Technische Universität Braunschweig

Model-based Testing – Part 1

Prof. Dr.-Ing. Ina Schaefer – SFM:ESM - Bertinoro - 18 June 2014

Contents

Part 1: Foundations of Testing and Model-based Testing

- **Fundamental Notions and Concepts of Software Testing**
- Model-based Testing
- A Theoretical Perspective on Model-based Testing

Part 2: Model-based Testing of Software Product Lines

- Sample-based Software Product Line Testing
- Regression-based Software Product Line Testing
- Variability-Aware Software Product Line Testing

Testing is …

[…] “an activity performed for evaluating product quality, and for improving it, by identifying defects and problems.”

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

[IEEE, 1990]

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."

[Tretmans, 1999]

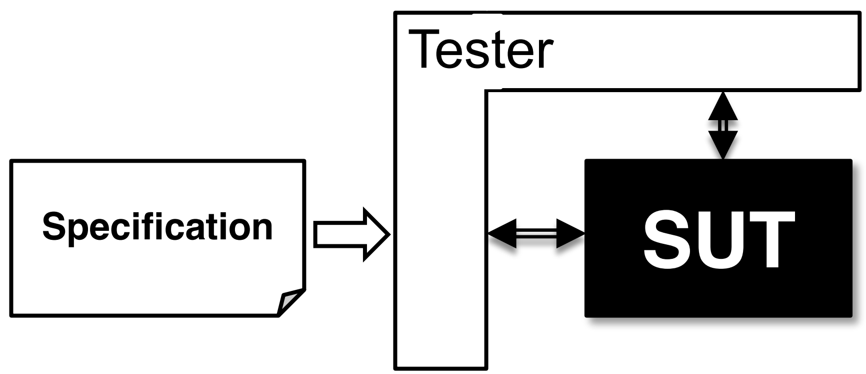
Factors for Testing

test aims

- functional
- non-functional
- robustness
- performance
- reliability

test methods

- static testing:
  e.g. systematic code inspections
- dynamic testing:
  e.g. experimental executions



Factors for Testing

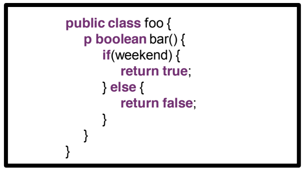
test scale

- unit tests
- component tests
- integration tests
- system tests

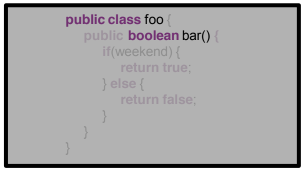
information base

- black box

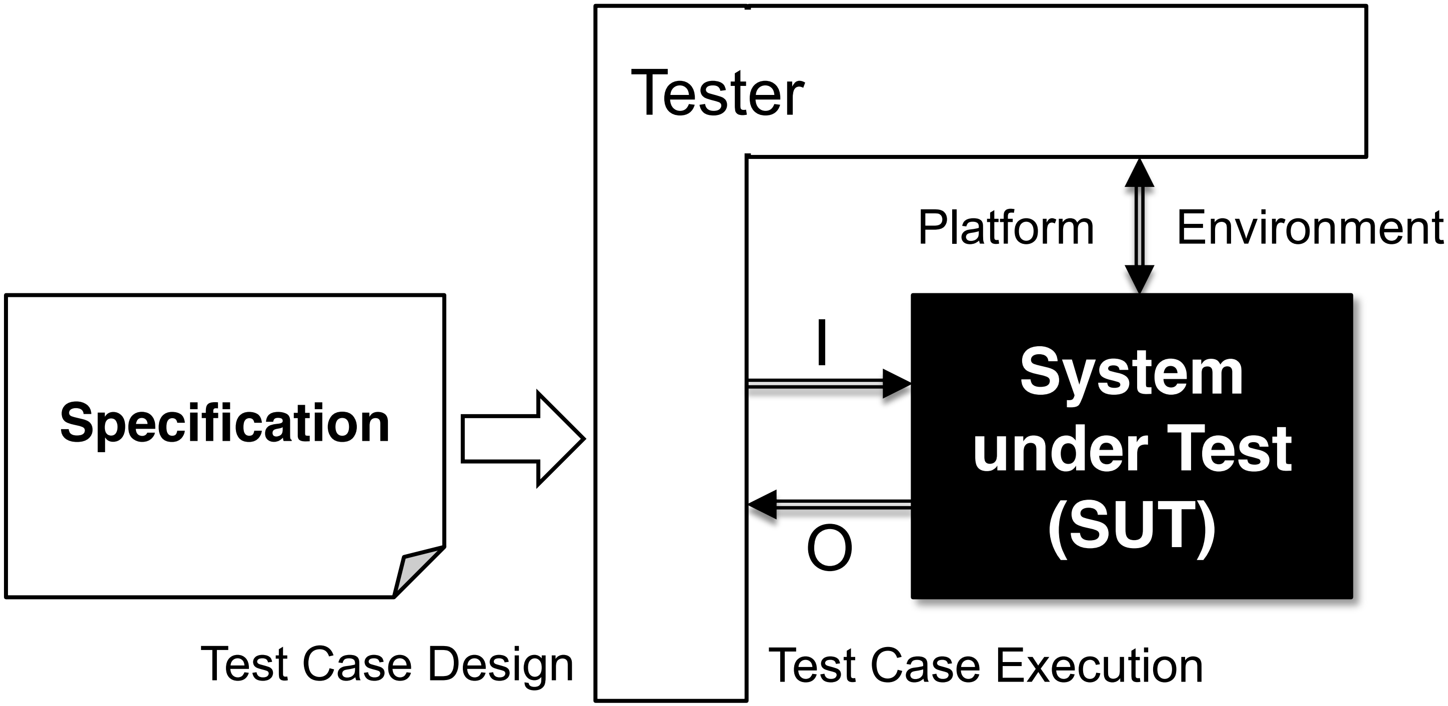
- white box



- grey box



Dynamic Software Testing



Failure, Fault, Error

failure - A failure is an undesired observable behavior of an SUT.

fault – A fault in an SUT causes a failure during test execution.

error – An error is a logical flaw in the implementation.

The Notion of Software Testing used in this Lecture

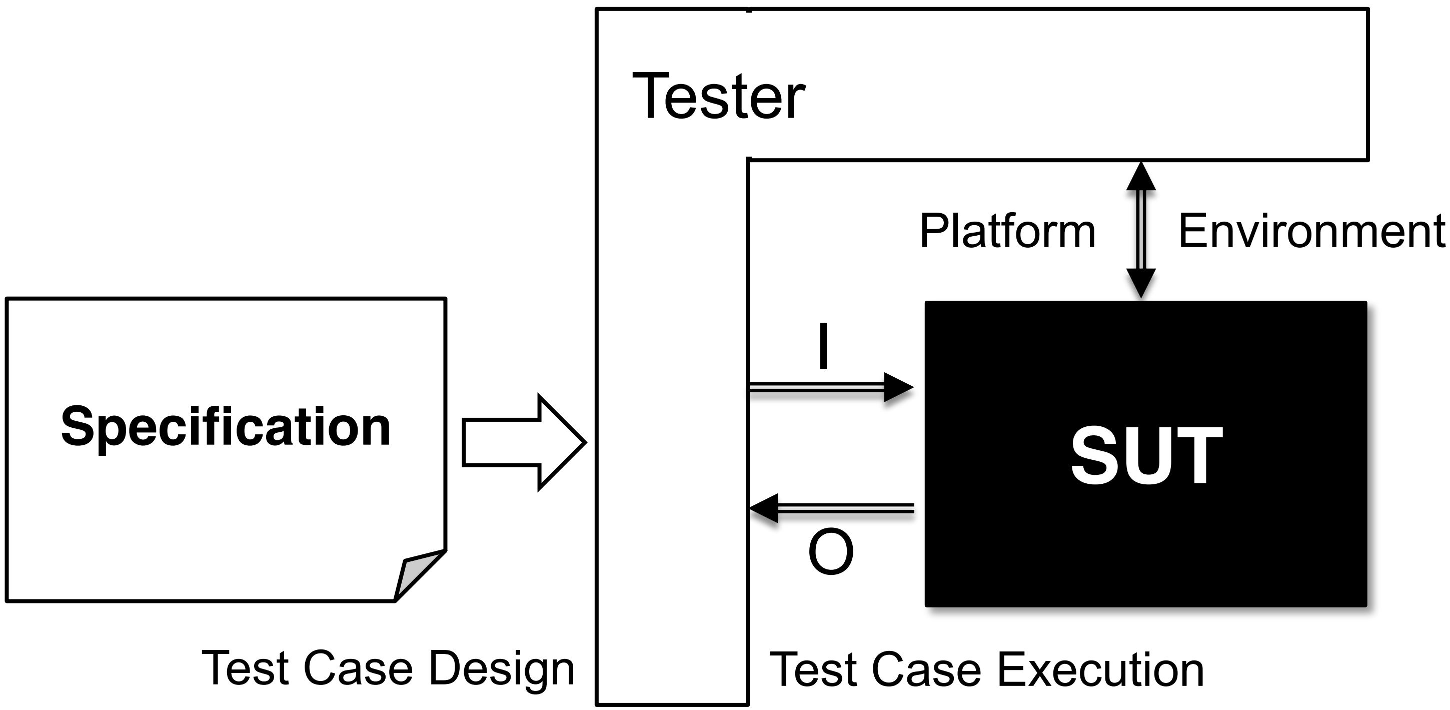
Software testing consists of the **dynamic validation/verification** of the behavior of a program on a **finite set of test cases** suitably selected from the usually infinite **input domain** against the **expected** behavior.

