) := \{\mu \in U \mid \exists p \in P : p \xrightarrow{\mu}\} \cup \{\delta \mid \exists p \in P : \delta(p)\}$,
- $Straces(p) := \{\sigma' \in (I_S \cup U_S \cup \{\delta\})^* \mid p \stackrel{\sigma'}{\Longrightarrow}\}$ *where $q \xrightarrow{\delta} q$ iff $\delta(p)$.*

If not stated otherwise, we assume a given LTS to be implicitly enriched by transitions $q \xrightarrow{\delta} q$ for all quiescent states q with $\delta(p)$. Action δ may be interpreted as observational quiescence, i.e., if δ is observed, then the system awaits some input to proceed. We write $act_\delta = (act \setminus \{\tau\}) \cup \{\delta\}$ as a short hand for the set of visible actions including quiescence.

Example 3. By adding δ-transitions to quiescent states in the sample LTS specifications in Fig. 2, we are able to define the specified behaviors in terms of their suspension traces. For instance, in Fig. 3(b) the resulting LTS for q_1 is shown, where we have

$$Straces(q_1) = \{\delta, ?1€, \delta{\cdot}?1€, ?1€{\cdot}!coffee, ?1€{\cdot}!coffee \cdot \delta, \ldots\}.$$

This way, we are now able to further discriminate the behaviors of the different specifications, e.g., $?1€{\cdot}\delta \notin Straces(q_6)$, whereas $?1€{\cdot}\delta \in Straces(q_7)$

As described in Sect. 3.1, A behavioral conformance relation **conforms** to hold between implementation i and specification s requires the inclusion of all observable behaviors of i in those of s. When applying a conformance relation **conforms** the input/output conformance relation **ior** by means of suspension trace inclusion, we obtain the following definition.

Definition 7 (I/O Conformance). *Let $s \in \mathcal{LTS}(I \cup U)$ and $i \in \mathcal{IOTS}(I, U)$.*

$$i \textbf{ ior } s \;:\Leftrightarrow\; \forall \sigma \in act_\delta^* : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma).$$

Thus, input/output conformance i **ior** s ensures for every state reachable in i via a trace σ to (1) show *at most* those outputs as permitted by respective states in s reachable via σ and (2) to be *quiescent* iff a quiescent state is reachable in s via σ. Note that in case of deterministic behaviors, p **after** σ contains at most one element for all traces σ.

As a direct consequence of the definition of I/O conformance, we obtain a preorder correspondence between i and s as follows.

Lemma 1. *Let* $s \in \mathcal{LTS}(I \cup U)$ *and* $i \in \mathcal{IOTS}(I, U)$*. Then it holds that*

$$i \textbf{ ior } s \;\Leftrightarrow\; \mathit{Straces}(i) \subseteq \mathit{Straces}(s).$$

Hence, input/output conformance requires that the reaction of i to every possible environmental behavior σ is checked against those of s independent of the fact whether a proper reaction to σ is actually specified in s. In practice, conformance testing is usually limited to *positive* cases. I.e., only for those behaviors which are explicitly specified in s, the corresponding reaction of i has to be checked for behavioral input/output conformance (ioco).

Definition 8 (IOCO [54]). *Let* $s \in \mathcal{LTS}(I \cup U)$ *and* $i \in \mathcal{IOTS}(I, U)$*.*

$$i \textbf{ ioco } s \;:\Leftrightarrow\; \forall \sigma \in \mathit{Straces}(s) : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(s \textbf{ after } \sigma).$$

Again, from i **ioco** s it follows that i shows at most the behaviors that are specified in s. But, in contrast to **ior**, i may show arbitrary reactions for those behaviors not specified in s. As a consequence, Lemma 1 does not hold for **ioco** and we obtain the following correspondence.

Lemma 2. **ior** $\subset$ **ioco**.

Example 4. Again, consider the sample LTS specifications in Fig. 2 assuming δ-transitions to be added to quiescent states. Investigating the observable behavior for the possible environmental stimuli σ =?1€ and σ' =?2€, this leads to

- $\mathit{out}(q_1 \textbf{ after } \sigma) = \{\mathit{coffee}, \mathit{tea}\}$, $\mathit{out}(q_1 \textbf{ after } \sigma') = \{\}$
- $\mathit{out}(q_2 \textbf{ after } \sigma) = \{\mathit{coffee}, \mathit{tea}\}$, $\mathit{out}(q_2 \textbf{ after } \sigma') = \{\mathit{coffee}, \mathit{tea}\}$,
- $\mathit{out}(q_3 \textbf{ after } \sigma) = \{\mathit{coffee}\}$, $\mathit{out}(q_3 \textbf{ after } \sigma') = \{\mathit{coffee}\}$,
- $\mathit{out}(q_4 \textbf{ after } \sigma) = \{\mathit{coffee}\}$, $\mathit{out}(q_4 \textbf{ after } \sigma') = \{\mathit{tea}\}$
- $\mathit{out}(q_5 \textbf{ after } \sigma) = \{\delta\}$, $\mathit{out}(q_5 \textbf{ after } \sigma') = \{\delta\}$
- $\mathit{out}(q_6 \textbf{ after } \sigma) = \{\mathit{coffee}\}$, $\mathit{out}(q_6 \textbf{ after } \sigma') = \{\}$
- $\mathit{out}(q_7 \textbf{ after } \sigma) = \{\mathit{coffee}, \delta\}$, $\mathit{out}(q_7 \textbf{ after } \sigma') = \{\}$
- $\mathit{out}(q_8 \textbf{ after } \sigma) = \{\}$, $\mathit{out}(q_8 \textbf{ after } \sigma') = \{\}$.

For instance, assume that q_6 is adapted to be weak input-enabled, then it holds that q_6 **ioco** q_7, but not vice versa due to the additional quiescent state of q_7. Similarly, we have q_2 **ioco** q_1 as no behavior for !2€ in q_2 is specified in q_1, thus, leaving open implementation freedom in q_2. In contrast, q_1 **ioco** q_2 does not hold as q_1 shows quiescent behavior for !2€ which is not permitted by q_2.

Although the set of suspension traces which has to be verified on i for establishing **ioco** to some s is now limited to $\mathit{Straces}(s)$, this set is, however, still potentially infinite making input/output conformance verification impracticable. The set of suspension traces under consideration is further restricted to (finite) sub sets $\mathcal{F} \subseteq \mathit{act}_\delta^*$ and the resulting restricted **ioco**-relation is denoted as

$$i \textbf{ ioco}_{\mathcal{F}} \; s \;:\Leftrightarrow\; \forall \sigma \in \mathcal{F} : \mathit{out}(i \textbf{ after } \sigma) \subseteq \mathit{out}(s \textbf{ after } \sigma),$$

where **ior** = $\textbf{ioco}_{\mathit{act}_\delta^*}$ and **ioco** = $\textbf{ioco}_{\mathit{Straces}(s)}$ holds.

The notions considered so far constitute *intentional* characterizations of behavioral input/output conformance. In addition, an *extensional* characterization of **ioco** is given by means of test cases t, i.e., observer processes derivable from a specification s and applicable to SUT i such that

$$i \textbf{ passes } t \;:\Leftrightarrow\; obs(i,t) \approx obs(s,t).$$

In order to define the interaction of a test case t with SUT i during test case execution in a formal way, test cases are also represented as I/O labeled transition systems. In particular, considering a specification $s \in \mathcal{LTS}(I,U)$ and a corresponding SUT $i \in \mathcal{IOTS}(I,U)$, the domain $\mathcal{TEST} \subseteq \mathcal{LTS}(U,I)$ contains those *test cases* derived from s and applied to i for verifying input/output conformance of i with respect to s. Due to the asymmetric nature of communication between I/O-labeled LTS, the input and output alphabets are reversed in $t \in \mathcal{TEST}$ compared to those of s and i. In addition, the special input action $\Theta \notin U \cup I$ represents the counterpart of δ, i.e., when observed during test case execution, Θ denotes the occurrence of a quiescent state in i.

Definition 9 (Test Case). *A* test case t *is an LTS such that*

- t *is* deterministic *and has a* finite *set of traces,*
- Q *contains terminal states* **pass** *and* **fail** *with* $init(\textbf{pass}) = init(\textbf{fail}) = \varnothing$ *and*
- *for each non-terminal state* $q \in Q$ *either (1)* $init(q) = \{a\}$ *for* $a \in I$ *or (2)* $init(q) = U \cup \{\Theta\}$

holds.

Thus, each test case corresponds to a suspension trace of s such that in every test step, i.e., a transition in t, either (1) one particular input is stimulated in i, or (2) every possible output potentially emitted by i is accepted (including quiescence). If an unexpected output is observed, termination state **fail** is immediately entered and, otherwise, termination state **pass** is eventually reached after a finite sequence of (alternating) test steps. An algorithm for deriving test cases t from specifications s after a transformation into a respective *suspension automaton* can be found in [54].

Example 5. Consider the test case in Fig. 4(a) derived from specification q_1 in Fig. 2(a). For an implementation i to pass this test case, it has to accept the input !1€ and then either to return a *coffee*, or a *tea* as output, whereas no output, i.e., quiescence Θ, is an erroneous behavior. In contrast, the test case in Fig. 4(b) is derived from q_7 in Fig. 8(c) permitting *coffee* as well as *nothing* as outputs after inserting 1€ as input. Thus, this test case is, e.g., capable to distinguish implementations complying q_7 from those complying q_6 for which no quiescence is allowed after inserting 1€.

A test suite $T \subseteq \mathcal{TEST}$ is a finite set of test cases. The following properties have been proven to hold for input/output conformance testing based test suites T designed on the basis of suspension traces [54].

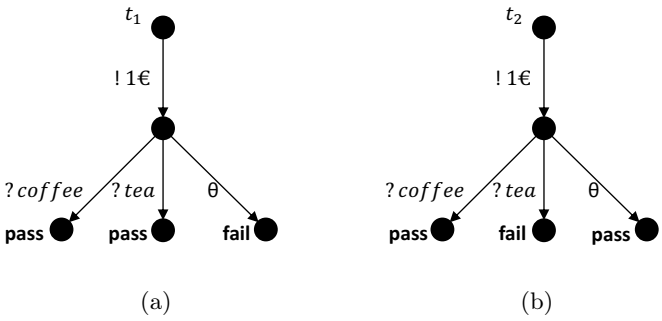


Fig. 4. Sample Test Cases for the Vending Machine

Theorem 1. *Let* $s \in \mathcal{LTS}(I \cup U)$, $i \in \mathcal{IOTS}(I \cup U)$ *and* $\mathcal{F} \subseteq Straces(s)$. *Then it holds that*

1. *any derivable test case* $t \in \mathcal{TEST}$ *is* sound, *i.e., i* ***ioco*** *o implies that i passes t and*
2. *the set* $\mathcal{TEST}$ *of all derivable test cases is* exhaustive, *i.e., i* ***ioco*** *o if i passes all* $t \in \mathcal{TEST}$.

Based on this fundamental concept of formal input/output conformance, various enhanced results, e.g., concerning compositionality properties of **ioco** [8], as well as extensions to **ioco** concerning advanced system characteristics, e.g., real time [52] and hybrid behaviors [43] have been proposed.

4 Model-Based Testing of Software Product Lines

Until now, we assumed an SUT to constitute a monolithic software system with predefined and fixed amount of functionality. However, modern software systems usually expose various kinds of *diversity*, e.g., due to extensible configurability [50]. The corresponding software implementations comprise *families* of similar, yet well-distinguished software product *variants*. Software product line engineering [12] is a well-established paradigm for concisely engineering those kinds of variant-rich software implementations including strategies for efficiently testing families of similar product variants under test.

4.1 Software Product Line Engineering and Testing

A software product line constitutes a configurable software system built upon a common core platform [12]. Product implementation variants are derivable from those generic implementations in an automated way by selecting a set of domain *features*, i.e., user-visible product characteristics, to be assembled into a customized product variant. Software product line engineering defines a comprehensive process for building and maintaining a product line. During domain

engineering, a product line is designed by (1) identifying the set of relevant domain features within the problem space and (2) by developing corresponding engineering artifacts within the solution space associated with a feature (combination) for assembling implementation variants for feature selections. During domain engineering, logical dependencies between features further refine the *valid configuration space* by restricting combinations of features. For instance, domain feature models provide an intuitive, visual modeling language for specifying the configuration space of a product line [27] (cf. Sect. 4.2).

Features not only correspond to configuration parameters within the problem space of a product line, but also refer (to assemblies of) engineering artifacts within the solution space at any level of abstraction. For instance, concerning the behavioral specification of variable software systems at component level, modeling approaches such as state machines are equipped with feature parameters denoting well-defined variation points within a generic product line specification including any possible model variant [11]. This way, explicit specifications of common and variable parts among product variants within the solution space allow for a systematic reuse of engineering artifacts among the members of a product family.

Also testing is considered an integral part of software product line engineering. *Reusable test artifacts*, e.g., variable test models and test cases designed during domain engineering are applied to those SUT assembled for the respective *product variants under test* during application engineering. McGregor was one of the first to provide a systematic overview of how to adopt recent testing notions and activities to product line engineering [37]. He identified different *scopes* under consideration in SPL testing, namely the entire SPL, a particular product, as well as individual assets, i.e., feature components and their integration. In order to facilitate large-scale reuse of test artifacts among product variants when testing an SPL, common test artifacts can be organized as SPL artifacts as well. In [38], McGregor et al. further elaborate reuse potentials in SPL testing by proposing SPL testing approaches explicitly taking variability among *products under test* into account. A first survey on product family testing approaches is given by Tevanlinna et al. [53]. The authors focus on the adoption of regression testing principles for variability-aware testing collections of similar products. The application of model-based testing principles to SPL testing was first mentioned in [42] as well as in [49]. In [46], Oster et al. provide a survey on SPL testing approaches focusing on model-based testing. Further comprehensive surveys on recent SPL testing approaches can be found in [20] and in the mapping study provided in [40].

The general setting for the (model-based) testing of a product line is illustrated in Fig. 5 extending the previous setting for single system testing in Fig.1. Testing a product line implementation with n possible product variants against a product line specification essentially requires to verify that

$$i_k \textbf{ conforms } s_k, 1 \leqslant k \leqslant n,$$

holds, i.e., to test every individual product implementation variant i_k against its corresponding specification variant s_k. As the number n of product variants

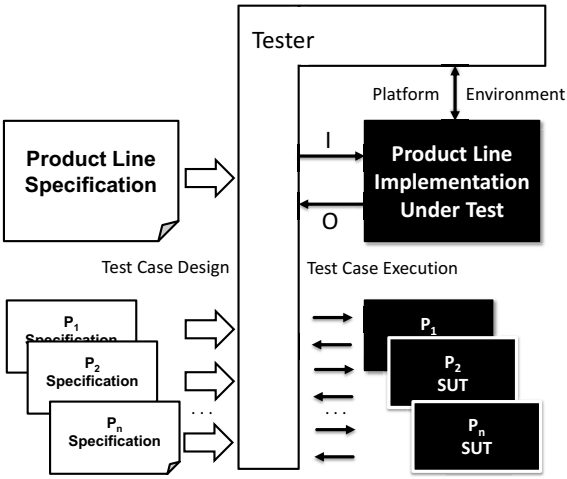


Fig. 5. General Setting of Product Line Testing

grows exponentially with the number of features, this product-by-product approach is, in general, infeasible. In addition, due to the high degree of similarity and corresponding (specification and implementation) artifact reuse potentials among the different variants, repetitive exhaustive test modeling, test suite derivation and test execution for every particular variant causes lots of redundant efforts. To cope with those challenges, different product line testing strategies have been proposed in the literature and are explained in the following.