Some Literature on Software Testing

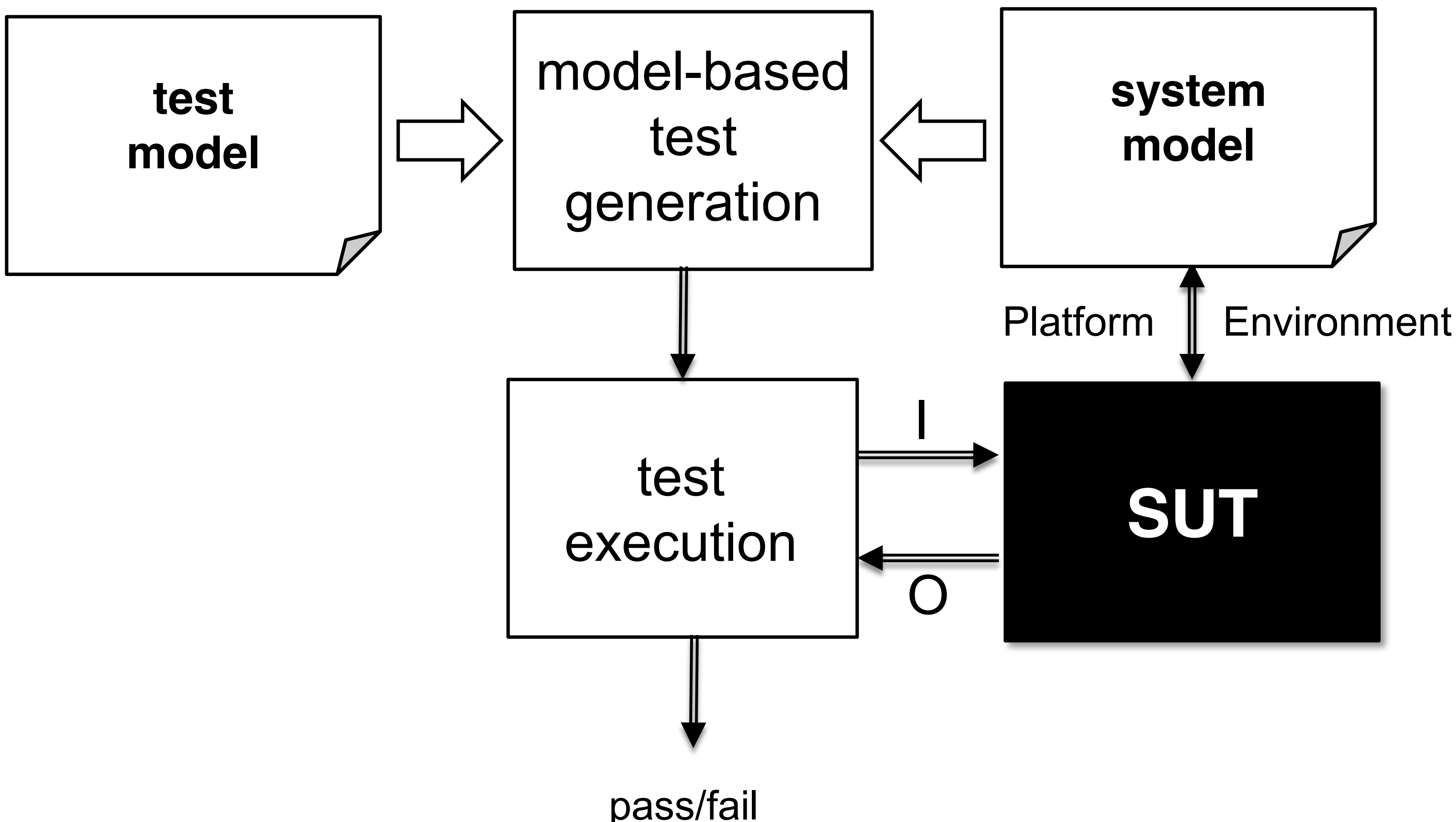
- Myers, G.J.: The Art of Software Testing. Wiley, New York
- Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold Co.
- Broy; M. (ed.): Model-Based Testing of Reactive Systems: Advanced Lectures. Springer, Berlin Heidelberg
- IEEE: Standard Glossary of Software Engineering Technology 610.121990
- IEEE: Standard for Software Test Documentation Std. 829-2008
- van Veenendaal, E. (ed.): ISTQB Glossary of Testing Terms 2.2. Glossary Working Party