Reusable Product Line Test Model. In contrast to (re-)modeling every product variant test model specification anew from scratch, a *reusable* product line test model is built that (virtually) comprises every possible model variant. Those model elements that are common to all members of the product line become part of every test model variant, whereas variable elements are only mapped into those model variants for which they are relevant. One of the most common approaches for reusable product line test modeling are so-called 150% specifications, where all common and variable elements are part of one model whose set of elements constitutes a superset of the test model variants. The projection of a particular test model variant from a 150% model is done, e.g., by adding explicit annotations to the variable elements by means of selection conditions over feature parameters [14], or by defining implicit behavioral restrictions by means of modal specifications combined with deontic logics [4,3,2]. Hence, a product line test model comprises two parts, i.e., (1) a configuration model, e.g., given by a domain feature model (cf. Sect. 4.2) defining the valid product space of the product line under test and (2) a 150% test model, e.g., by means of a modal I/O labeled transition system (cf. Sect.4.3). Based on a 150% test model, a product line may be tested product-by-product without re-modeling every variant anew.

However, to also reduce the efforts for test suite derivations and executions, further strategies have been proposed.

Sample-based Product Line Testing. In this strategy, only a *representative subset* of variants is considered for which product-specific test suites are generated, whereas those variants that are unselected remain untested. Various coverage criteria and subset selection heuristics have been proposed, e.g., inspired by combinatorial testing [33,25,44,13,47,23,29,24].

Regression-based Product Line Testing. In this strategy, only those test cases are generated anew for a variant under test that are not reusable from a previously tested variant. The required test case reuse analysis is similar to change impact analysis techniques known from regression testing [53,19,34]. Again, every single variant has to be considered to guarantee a complete product line test coverage.

Family-based Product Line Testing. In this strategy, a test suite is derived from a 150% test model rather than from the test model variants. This way, each test case is connected to the subset of product variants for which it is valid. A complete test coverage of a product line is achievable without considering any particular product variant [11].

In this tutorial, we will focus on two approaches for model-based product line testing. In Sect. 4.2, we present a technique for sample-based product line testing using a domain feature model for selecting a representative product subset under test, and in Sect. 4.3, a family-based approach for variability-aware product line test modeling and test suite design is presented based on modal input/output LTS specifications.

4.2 Sample-Based Software Product Line Testing

The *domain model* (sometimes referred to as variability model) plays a pivotal role in software product line engineering plays, because it contains information about the product features and its dependencies. A common representation for the domain model are feature models (FM), which are usually created during a (feature-oriented) domain analysis [15,27]. In Fig. 6, we show an exemplary feature model of a vending machine where features refer to different drinks or payment methods. A *feature model* is a hierarchical structure, where features can be selected in a top-down manner. Different constraints can be modeled for the contained features: First, each feature can be optional (denoted by the white bullet, e.g. **Tea**) or mandatory (denoted by the black bullet, e.g. **Coffee**). Second, features may be used for grouping (e.g. feature **Beverage**) and contain no functionality themselves. Additionally, we can specify group constraints on sibling features, like *alternative* and *or* groups. Alternative-features are mutually exclusive and cannot be selected for the same variant, whereas or-features have no upper bound. Finally, we can express dependencies between features using *cross-tree constraints*, which are expressed by propositional formulas.

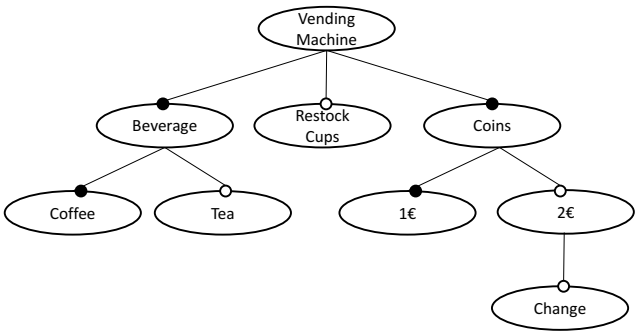


Fig. 6. Domain Feature Model of Vending Machine SPL

In most cases, features are developed and tested separately by several teams. But even if we assume that they work without errors for themselves, it is not assured that they still work error-free after an integration of several features into a larger system. Such erroneous and unexpected behavior is referred to as *feature interaction (FI)*. One of the most promising techniques to detect feature interactions is *sample-based combinatorial interaction testing (CIT)*, because is uses the domain feature model to derive a small number of variants which have to be tested. This set of product variants is supposed to cover relevant combinations of features. The sample-based product line testing technique is subdivided into three steps:

1. Create the feature model
2. Generate a subset of variants based on the FM, covering relevant combinations of features
3. Apply single system testing to the selected variants

One method for CIT is *pairwise testing*, which tries to cover all combinations of two features by the selected set of variants and is able find FIs between two features. To this end, both features must be present, not present and only one must be present in at least one tested variant to fulfill the pairwise testing criterion. It is possible to cover combinations of one, three, four, five and six features as well, but there exists a trade-off between computation time and test coverage. The higher t (where t is the number of features), the higher is the test coverage, but also the computation time to find a corresponding set of product variants [30] and the resulting number of product variants to be tested. Pairwise testing detects about 70% of all errors in a system (3-wise 95% error detection).

The selected set of products to be tested is also called *covering array*. Generating covering arrays is equivalent to the set covering problem which is a NP-complete decision problem in combinatorics. We explain the problem in the following by means of a small example. Given a set S with $S = \{a, b, c, d, e\}$. S is divided into several subsets $M = \{\{a, b, c\}, \{b, d\}, \{c, d\}, \{d, e\}\}$, which represent valid product configurations. The challenge of the set covering problem is to

find the minimal number of sets in M that cover the complete set of S. In our example, the solution is $L = \{\{a, b, c\}, \{d, e\}\}$.

However, this approach requires that all product variants are already known. Since, the number possible solutions in a FM grows exponentially with the number of features, it is almost impossible to compute all valid variants before [25]. Even finding a single valid variant in a large feature model is equal to the NP-complete Boolean Satisfiability Problem (SAT). Luckily, we are mostly dealing with realistic FMs. Actual customers should be able to configure the FM of the SPL in a decent amount of time. No company would introduce a FM, where a customer needs thousands of years to select a valid variant. Mendonca et al. [39] proved the efficient satisfiability of realistic FMs with additional cross-tree constraints.

Chvátal's Algorithm (1979). One of the first heuristic greedy algorithms to solve the set covering problem was developed in 1979 by Chvátal [10]. The algorithm does not calculate the optimal solution. It is also not yet specialized for product variant selection.

The algorithm is divided into four separate steps. The solution, i.e., the set cover, is stored in the set L which is empty at the beginning. The set M contains the set of possible subsets M_i. The algorithm selects the set M' with the highest number of uncovered elements. This set is added to the solution L and removed from all sets M_i. The algorithm terminates if there are no more elements to select.

1. Step: Set $L = \varnothing$
2. Step: **If** $M_i = \varnothing, \forall\ i, i \in \{1, 2, ..., n\}$ **Stop**. **Else** find M', where number of uncovered elements is maximized
3. Step: Add M' to L and replace each M_i by $M_i - M'$
4. Step: Jump to Step 2

The worst case is if M only consists of subsets with different elements so that the algorithm must add each subset to L and $L = M$ holds at the end.

Adaptation of Chvátal's Approach to FMs. Johansen et al. [25,26] have done an extensive amount of research in adapting and improving the original algorithm of Chvátal for product variant selection on the basis of FMs. In the following, we explain their algorithm shown as Algorithm 1.

Initially, the algorithm needs an FM as input. All possible t-tuple combinations of features are generated and written into the set S. This set includes invalid tuples as well. For, e.g. $t = 2$, all combinations of two features are present in S after the first step. After the creation of a new empty product configuration k (line 3), the algorithm iterates through all tuples in S and tries to add the tuple $p \in S$ to the configuration k. This is only possible, if the configuration stays valid with the selected tuple with respect to the feature model. The validity check is done by a standard SAT-Solver. As a result, the configuration grows, and the set S shrinks, since covered tuples are removed from S. A configuration k is added

```
   input  : arbitrary FM
   output: t-wise covering array
 1 S ← all t-tuples
 2 while S ≠ ∅ do
 3 |  k ← new and empty configuration
 4 |  counter ← 0
 5 |  foreach tuple p in S do
 6 |  |  if FM is satisfiable with k ∪ p then
 7 |  |  |  k ← k ∪ p
 8 |  |  |  S ← S\{p}
 9 |  |  |  counter ← counter + 1
10 |  |  end
11 |  end
12 |  if counter > 0 then
13 |  |  L ← L ∪ (FM satisfy with {k})
14 |  end
15 |  if counter < # of features in FM then
16 |  |  foreach tuple p in S do
17 |  |  |  if FM not satisfiable with p then
18 |  |  |  |  S ← S\{p}
19 |  |  |  end
20 |  |  end
21 |  end
22 end
```

Algorithm 1. Adaptation of the algorithm for FMs

to the final solution L (line 13) in case that at least one tuple is contained. A configuration is extended with other features, e.g., mandatory features, in order to generate a valid product variant for the feature model based on the tuples in k (cf. "FM is satisfiable with k").

The variable *counter* in the last loop makes sure that all invalid t-tuples are removed from S at some point during computation. This point has been identified by empirical studies. It would be inefficient, e.g., to remove the invalid feature tuples at he beginning, because the SAT-Solver must check too many valid tuples [25].

The vending machine example (see Fig.6) has exactly 12 valid configurations. The covering array for $t = 2$ contains only six variants (see Table 1). Even in such small FMs, we are able to save 50% time for tests with the help of sample-based product line testing.

Improved algorithm ICPL. ICPL is one of the most advanced and efficient algorithms for computing a t-wise covering array. It is based on the above algorithm with several logical and technical improvements. The main goal is to find all valid tuples to be covered by the t-wise covering array as fast as possible since checking invalid tuples slow the whole process down. Single satisfiability checks

Table 1. Covering array for the vending machine

Feature\Product	0	1	2	3	4	5
Coffee	X	X	X	X	X	X
Beverage	X	X	X	X	X	X
2€		X	X	X	X	
Change		X			X	
Tea		X	X			X
Restock Cups		X		X		X
1€	X	X	X	X	X	X
Coins	X	X	X	X	X	X
Vending Machine	X	X	X	X	X	X

for each tuple are not efficient to identify invalid tuples. ICPL takes advantage of the property that a covering array of strength t is a subset of an array with strength $t + 1$, which is proved in [26]. The algorithm calculates all t-wise covering arrays $1 \leqslant n < t$, where $n, t \in \mathbb{N}$, at first. This step improves the overall tuple covering process and provides an earlier identification of invalid tuples (see [25] for more detailed information). In one iteration, one t-tuple is covered after the other. ICPL uses the knowledge that an already covered tuple cannot be added to the current configuration with another assignment of the features which means that t-tuples with other assignments for the contained features can be skipped instantly. The skipped tuples must not be deleted from the set, since they may be valid in another configuration. Likewise, if all features of the FM are already contained in the current configuration, it is not possible to add any other tuple. All remaining tuples can be skipped for this iteration.

The parallelization of the algorithm allows shortening the computation time significantly. With respect to the number of CPU cores, the original t-sets are split up and equally divided over the cores. It is done in several points in ICPL, e.g., for finding invalid t-tuples. The whole computation time of ICPL is almost inversely proportional with the number of cores due to high parallelism.

To provide an impression of the computation time of ICPL and the size of the computed covering array, Table 2 shows the results for four larges FMs in terms of the number of features and the number of constraints. The largest FM with nearly 7000 features is one version of the popular Linux kernel. Instead of testing millions or billions of variants, we only need to test the 480 product configurations, which are calculated by the ICPL algorithm for the Linux kernel. The computation time with roughly nine hours is quite fast.

Further Improvements to Feature Interaction Coverage. The main goal of sample-based software product line testing is to identify errors caused by feature interactions. The standard CIT methods, as decried above, use all t-tuple combinations for features to ensure a 100% coverage of the FM. It is possible to be more efficient at this point, since not all features interact with each other. Feature interactions usually occur via shared resources or communication. These

Table 2. ICPL Evaluation [25]

Feature Model	Features	Constraints	2-wise size	2-wise time (s)
2.6.28.6-icse11.dimacs	6,888	187,193	480	33,702
freebsd-icse11.dimacs	1,396	17,352	77	240
ecos-icse11.dimacs	1,244	2,768	63	185
Eshop-fm.xml	287	22	21	5

two interaction types are identified as the most crucial ones [35,21]. The information about such interactions is present in development documents, such as system specifications or architectural descriptions. Based on such specifications, we can annotate FMs with the respective information about shared resources and communication between features. Annotating the FM with this additional information provides us with the advantage of reducing the t-tuple input set for the CIT algorithm. We generate only the most important tuples, where interactions are most likely to occur with regard to the specification. Less tuples have to be covered, which results in a faster computation time and a smaller covering array [29].

4.3 Variability-Aware Software Product Line Testing

Interface theories provide formal approaches for the definition of the observable behaviors which a component implementation is allowed to show by abstracting from the concrete implementation details [48,5]. Modal interface specifications further distinguish between *optional* and *mandatory* behaviors by means of *may*/*must* modality in order to leave open implementation freedom up to a certain degree. In a *modal transition system* (MTS) each transition is either a *may*, or a *must* transition [32,48,5,36]. The set of valid implementations of a modal specification correspond to the set of modal *refinements* of that specification each comprising at least all must behaviors and at most all may behaviors. In addition, a *compatibility* notion defines criteria for valid compositions of components with respect to their modal specifications.

Various approaches for applying modal specifications as product line modeling formalism have been proposed [22,31,4]. By interpreting *must* behaviors as commonality and *may* behaviors as variability among the product line variants, each implementation corresponds to one particular product configuration. Hence, modal refinement corresponds to component implementation variant derivation within the solution space. Those sets of variants may be further restricted in terms of compatibility to other components and/or the environment/user [31]. Recent approaches focus on family-based product line model checking based on modal specifications [4,3,2]. In contrast, we consider MTS to denote variable test model specifications as a basis for an intentional characterization of model-based input/output conformance testing for a software product line implementations under test.