Contents

Model-Based Testing

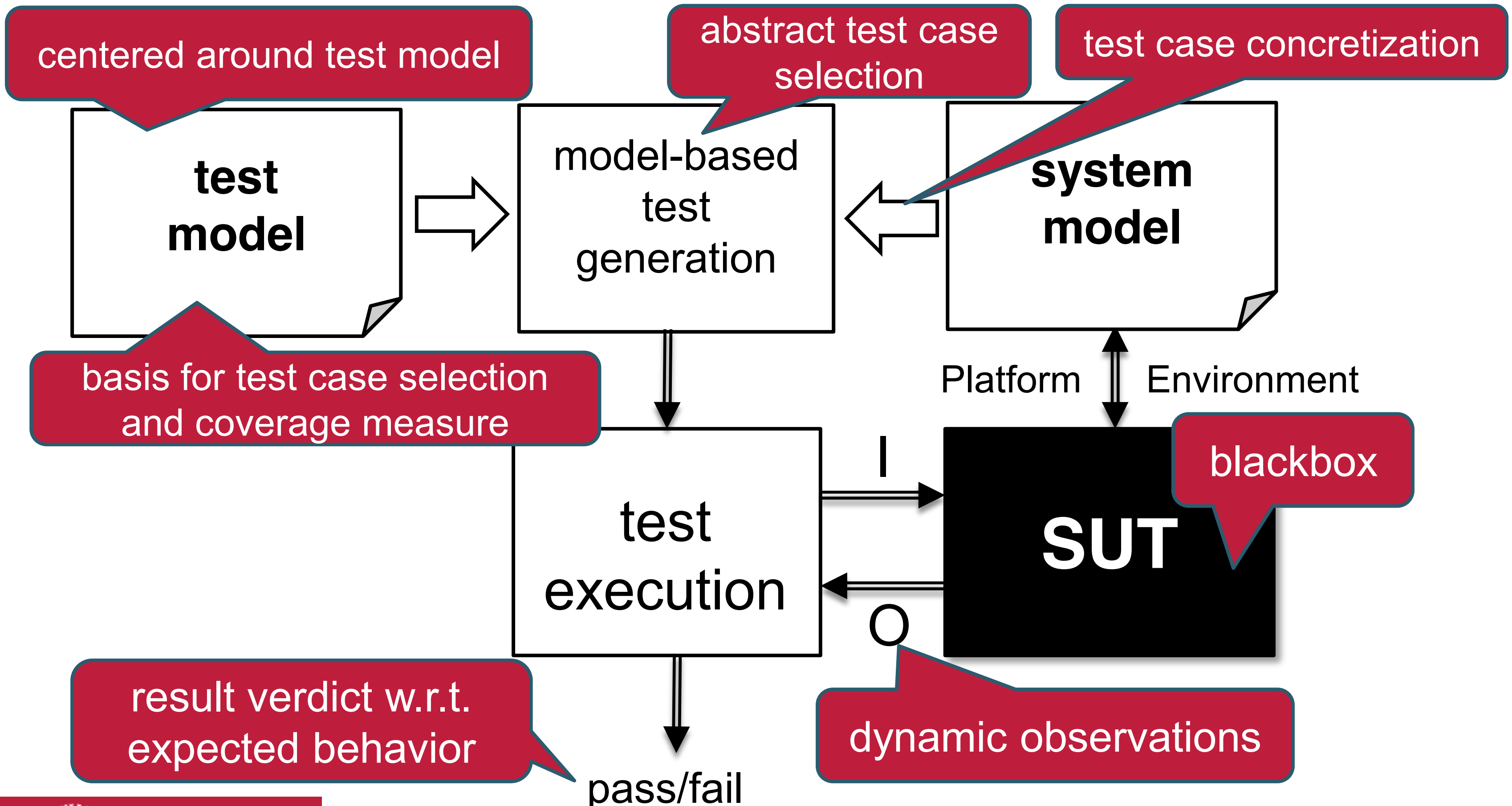


Model-Based Testing



Model-based Testing

Model-based testing is the automation of black box tests.



Contents

MBT from a Theoretical Point of View

implementation relation: $i \simeq s$ with *implemetation i* and *formal behavioral specification s*

preorder relation: $i \sqsubseteq s$ implementation shows at most the behaviors of the specification

intentional conformance: $i\ conforms\ s :\Leftrightarrow [\![i]\!] \subseteq [\![s]\!]$

where $[\![\cdot]\!]$ defines sets of all observable behaviors

extensional conformance: $i\ conforms\ s :\Leftrightarrow \forall u \in \mathcal{U} : obs(u, i) \approx obs(u, s)$

where $\mathcal{U}$ defines sets of all observers

Model-based I/O Conformance Testing

- Proposed by Jan Tretman in the 90's
- Model-based functional conformance testing of systems with reactive, non-deterministic behaviors
- Input, output, and quiescence based testing theory
- Based on I/O labeled transition systems as test models AND implementation models
- Proven sound and exhaustive
- Rich tool support
- Formal basis for many advanced testing frameworks

Testing Concurrent Systems: A Formal Approach

Jan Tretmans

University of Twente *
Faculty of Computer Science, Formal Methods and Tools research group
P.O. Box 217, 7500 AE Enschede, The Netherlands
tretmans@cs.utwente.nl

Abstract. This paper discusses the use of formal methods in testing of concurrent systems. It is argued that formal methods and testing can be mutually profitable and useful. A framework for testing based on formal specifications is presented. This framework is elaborated for labelled transition systems, providing formal definitions of conformance, test execution and test derivation. A test derivation algorithm is given and its tool implementation is briefly discussed.

1 Introduction

During the last decades much theoretical research in computing science has been devoted to formal methods. This research has resulted in many formal languages and in verification techniques, supported by prototype tools, to verify properties of high-level, formal system descriptions. Although these methods are based on sound mathematical theories, there are not many systems developed nowadays for which correctness is completely formally verified using these methods.

On the other hand, the current practice of checking correctness of computing systems is based on a more informal and pragmatic approach. Testing is usually the predominant technique, where an implementation is subjected to a number of tests which have been obtained in an ad-hoc or heuristic manner. A formal, underlying theory for testing is mostly lacking.

The combination of testing and formal methods is not very often made. Sometimes it is claimed that formally verifying computer programs would make testing superfluous, and that, from a formal point of view, testing is inferior as a way of assessing correctness. Also, some people cannot imagine how the practical, operational, and 'dirty-hands' approach of testing could be combined with the mathematical and 'clean' way of verification using formal methods. Moreover, the classical biases against the use of formal verification methods, such as that formal methods are not practical, that they are not applicable to any real system

* This research is supported by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms; URL: http://fmt.cs.utwente.nl/CdR.

Jos C.M. Baeten, Sjouke Mauw (Eds.): CONCUR'99, LNCS 1664, pp. 46-[illegible] 1999.

Running Example

Beverage vending machine

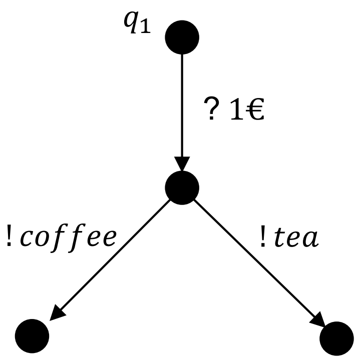
- Input actions
  - $I = \{1€, 2€\}$
  - Transitions labels prefixed with “?”
- Output actions
  - $U = \{coffee, tea\}$
  - Transition labels prefixed with “!”

I/O-Labeled Transition Systems

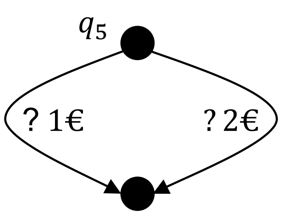
I/O Labeled Transitionsystem: $(\boldsymbol{Q}, \boldsymbol{q_0}, \boldsymbol{I}, \boldsymbol{U}, \rightarrow)$, where

- $\boldsymbol{Q}$ is a countable set of states,
- $\boldsymbol{q_0} \in \boldsymbol{Q}$ is the initial state,
- $\boldsymbol{I}$ and $\boldsymbol{U}$ are disjoint sets of input actions and output actions, and
- $\rightarrow \subseteq \boldsymbol{Q} \times \boldsymbol{act} \times \boldsymbol{Q}$ is a labeled transition relation.

LTS - Examples



$$Tr(q_1) = \{?\,1€, ?\,1€ \cdot !\,coffee, ?\,1€ \cdot !\,tea\}$$



$$Tr(q_5) = \{?\,1€, ?\,2€\}$$

LTS Trace Semantics

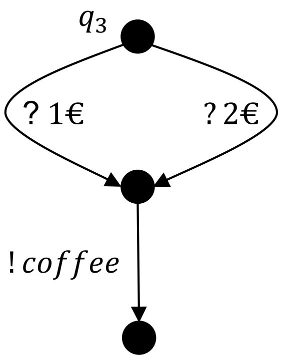
Each compution refers to some path

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

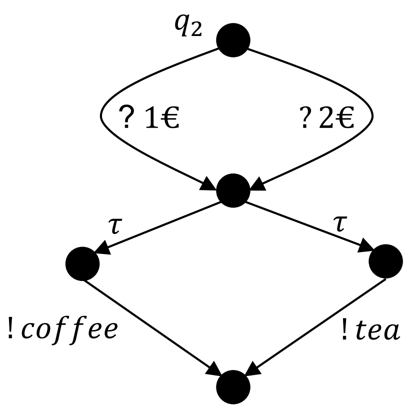
The behavior of a computation is defined by a trace

$$trace\ \sigma = \mu_1 \mu_2 \cdots \mu_n \in act^*$$

LTS - Examples



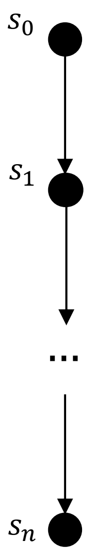
$$Tr(q_3) = \{?\,1€, ?\,2€, ?\,1€ \cdot !\,coffee, ?\,2€ \cdot !\,coffee\}$$



$$Tr(q_2) = \{?\,1€, ?\,2€, ?\,1€ \cdot !\,coffee, ?\,1€ \cdot !\,tea, ?\,2€ \cdot !\,coffee, ?\,2€ \cdot !\,tea\}$$