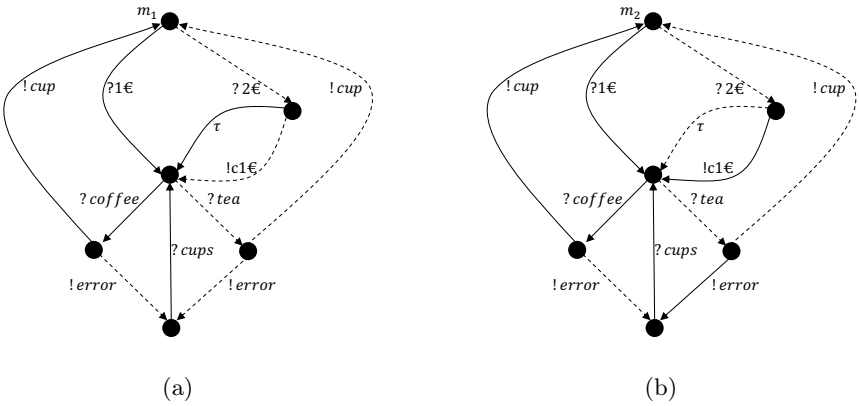


Fig. 7. Sample MTS Vending Machine Product Line Specifications

Formally, modal transition systems extend LTS by incorporating two transition relations to distinguish two different transition modalities, namely (possible) *may* and (mandatory) *must* transitions.

Definition 10 (Modal I/O Transition System). *A* model I/O transition system *is a tuple* $(Q, q_0, I, U, \longrightarrow_\diamond, \longrightarrow_\square)$*, where* Q *is a countable set of* states*,* $q_0 \in Q$ *is the* initial state*,* I *and* U *are disjoint sets of* input actions *and* output actions*, respectively,* $\longrightarrow_\diamond \subseteq Q \times act \times Q$ *is a may-transition relation, and* $\longrightarrow_\square \subseteq Q \times act \times Q$ *is a must-transition relation such that* $\longrightarrow_\square \subseteq \longrightarrow_\diamond$ *holds.*

Requiring $\longrightarrow_\square \subseteq \longrightarrow_\diamond$ ensures an MTS to be *syntactically consistent*, i.e., mandatory behaviors are always also allowed. By $\mathcal{MTS}(act)$ we denote the set of modal transition systems labeled over alphabet *act*. Again, we often use an MTS m and its initial state q_0 as synonyms in the subsequent definitions and examples.

Example 6. The graphical representation of an MTS is illustrated in Fig. 7 specifying two modal versions of a vending machine extending the simple vending machines from Sect. 3. We now have $I = \{1€, 2€, \mathit{coffee}, \mathit{tea}, \mathit{cups}\}$ and $U = \{c1€, \mathit{cup}, \mathit{error}\}$, respectively. Transitions with *may* modality are denoted by dashed arrows, whereas those with *must* modality are denoted by solid arrows. Hence, a vending machine implementing specification m_1 in Fig. 7(a) must accept 1€ coins as inputs and may optionally also accept 2€ coins. As each beverage costs 1€, the vending machine may further output 1€ change (output action $c1€$) if 2€ have been thrown in. The machine offers coffee per default but may also allow for choosing tea via inputs *coffee* and *tea*. A cup containing the selected beverage is dispensed as long as cups are available within the machine. Otherwise, an (optional) error output may be given and new cups may be inserted. The alternative specification m_2 alters m_1 in two ways: (1) providing change after inserting 2€ must be implemented whereas omitting $c1€$ is

optionally allowed via the τ-transition and (2) whenever tea is selectable and the machine is running out of cups, error handling must take place.

We adapt the notion of traces previously defined for LTS to MTS by taking the modality $\gamma \in \{\Box, \Diamond\}$ of transitions $s \xrightarrow{\mu}_{\gamma} s'$ into account.

Definition 11 (MTS Trace Semantics). *Let m be an MTS. The set of* modal traces *is defined as*

$$Tr_{\gamma}(m) := \{\sigma \in (I \cup U)^* \mid \exists s \in Q : q_0 \overset{\sigma}{\Longrightarrow}_{\gamma} s\}.$$

From syntactical consistency, it follows that $Tr_{\Box}(s) \subseteq Tr_{\Diamond}(s)$ holds. Thereupon, an adaption of the further trace notations for LTS (cf. Def. 4) to MTS can be done, correspondingly.

An MTS m constitutes a *partial* system specification in which those behaviors corresponding to *may*-traces are considered *optional* leaving open implementation freedom within well-defined bounds. Retrieving an implementation variant from a modal specification by selecting/neglecting optional behaviors corresponds to the concept of *modal refinement*. A modal specification m_1 is a *refinement* of a modal specification m_2 if (1) the mandatory behaviors of m_2 are preserved by m_1 and (2) the possible behaviors of m_1 are permitted by m_2.

Definition 12 (Modal Refinement). *Let s, t be two MTS with $act = act_s = act_t$. A relation $R \subseteq Q_s \times Q_t$ is a (weak)* modal refinement *iff whenever sRt and $a \in act \setminus \{\tau\}$ it holds that*

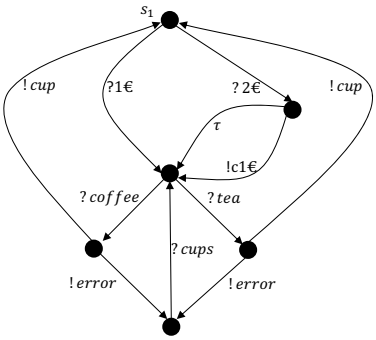
1. *if $t \xrightarrow{a}_{\Box} t'$ then $\exists s' : s \xrightarrow{\tau}{}^{*}_{\Box} \xrightarrow{a}_{\Box} s'$ and $(s', t') \in R$,*
2. *if $s \xrightarrow{a}_{\Diamond} s'$ then $\exists t' : t \xrightarrow{\tau}{}^{*}_{\Diamond} \xrightarrow{a}_{\Diamond} t'$ and $(s', t') \in R$, and*
3. *if $s \xrightarrow{\tau}_{\Diamond} s'$ then $\exists t' : t \xrightarrow{\tau}{}^{*}_{\Diamond} t'$ and $(s', t') \in R$.*

The largest (weak) modal refinement relation is denoted by $\leqslant_m$ and s is a (weak) model refinement *of t iff there is weak modal refinement containing (s_0, t_0).*

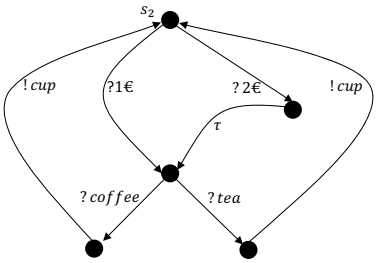
A modal refinement s is *complete* if $\longrightarrow_{\Box} = \longrightarrow_{\Diamond}$ holds. A complete refinement s of t is an *implementation* of t.

Example 7. Consider the complete refinements in Fig. 8 referring to the modal vending machine specifications in Fig. 7. Please note that we assume an (implicit) pruning of the state-transition removing those transitions becoming unreachable after a modal refinement. We observe the following (complete) refinements.

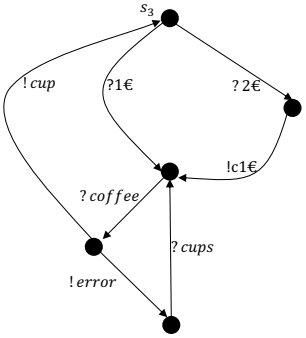
- $s_1 \leqslant_m m_1$ and $s_1 \leqslant_m m_2$
- $s_2 \leqslant_m m_1$ and $s_2 \not\leqslant_m m_2$
- $s_3 \not\leqslant_m m_1$ and $s_3 \leqslant_m m_2$
- $s_4 \leqslant_m m_1$ and $s_4 \not\leqslant_m m_2$
- $s_5 \leqslant_m m_1$ and $s_5 \leqslant_m m_2$
- $s_6 \not\leqslant_m m_1$ and $s_6 \not\leqslant_m m_2$



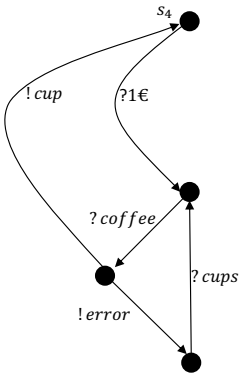
(a)



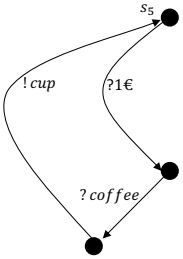
(b)



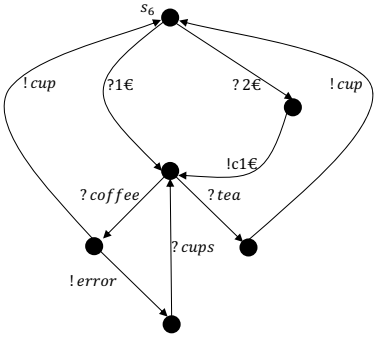
(c)



(d)



(e)



(f)

Fig. 8. Sample MTS Vending Machine Implementations

In [31], Larsen et al. proposed an approach for behavioral variability modeling and formal verification of a product line based on modal transition systems. A product line specification is given as modal LTS s comprising the set of all product variants p of s by means of complete modal refinements $p \leqslant_m s$. To further tailor the set of valid product variants, the notion of *compatibility* defined by a *partial* modal composition operator is used. This way, refinements of variable product line components are implicitly restricted to those which are compatible to other components and/or an environmental specification. Alternatively, the product line modeling theory of Asirelli et al. uses deontic logics to restrict the set of modal refinements [4,3,2].

Here, we limit our considerations to a modal product line test modeling theory with unrestricted modal refinements for product variant derivations. According to the model-based testing assumption of Bernot [7], we assume a product line specification $s \in \mathcal{MTS}(I \cup U)$ and a product line implementation $i \in \mathcal{MTS}(I \cup U)$ which both correspond to a modal input/output labeled transition system. By adopting the notion of input-enabledness to modal specifications, we obtain the following definition.

Definition 13 (Modal I/O Transition System). *An MTS is* γ-input-enabled *iff for every state* $s \in Q$ *with* $q_0 \Longrightarrow_\gamma^* s$ *and for all* $a \in I$ *it holds that* $s \overset{a}{\Longrightarrow}_\gamma$. *By* $\mathcal{IOMTS}_\gamma(I, U)$ *we denote the set of* γ*-input-enabled MTS labeled over act* $= (I \cup U \cup \{\tau\})$ *and conclude that*

$$\mathcal{IOMTS}_\Box(I, U) \subset \mathcal{IOMTS}_\Diamond(I, U) \subset \mathcal{MTS}(I \cup U)$$

holds.

For instance, considering the sample MTS models in Fig. 7, both m_1 and m_2 are neither *may*-input-enabled, nor *must*-input-enabled. Again, γ-input-enabledness for a given MTS may be achieved by adding corresponding transitions $s \overset{a}{\rightarrow}_\gamma s$ for every input action $a \in I$ to every reachable state s with $\nexists s' : s \overset{a}{\Longrightarrow}_\gamma s'$.

Similar to single system testing, we require a product line implementation i as well as all derivable product implementation variants $i' \leqslant_m i$ to be input-enabled. Unfortunately, γ-input-enabledness is not preserved under modal refinement. As an example, consider some state s with $q_0 \Longrightarrow_\gamma^* s$ and $s \overset{a}{\rightarrow}_\gamma$ for every $a \in I$.

- For $\gamma = \Diamond$, assume some transition $s \overset{a}{\rightarrow}_\Diamond s'$ which is removed such that $\neg s \overset{a}{\rightarrow}_\Diamond$ holds after refinement.
- For $\gamma = \Box$, assume some transition $s \overset{a}{\rightarrow}_\Diamond s'$ with $\neg q_0 \Longrightarrow_\Box^* s'$ which is refined to $s \overset{a}{\rightarrow}_\Box s'$ and s' to obstruct *must*-input-enabledness.

To solve this problem, we consider the following assumptions for applying modal LTS as a basis for a product line testing theory.

- Product line implementations i under test are *may*-input-enabled, i.e., $i \in \mathcal{IOMTS}_\Diamond(I, U)$, whereas for product line specification we only require $s \in \mathcal{MTS}(I \cup U)$ as usual.

– Derivations of product implementation variants i' are restricted to those preserving may-input-enabledness denoted $i' \leqslant_m^{\diamond} i$ such that

$$\leqslant_m^{\diamond} \subseteq \mathcal{IOMTS}_{\diamond}(I, U) \times \mathcal{IOMTS}_{\diamond}(I, U) \subset \leqslant_m .$$

As a result, it holds that $i' \in \mathcal{IOMTS}_{\Box}(I, U)$ for every complete refinement i' of i. Similar to the notions of traces and input-enabledness, also the auxiliary definitions for defining input/output conformance relations on LTS are adaptable to MTS as follows.

Definition 14. *Let s be an MTS, $p \in Q$, $P \subseteq Q$, $\sigma \in (I \cup U)^*$, and $\gamma \in \{\Box, \diamond\}$.*

1. $init_\gamma(p) := \{\mu \in (I \cup U) \mid p \xrightarrow{\mu}_\gamma\}$,
2. *p is* may-quiescent, *denoted by $\delta_\diamond(p)$, iff $init_\Box(p) \subseteq I$, p is* must-quiescent, *denoted by $\delta_\Box(p)$, iff $init_\diamond(p) \subseteq I$,*
3. p **after**$_\gamma$ $\sigma := \{q \in Q \mid p \overset{\sigma}{\Longrightarrow}_\gamma q\}$,
4. $Out_\gamma(P) := \{\mu \in U \mid \exists p \in P : p \xrightarrow{\mu}_\gamma\} \cup \{\delta_\gamma \mid \exists p \in P : \delta_\gamma(p)\}$, *and*
5. $Straces_\gamma(p) := \{\sigma' \in (I \cup U \cup \{\delta\})^* \mid p \overset{\sigma'}{\Longrightarrow}_\gamma\}$ *where* $q \xrightarrow{\delta}_\gamma q$ *iff* $\delta_\gamma(p)$.

Again, if not stated otherwise, we assume a given MTS to be implicitly enriched by transitions $q \xrightarrow{\delta}_\gamma q$ for γ-quiescent states q.