LTS Trace Notations

- Let **s** be an **I/O** $\boldsymbol{\mathcal{LTS}}$, $\boldsymbol{\mu_i} \in \boldsymbol{I} \cup \boldsymbol{U} \cup \{\boldsymbol{\tau}\}$ and $\boldsymbol{a_i} \in \boldsymbol{I} \cup \boldsymbol{U}$



$$s \xrightarrow{\mu_1 \cdots \mu_n} s' := \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s_n = s'$$

$$s \xrightarrow{\mu_1 \cdots \mu_n} := \exists s' : s = s \xrightarrow{\mu_1 \cdots \mu_n} s'$$

$$\neg s \xrightarrow{\mu_1 \cdots \mu_n} := \nexists s' : s \xrightarrow{\mu_1 \cdots \mu_n} s'$$

LTS Trace Notations

Let **s** be an **I/O** $\boldsymbol{\mathcal{LTS}}$, $\boldsymbol{\mu_i} \in \boldsymbol{I} \cup \boldsymbol{U} \cup \{\boldsymbol{\tau}\}$ and $\boldsymbol{a_i} \in \boldsymbol{I} \cup \boldsymbol{U}$

$$s \stackrel{\epsilon}{\Rightarrow} s' \coloneqq s = s' or\ s \xrightarrow{\tau\cdots\tau} s'$$

$$s \stackrel{a}{\Rightarrow} s' \coloneqq \ \exists s_1, s_2 : s \stackrel{\epsilon}{\Rightarrow} s_1 \stackrel{a}{\rightarrow} s_2 \stackrel{\epsilon}{\Rightarrow} s'$$

$$s \xRightarrow{a_1\cdots a_n} s' \coloneqq \ \exists s_0, \dots, s_n : s = s_0 \stackrel{a_1}{\Rightarrow} s_1 \stackrel{a_2}{\Rightarrow} \dots \stackrel{a_n}{\Rightarrow} s_n = s'$$

Technische Universität Braunschweig

Institut für Softwaretechnik und Fahrzeuginformatik

LTS Trace Notations

Let **s** be an **I/O** $\boldsymbol{\mathcal{LTS}}$, $\boldsymbol{\sigma} \in (\boldsymbol{I} \cup \boldsymbol{U})^*$

$$s \overset{\sigma}{\Rightarrow} := \exists s' : = \exists s' :\ s \overset{\sigma}{\Rightarrow} s'$$

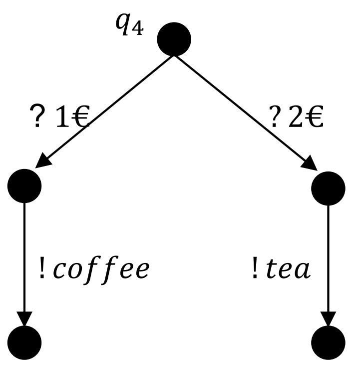
$$\neg s \overset{\sigma}{\Rightarrow} :=\ \nexists s\ ' : s \overset{\sigma}{\Rightarrow} s'$$

LTS Trace Notations

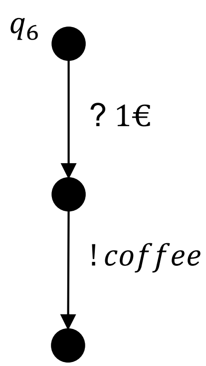
The set of traces in an $\mathcal{LTS}$ is defined as

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

LTS - Examples

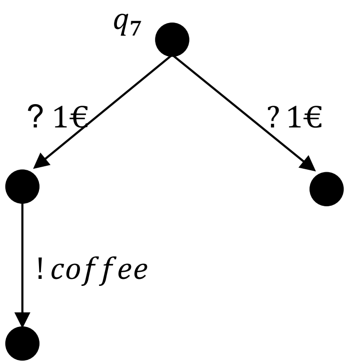


$$Tr(q_4) = \{?\,1€, ?\,2€, ?\,1€ \cdot !\,coffee, ?\,2€ \cdot !\,tea\}$$

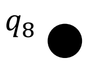


$$Tr(q_6) = \{?\,1€, ?\,1€ \cdot !\,coffee\}$$

LTS - Examples



$$Tr(q_7) = \{?\,1€, ?\,1€ \cdot !\,coffee\}$$

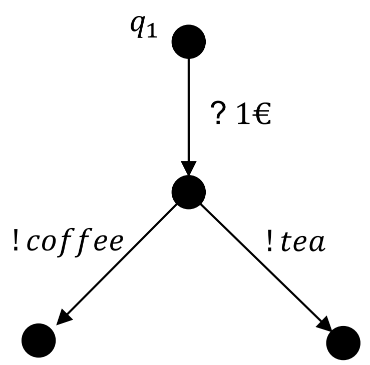


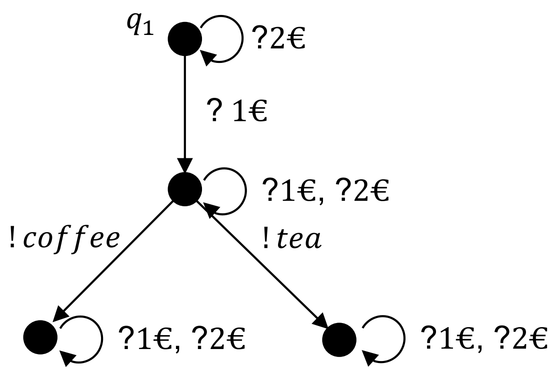
$$Tr(q_8) = \{\}$$

Input-Enabled Transition Systems

An $\mathcal{LTS}$ is (weak) input-enabled iff for every state $s \in Q$ with $q_0 \Rightarrow^* s$ and for all $a \in \mathrm{I}$ it holds that $s \overset{a}{\Rightarrow}$.

Input Completion - Example





Not input-enabled LTS

Input-enabled LTS

A First Attempt: Conformance as Trace Inclusion

$$i\ \boldsymbol{conforms}\ s :\Leftrightarrow Tr(i) \subseteq Tr(s)$$

Solution: explicit notion of quiescent behavior

q_8

- Fails to refuse trivial implementations

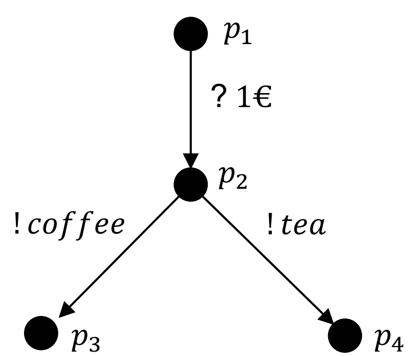
- Fails to take the asymetric nature of $\mathcal{LTS}$ traces with I/O actions into account

Solution: distinguish input and output behaviors in traces

Some Auxiliary Definitions: Init Sets

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

$$init(p) \coloneqq \{\mu \in (I \cup U) | p \xrightarrow{u} \}$$

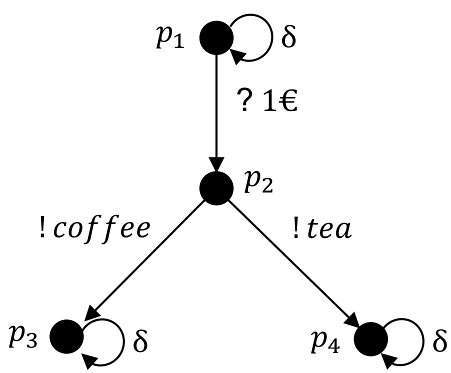


$$init(p_1) \coloneqq \{?\,1€\}$$
$$init(p_2) \coloneqq \{!\,coffee, !\,tea\}$$
$$init(p_3) \coloneqq \{\quad\}$$
$$init(p_4) \coloneqq \{\quad\}$$