Example 8. Considering the sample MTS specifications in Fig. 7 and $\sigma =?1€ \cdot ?tea$ it holds that

– $Out_\diamond(m_1$ **after**$_\diamond$ $\sigma) = \{cup, error\}$
– $Out_\diamond(m_2$ **after**$_\diamond$ $\sigma) = \{cup, error\}$
– $Out_\Box(m_1$ **after**$_\diamond$ $\sigma) = \{\}$
– $Out_\Box(m_2$ **after**$_\diamond$ $\sigma) = \{error\}$
– $Out_\Box(m_1$ **after**$_\Box$ $\sigma) = \{\}$
– $Out_\Box(m_2$ **after**$_\Box$ $\sigma) = \{\}$

whereas for $\sigma' =?2€$

– $Out_\diamond(m_1$ **after**$_\diamond$ $\sigma') = \{c1€, \delta\}$
– $Out_\diamond(m_2$ **after**$_\diamond$ $\sigma') = \{c1€, \delta\}$
– $Out_\Box(m_1$ **after**$_\diamond$ $\sigma') = \{\delta\}$
– $Out_\Box(m_2$ **after**$_\diamond$ $\sigma') = \{c1€\}$
– $Out_\Box(m_1$ **after**$_\Box$ $\sigma') = \{\}$
– $Out_\Box(m_2$ **after**$_\Box$ $\sigma') = \{\}$

holds.

According to the intuition of modal consistency, we observe the following correspondences.

Proposition 1. *Let s be an MTS, $p \in Q$, $P \subseteq Q$, $\sigma \in (I \cup U)^*$.*

1. $init_\Box(p) \subseteq init_\diamond(p)$,
2. $\delta_\Box \subseteq \delta_\diamond$,

3. p **after**$_\Box$ $\sigma \subseteq p$ **after**$_\Diamond$ σ,
4. $Out_\Box(P) \subseteq Out_\Diamond(P)$, *and*
5. $Straces_\Box(p) \subseteq Straces_\Diamond(p)$.

Testing a modal implementation $i \in \mathcal{IOMTS}_\Diamond(I, U)$ against a modal specification $s \in \mathcal{MTS}(I, U)$ aims at verifying that every derivable product implementation variant $i' \leqslant_m^\Diamond i$ conforms to a corresponding product specification variant $s' \leqslant_m s$. For this to hold, an intuitive notion of modal input/output conformance should ensure that

- all *possible* behaviors of a product line implementation are *allowed* and that
- all *mandatory* behaviors of a product line implementation are *required*

by the respective product line specification. Hence, for a model I/O conformance relation i **mior** s to hold, it requires trace inclusion of both *may*-suspension-traces and *must*-suspension-traces, respectively.

However, if we interpret the set of *must*-behaviors specified by s as the product line *core* behavior to be shown by all product variants, this notion of I/O conformance fails to fully capture this intuition. Similar to the non-modal version, suspension trace inclusion solely ensures *some* behavior of the specified behaviors to be actually implemented (if any), but it does not differentiate within the set of allowed behaviors between mandatory and optional ones. To overcome this drawback, we consider an alternative definition for modal I/O conformance, i **mior**$_\leqslant$ s, that is closer to the very essence of modal refinement requiring *alternating* suspension trace inclusions as follows.

Definition 15 (Modal I/O Conformance). *Let* $s \in \mathcal{MTS}(I, U)$ *and* $i \in \mathcal{IOMTS}_\Diamond(I, U)$.

i **mior** s $:\Leftrightarrow$

1. $\forall \sigma \in act_\delta^* : Out_\Diamond(i \textbf{ after}_\Diamond \sigma) \subseteq Out_\Diamond(s \textbf{ after}_\Diamond \sigma)$ *and*
2. $\forall \sigma \in act_\delta^* : Out_\Box(i \textbf{ after}_\Box \sigma) \subseteq Out_\Box(s \textbf{ after}_\Box \sigma)$.

i **mior**$_\leqslant$ s $:\Leftrightarrow$

1. $\forall \sigma \in act_\delta^* : Out_\Diamond(i \textbf{ after}_\Diamond \sigma) \subseteq Out_\Diamond(s \textbf{ after}_\Diamond \sigma)$ *and*
2. $\forall \sigma \in act_\delta^* : Out_\Box(s \textbf{ after}_\Box \sigma) \subseteq Out_\Box(i \textbf{ after}_\Box \sigma)$.

Hence, the **mior**$_\leqslant$ relation requires a product line implementation i to show

- *at least* all mandatory behaviors and
- *at most* the allowed behaviors

of a product line specification s. The respective modal versions of the **ioco** relation can be defined, accordingly.

Definition 16 (Modal IOCO). *Let* $s \in \mathcal{MTS}(I, U)$ *and* $i \in \mathcal{IOMTS}_\Diamond(I, U)$.

i **mioco** s $:\Leftrightarrow$

1. $\forall \sigma \in Straces_{\diamond}(s) : Out_{\diamond}(i \textbf{ after}_{\diamond} \sigma) \subseteq Out_{\diamond}(s \textbf{ after}_{\diamond} \sigma)$ *and*
2. $\forall \sigma \in Straces_{\square}(i) : Out_{\square}(i \textbf{ after}_{\square} \sigma) \subseteq Out_{\square}(s \textbf{ after}_{\square} \sigma)$.

$i \textbf{ mioco}_{\leqslant} s \ :\Leftrightarrow$

1. $\forall \sigma \in Straces_{\diamond}(s) : Out_{\diamond}(i \textbf{ after}_{\diamond} \sigma) \subseteq Out_{\diamond}(s \textbf{ after}_{\diamond} \sigma)$ *and*
2. $\forall \sigma \in Straces_{\square}(i) : Out_{\square}(s \textbf{ after}_{\square} \sigma) \subseteq Out_{\square}(i \textbf{ after}_{\square} \sigma)$.

Based on the previous observations, we conclude the following notion of soundness for modal **ioco**.

Theorem 2 (Soundness). *Let Let* $s \in \mathcal{MTS}(I, U)$, $i \in \mathcal{IOMTS}_{\diamond}(I, U)$ *and* $i \textbf{ mioco}_{\leqslant} s$. *Then it holds that* $\forall i' \leqslant_m^{\diamond} i : i' \textbf{ mioco}_{\leqslant} s$.

Hence, family-based product line conformance testing in terms of the presented intentional characterization of modal I/O conformance ensures (1) safety as it permits implementation variants to only show allowed behaviors and (2) liveness as it enforces implementation variants to at least show all core behaviors. Further results, e.g., concerning completeness and exhaustiveness notions, as well as an extensional characterization of modal **ioco** is open for future work. Concerning the latter, two main adaptations with respect single system testing are required.

1. The modality $\gamma \in \{\diamond, \square\}$ of an action a (including quiescence) occurring at a transition $s \xrightarrow{a}_{\gamma} s'$ is *observable*, e.g., by defining two separate alphabets $act_{\square} = act \times \{\square\}$ and $act_{\diamond} = act \times \{\diamond\}$, respectively.
2. The definition of a test case (cf. Fig. 4) is to be adopted for modal testing to require *every* must-behavior to be observed before giving the verdict **pass**, therefore, potentially requiring multiple test runs.

Clause 2. reflects that verifying the inclusion of all *must*-behaviors of the specification to be contained in the respective set of the implementation literally requires the specification to be (implicitly) tested against the implementation.

5 Conclusion

In this tutorial, we have presented the foundations of model-based testing. We have considered dynamic testing of functional characteristics at component-level based on model-based black-box knowledge for single system testing. This testing discipline is usually referred to as model-based input/output conformance testing. Furthermore, we have presented model-based testing techniques for variant-rich software systems, such as software product lines. We have explained sample-based product line testing based on variant selection techniques and a theory for variability-aware product line conformance testing.