Some Auxiliary Definitions: Quiescent States

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

$$p\ is\ \boldsymbol{quiescent}, denoted\ \delta(p), iff\ init(p) \subseteq I$$



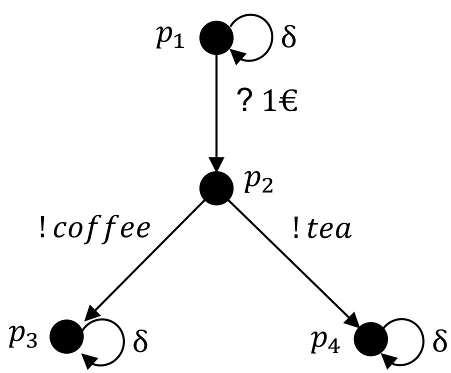
$$\boxed{init(p_1) \coloneqq \{?\,1€\}}$$
$$init(p_2) \coloneqq \{!\,coffee, !\,tea\}$$
$$\boxed{init(p_3) \coloneqq \{\quad\}}$$
$$\boxed{init(p_4) \coloneqq \{\quad\}}$$

$$I = \{?\,1€, ?\,2€\}$$

Some Auxiliary Definitions: After Sets

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

$$p\ \boldsymbol{after}\ \sigma \coloneqq \{q \in U \mid p \overset{\sigma}{\Rightarrow} q\}$$



$$U = \{!\,coffee, !\,tea\}$$

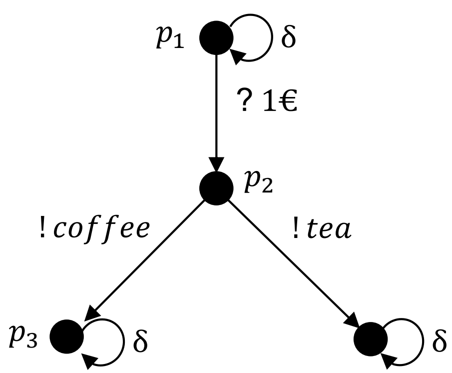
$$p_2\ \boldsymbol{after}\ !\,coffee = \{p_3\}$$
$$p_2\ \boldsymbol{after}\ !\,tea = \{p_4\}$$

$$p_1\ \boldsymbol{after}\ ?\,2€ = \{\quad\}$$
$$p_4\ \boldsymbol{after}\ !\,tea = \{\quad\}$$

Some Auxiliary Definitions: Out Sets

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

$$out(P) \coloneqq \{\mu \in U \mid \exists p \in P : p \xrightarrow{\mu}\} \cup \{\delta \mid \exists p \in P : \delta(p)\}$$



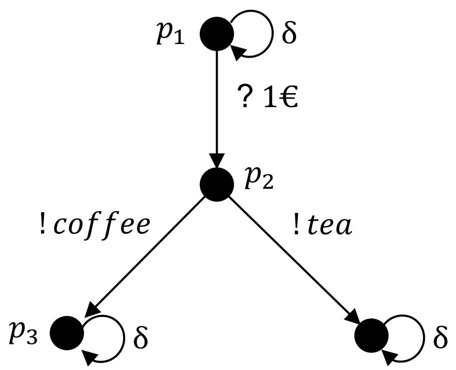
$$P = p\ \boldsymbol{after}\ \sigma$$

$$P_1 = \{p_1\ \boldsymbol{after}\ \delta\} = \{p_1\}$$
$$P_2 = \{p_2\ \boldsymbol{after}\ !tea, p_2\ \boldsymbol{after}\ !coffee\} = \{p_3, p_4\}$$
$$P_3 = \{p_3\ \boldsymbol{after}\ \delta\} = \{p_3\}$$
$$P_4 = \{p_4\ \boldsymbol{after}\ \delta\} = \{p_4\}$$

Some Auxiliary Definitions: After-Out Sets

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

$$out(P) \coloneqq \{\mu \in U \mid \exists p \in P : p \xrightarrow{\mu}\} \cup \{\delta \mid \exists p \in P : \delta(p)\}$$



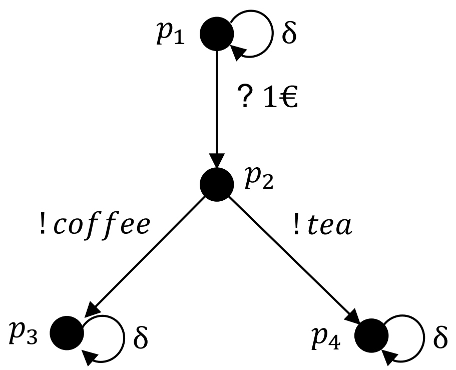
$$P_1 = \{p_1\ \boldsymbol{after}\ \delta\} = \{p_1\}$$
$$P_2 = \{p_2\ \boldsymbol{after}\ !tea, p_2\ \boldsymbol{after}\ !coffee\} = \{p_3, p_4\}$$
$$P_3 = \{p_3\ \boldsymbol{after}\ \delta\} = \{p_3\}$$
$$P_4 = \{p_4\ \boldsymbol{after}\ \delta\} = \{p_4\}$$

$$Out(P_1) = \{\delta\}$$
$$Out(P_2) = \{!tea, !coffee\}$$
$$Out(P_3) = \{\delta\}$$
$$Out(P_4) = \{\delta\}$$

Some Auxiliary Definitions: Suspension Traces

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

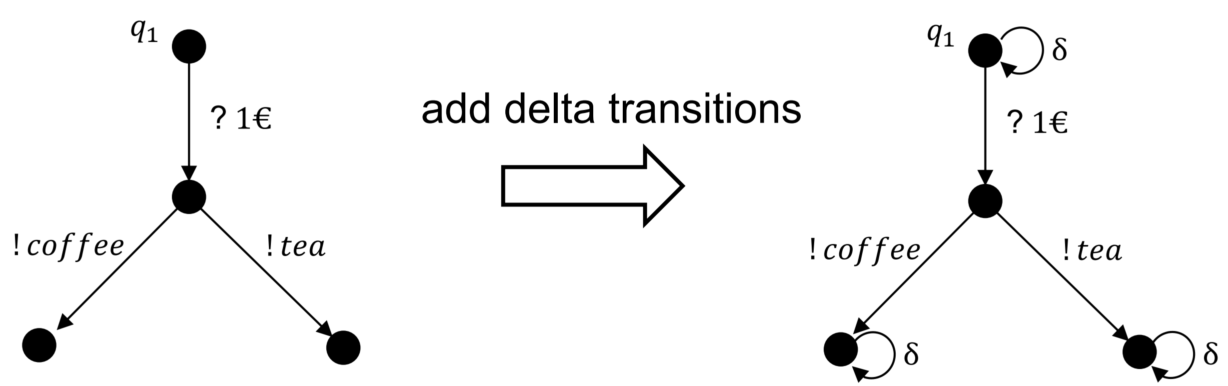
$$Straces(p) \coloneqq \{\sigma' \in (I_S \cup U_S \cup \{\delta\})^* \mid p \overset{\sigma'}{\Rightarrow}\}\ where\ q \overset{\delta}{\rightarrow} q\ iff\ \delta(p)$$



$$Straces(p_1) = \{\delta, ?1€, ?1€ \cdot !coffee, ?1€ \cdot !tea, ?1€ \cdot !coffee \cdot \delta, ?1€ \cdot !tea \cdot \delta\}$$
$$Straces(p_2) = \{!coffee, !tea, !coffee \cdot \delta, tea \cdot \delta\}$$
$$Straces(p_3) = \{\delta\}$$
$$Straces(p_4) = \{\delta\}$$

Quiescent Behaviors

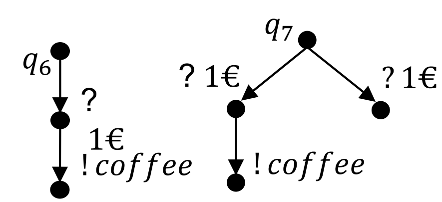
Trace $Tr(q_1)$



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

Allows to discriminate (non-)behaviors

- $?\,1€ \cdot \delta \notin Straces(q_6), whereas\ ?\,1€ \cdot \delta \in Straces(q_7)$
- …



Second Attempt: I/O Conformance (IOR)

Class of I/O LTS labeled over I and U

Subclass of input-enabled I/O LTS

$$Let\ s \in \mathcal{LTS}(I \cup U)\ and\ i \in \mathcal{IOTS}(I, U).$$

$$i\ \boldsymbol{ior}\ s \iff \forall \sigma \in act_{\delta}^{*} : out(i\ \boldsymbol{after}\ \sigma) \subseteq out(s\ \boldsymbol{after}\ \sigma)$$

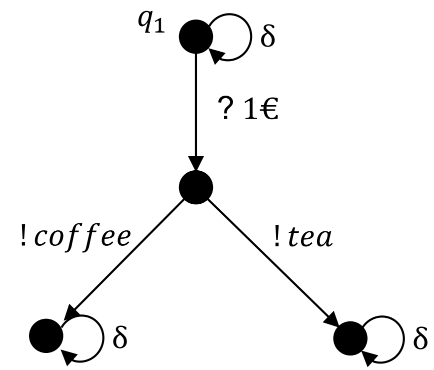
$$i\ \boldsymbol{ior}\ s \iff Straces(i) \subseteq Straces(s)$$

Example

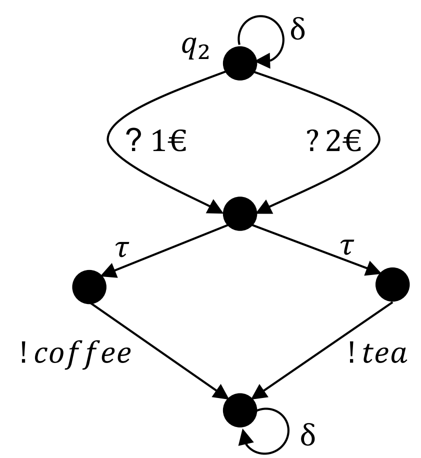
- Assume δ-transitions in the $\mathcal{LTS}$ example
- Possible environmental stimulations are $\sigma = ?1€$ and $\sigma' = ?2€$
- Investigate the observable behavior

Example

$\text{out}(\text{q}_1 \text{ after } \sigma) = \{coffee, tea\}, out(q_1 after\ \sigma') = \{\}$

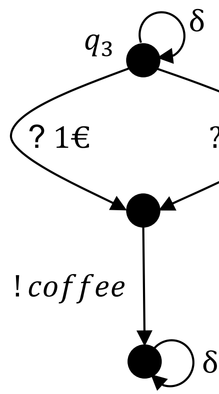


$out(q_2\ after\ \sigma) = \{coffee, tea\}, out(q_2 after\ \sigma') = \{coffee, tea\}$

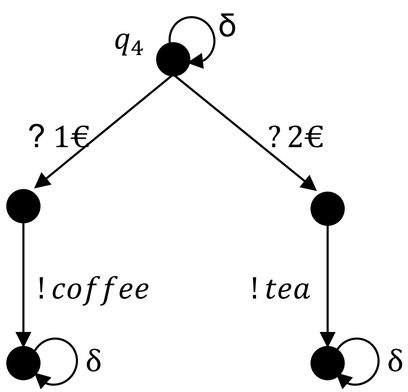


Example

$out(q_3\ after\ \sigma) = \{coffee\}, out(q_3\ after\ \sigma') = \{coffee\}$

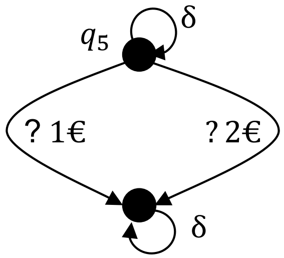


$out(q_4\ after\ \sigma) = \{coffee\}, out(q_4\ after\ \sigma') = \{tea\}$

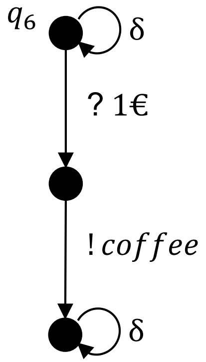


Example

$out(q_5\ after\ \sigma) = \{\delta\}, out(q_5\ after\ \sigma') = \{\delta\}$

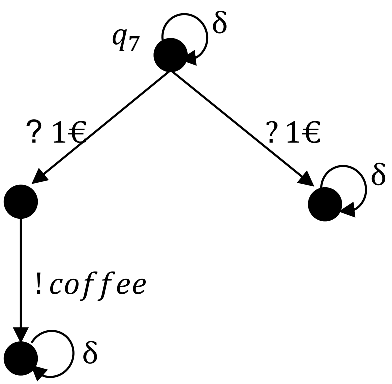


$out(q_6\ after\ \sigma) = \{coffee\}, out(q_6\ after\ \sigma') = \{\}$

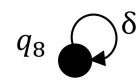


Example

$out(q_7\ after\ \sigma) = \{coffee, \delta\}, out(q_7\ after\ \sigma') = \{\}$



$out(q_8\ after\ \sigma) = \{\}, out(q_8\ after\ \sigma') = \{\}$



Second Attempt: I/O Conformance (IOR)

$$Let\ s \in \mathcal{LTS}(I \cup U)\ and\ i \in \mathcal{IOTS}(I, U).$$

Problem: this is quite a lot!

$$i\ \boldsymbol{ior}\ s \iff \forall \sigma \in act_\delta^* : out(i\ \boldsymbol{after}\ \sigma) \subseteq out(s\ \boldsymbol{after}\ \sigma)$$

$$i\ \boldsymbol{ior}\ s \iff Straces(i) \subseteq Straces(s)$$

Third Attempt: IOCO

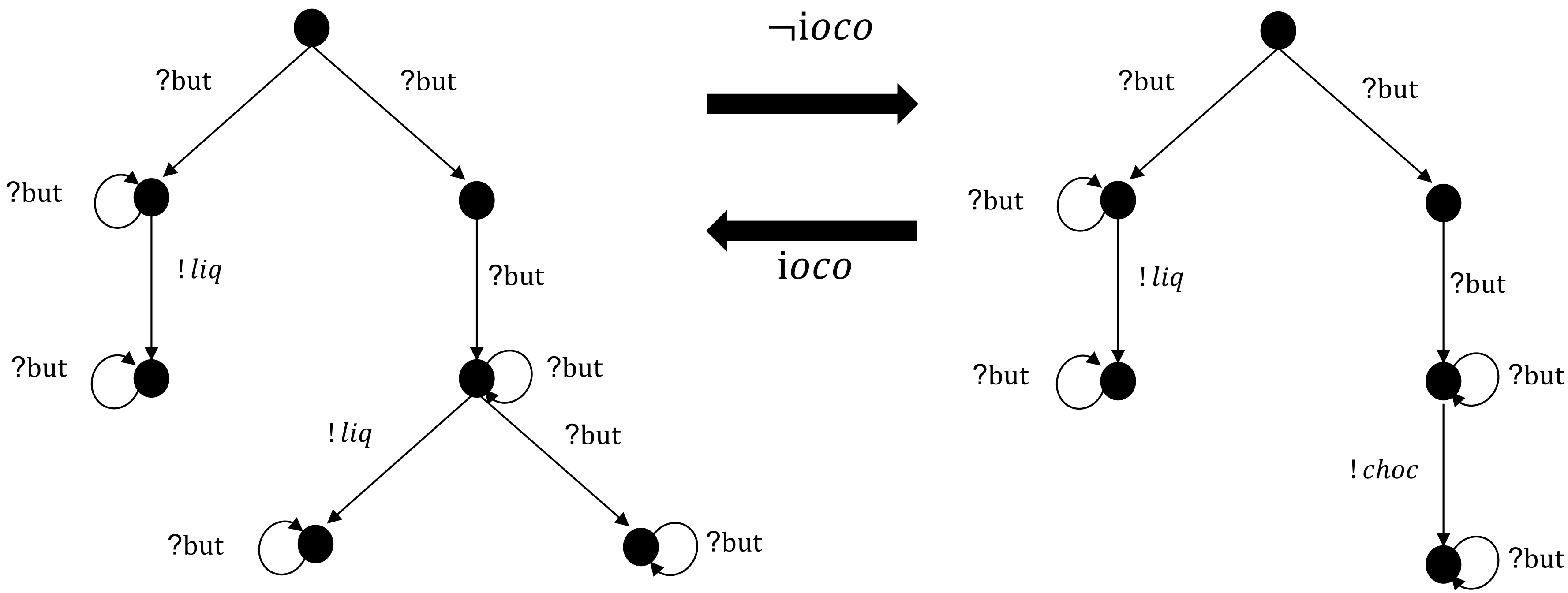
$$Let\ s \in \mathcal{LTS}(I \cup U)\ and\ i \in \mathcal{IOTS}(I, U).$$