References

1. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating Refinement Relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)
2. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Model-Checking Tool for Families of Services. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 44–58. Springer, Heidelberg (2011)
3. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: Formal Description of Variability in Product Families. In: SPLC 2011, pp. 130–139 (2011)
4. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Logical Framework to Deal with Variability. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 43–58. Springer, Heidelberg (2010)
5. Bauer, S.S., Hennicker, R., Janisch, S.: Interface Theories for (A)synchronously Communicating Modal I/O-Transition Systems. Electronic Proceedings in Theoretical Computer Science 46, 1–8 (2011)
6. Beizer, B.: Software Testing Techniques, 2nd edn. Van Nostrand Reinhold Co., New York (1990)
7. Bernot, G.: Testing against Formal Specifications: A Theoretical View. In: Abramsky, S. (ed.) TAPSOFT 1991, CCPSD 1991, and ADC-Talks 1991. LNCS, vol. 494, pp. 99–119. Springer, Heidelberg (1991)
8. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with IOCO. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
9. Broy, M. (ed.): Model-Based Testing of Reactive Systems: Advanced Lectures, 1st edn. Springer, Heidelberg (2005)
10. Chvatal, V.: A Greedy Heuristic For The Set-Covering Problem. Mathematics of Operations Research (1979)
11. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 425–439. Springer, Heidelberg (2011)
12. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Eng. Addison-Wesley (2001)
13. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and Adequacy in Software Product Line Testing. In: ISSTA, pp. 53–63. ACM (2006)
14. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
15. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley (2000)
16. De Nicola, R.: Extensional Equivalence for Transition Systems. Acta Informatica 24, 211–237 (1987), http://portal.acm.org/citation.cfm?id=25067.25074
17. of Electrical, I., Engineers, E.: IEEE Standard Glossary of Software Engineering Technology 610.121990 (1990)
18. of Electrical, I., Engineers, E.: IEEE Standard for Software Test Documentation IEEE Std 829-1998 (1998)
19. Engström, E.: Exploring Regression Testing and Software Product Line Testing - Research and State of Practice. Lic dissertation, Lund University (May 2010)
20. Engström, E., Runeson, P.: Software Product Line Testing – A Systematic Mapping Study. Information and Software Technology 53(1), 2–13 (2011)

21. Ferber, S., Haag, J., Savolainen, J.: Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, p. 235. Springer, Heidelberg (2002)
22. Fischbein, D., Uchitel, S., Braberman, V.A.: A Foundation for Behavioural Conformance in Software Product Line Architectures. In: Hierons, R.M., Muccini, H. (eds.) ISSTA 2006, pp. 39–48. ACM (2006)
23. Gustafsson, T.: An Approach for Selecting Software Product Line Instances for Testing. In: SPLiT (2007)
24. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Traon, Y.L.: Multi-objective Test Generation for Software Product Lines. In: SPLC (2013)
25. Johansen, M.F., Haugen, O., Fleurey, F.: An Algorithm for Generating t-wise Covering Arrays from Large Feature Models. In: SPLC, pp. 46–55. ACM (2012)
26. Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 638–652. Springer, Heidelberg (2011)
27. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
28. Kim, C.H.P., Batory, D., Khurshid, S.: Reducing Combinatorics in Testing Product Lines. In: AOSD, pp. 57–68. ACM (2011)
29. Kowal, M., Schulze, S., Schaefer, I.: Towards Efficient SPL Testing by Variant Reduction. In: VariComp, pp. 1–6. ACM (2013)
30. Kuhn, D.R., Wallace, D.R., Gallo, J. A.M.: Software fault interactions and implications for software testing. IEEE Trans. Softw. Eng. 30(6), 418–421 (2004)
31. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
32. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: LICS, pp. 203–210 (1988)
33. Lochau, M., Oster, S., Goltz, U., Schürr, A.: Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. Software Quality Journal, 1–38 (2011)
34. Lochau, M., Schaefer, I., Kamischke, J., Lity, S.: Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 67–82. Springer, Heidelberg (2012)
35. Lochau, M., Goltz, U.: Feature Interaction Aware Test Case Generation for Embedded Control Systems. Electron. Notes Theor. Comput. Sci. 264, 37–52 (2010)
36. Lüttgen, G., Vogler, W.: Modal Interface Automata. In: Baeten, J.C.M., Ball, T., de Boer, F.S. (eds.) TCS 2012. LNCS, vol. 7604, pp. 265–279. Springer, Heidelberg (2012)
37. McGregor, J.D.: Testing a Software Product Line. Tech. Rep. CMU/SEI-2001-TR-022, Carnegie Mellon, Software Engineering Inst. (2001)
38. McGregor, J.D., Sodhani, P., Madhavapeddi, S.: Testing Variability in a Software Product Line. In: Proceedings of the Software Product Line Testing Workshop (SPLiT), pp. 45–50. Avaya Labs, Boston (2004)
39. Mendonca, M., Wąsowski, A., Czarnecki, K.: SAT-based Analysis of Feature Models is Easy. In: Proc. Int'l Software Product Line Conference, pp. 231–240 (2009)
40. da, M.S., Neto, P.A., Carmo Machado, I.D., McGregor, J.D., de Almeida, E.S., de Lemos Meira, S.R.: A Systematic Mapping Study of Software Product Lines Testing. Inf. Softw. Technol. 53, 407–423 (2011)

41. Myers, G.J.: The Art of Software Testing. Wiley, New York (1979)
42. Olimpiew, E.M.: Model-Based Testing for Software Product Lines. Ph.D. thesis, George Mason University (2008)
43. van Osch, M.: Hybrid input-output conformance and test generation. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 70–84. Springer, Heidelberg (2006)
44. Oster, S., Lochau, M., Zink, M., Grechanik, M.: Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations. In: 3rd International Workshop on Feature-Oriented Software Development (FOSD) (2011)
45. Oster, S., Zorcic, I., Markert, F., Lochau, M.: MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In: VaMoS (2011)
46. Oster, S., Wübbeke, A., Engels, G., Schürr, A.: Model-Based Software Product Lines Testing Survey. In: Model-based Testing for Embedded Systems. CRC Press Taylor & Francis (2010) (to appear)
47. Perrouin, G., Sen, S., Klein, J., Le Traon, B.: Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In: ICST, pp. 459–468 (2010)
48. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: Modal Interfaces: Unifying Interface Automata and Modal Specifications. In: EMSOFT 2009, pp. 87–96. ACM (2009)
49. Reuys, A., Kamsties, E., Pohl, K., Reis, S.: Model-Based System Testing of Software Product Families. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 519–534. Springer, Heidelberg (2005)
50. Schaefer, I., Rabiser, R., Clarke, D., Bettini, L., Benavides, D., Botterweck, G., Pathak, A., Trujillo, S., Villela, K.: Software diversity: state of the art and perspectives. STTT 14(5), 477–495 (2012)
51. Scheidemann, K.: Verifying Families of System Configurations. Ph.D. thesis, TU Munich (2007)
52. Schmaltz, J., Tretmans, J.: On Conformance Testing for Timed Systems. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 250–264. Springer, Heidelberg (2008)
53. Tevanlinna, A., Taina, J., Kauppinen, R.: Product Family Testing: A Survey. ACM SIGSOFT Software Engineering Notes 29, 12–18 (2004)
54. Tretmans, J.: Testing concurrent systems: A formal approach. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, p. 46. Springer, Heidelberg (1999)
55. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, San Francisco (2007)
56. van Veenendaal, E. (ed.): ISTQB-Glossary-of-Testing-Terms-2-0. Glossary Working Party (2007)