Focus on specified behaviors only

$$i\ \boldsymbol{ioco}\ s \iff \forall \sigma \in Straces(s) : out(i\ \boldsymbol{after}\ \sigma) \subseteq out(s\ \boldsymbol{after}\ \sigma)$$

$$\boldsymbol{ior} \subset \boldsymbol{ioco}$$

Example [Tretmans, 1999]



Third Attempt: IOCO

$Let\ s \in \mathcal{LTS}(I \cup U)\ and\ i \in \mathcal{IOTS}(I, U).$

Still infinite in case of loops

$$i\ \boldsymbol{ioco}\ s \iff \forall \sigma \in Straces(s) : out(i\ \boldsymbol{after}\ \sigma) \subseteq out(s\ \boldsymbol{after}\ \sigma)$$

$$\boldsymbol{ior} \subset \boldsymbol{ioco}$$

$\mathrm{IOCO}_{\mathcal{F}}$

- The set of suspension traces under consideration is restricted to sub sets $\mathcal{F} \subseteq act_{\delta}^{*}$
- The restricted ioco relation is denoted as

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

where $\boldsymbol{ior} = \boldsymbol{ioco}_{act_{\delta}^{*}}$ and $\boldsymbol{ior} = \boldsymbol{ioco}_{Straces(s)}$ holds.

This is still an intentional characterization of conformance. How to prove this by testing?

Extensional IOCO

$$i\ \boldsymbol{passes}\ t :\Leftrightarrow obs(i,t) \approx obs(s,t)$$

Observers (testers) are characterized by a finite sets of test cases they perform on an SUT

Test Cases

A test case **t** is an I/O labeled $\mathcal{LTS}$ such that

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

  holds.

denotes observation of quiescence

By $\mathcal{TEST}$ we denote the subclass of I/O labeled $\mathcal{LTS}$ representing valid test cases **t**

Example

Test case for specification q_1

Stimulated input

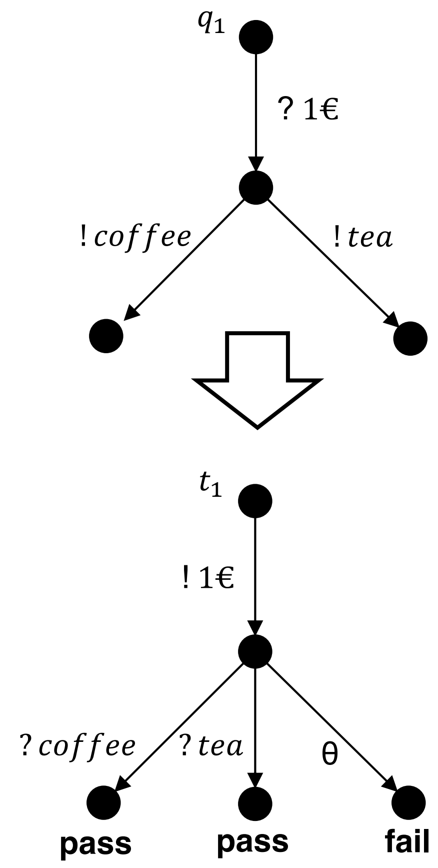
- *!1€*

Expected output

- *either coffee*
- *or tea*

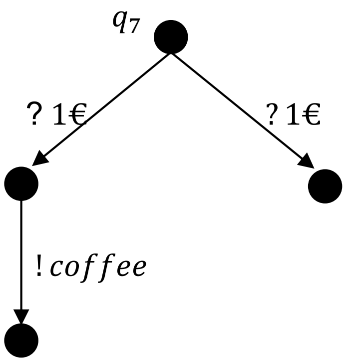
Observable errors

- No output occurs: Θ



Example

Test case for specification q_7



Stimulated input

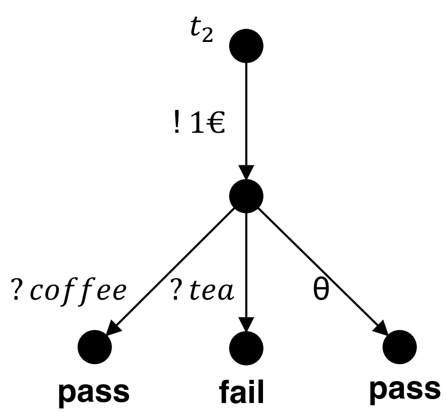
- *!1€*

Expected output

- *coffee*
- no output: Θ

Observable errors

- *tea*



IOCO is correct = sound + exhaustive

Let $s \in \mathcal{LTS}(I \cup U), i \in \mathcal{IOTS}(I \cup \mathrm{U})$ and $\mathcal{F} \subseteq Straces(s)$

Then it holds that

1. the set $\mathcal{TEST}$ of all derivable test cases is **sound** and
2. the set $\mathcal{TEST}$ of all derivable test cases is **exhaustive**.

Contents

Some Further Readings

- Van der Bijn, M.,Rensink, A., Tretmans, J.: Compositional Testing with IOCO, FATES‘04. vol. 2931, pp. 86-100, Springer, 2004
- Schmaltz, J., Tretmans, J.: On Conformance Testing for Timed Systems. FMORMATS‘08. vol. 5215, pp. 250-164, Springer, 2012
- Van Osch, M.: Hybrid Input-output Conformance and Test Generation. FATES‘06. vol. 4262, pp. 70-84, Springer, 2006
- ….

Technische Universität Braunschweig

Model-based Testing – Part 2

Prof. Dr.-Ing. Ina Schaefer – SFM:ESM - Bertinoro - 18 June 2014

Contents

Challenges of Testing variant-rich Software Systems

Observations:

- Complex systems with many interacting functions and features
- Many system variants and versions
- Large rate of changes, in particular in agile development processes

Consequences:

- Increasing testing effort
- Combinatorial explosion during integration and system testing
- Complete re-test in case of changes mostly infeasible

Describing and Managing Variant-rich Systems

Describing and Managing variant-rich Systems

- Variant-rich systems can be described as **Software Product Lines.**

- **SPLs** are systems, which have commonalities and variabilities between each other.

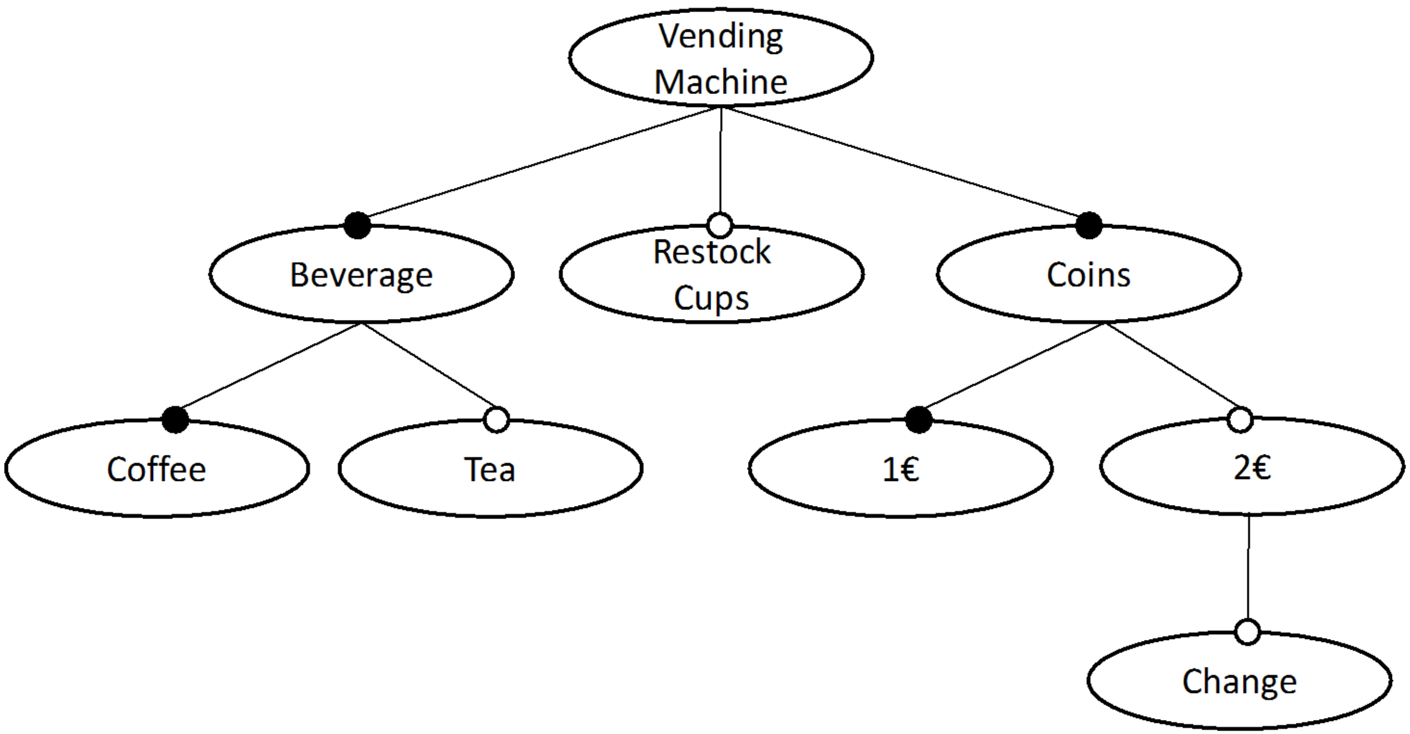
- A SPL consists of several **features** which are either **mandatory** or **optional.**

- There can be further constraints between features
  - Feature A excludes feature B
  - Feature A requires feature B
  - Feature A OR feature B has to be selected

- How to describe and manage these features and there connections?

Feature Models

- Kang et al. [Kang90] introduced **Feature-Models** as possibility to represent SPLs
- FMs are **tree**-structures, which represent features and their dependencies



Feature Interactions

- A **feature** is a customer-visible product characteristic.
- Each feature in isolation satisfies its specification.
- If features are combined, the single specifications are violated. There are unwanted side effects.

→ **Feature Interaction**!

Example: Combine Fire and Water Alarms

If there is fire, start sprinkling system.

If there is water, cut the main water line.

Reasons for Feature Interactions

Intended Feature Interactions:

- Communication via shared variables: one feature writes, another feature reads values.

Unintended Feature Interactions:

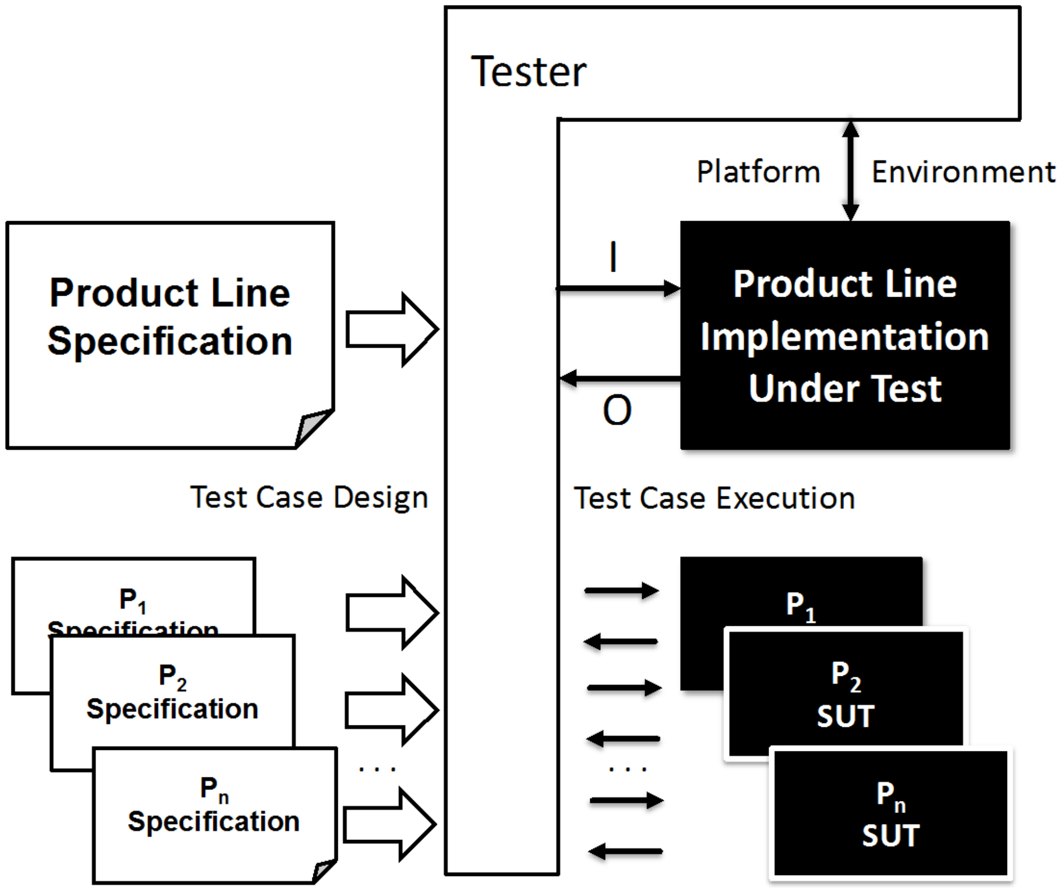
- Non-synchronized write access to shared resources, such as actuators, memory, shared variables, status flags

In general, **uncritical:**

- Shared read access to resources, e.g., sensors

SPL Testing Strategies

Software Product Line Testing



SPL Testing Strategies

Sample-based SPL testing

- Selection of representative subsets from a large set of possible variants

Regression-based SPL Testing:

- Reuse test cases and test results in order to efficiently test the selected variants

Family-based SPL Testing:

- Derive test suite from a 150%-SPL test model

Sample-based SPL Testing

Process of Sample-based SPL Testing

- **Problem:** Number of test cases growths exponentially
- **Solution:** Combinatorial Interaction Testing (CIT)

  1. Create 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

- Efficiency of t-wise Covering Arrays (CA)
  - ➢ 1-wise CA: 50% of all errors
  - ➢ 2-wise CA: 75% of all errors → Trade-Off
  - ➢ 3-wise CA: 95% of all errors

Set Covering Problem and CAs

- S = {a,b,c,d,e} SPL features
- M = {{a,b,c}, {b,d}, {c,d}, {d,e}} valid product configurations
- What is the optimal Covering Array?
- **Solution:** $L = M_1 + M_4$ minimal CA
- **Precondition:** All valid product configurations already known
  - SAT-problem, which is NP-complete
  - Fortunately, we deal with realistic FMs
- Foundation of pairwise testing

First Solution by Chvátal (1979)

- **Idea of the algorithm:**

  1. $Set\ L = \emptyset$
  2. $If\ M_i = \emptyset, \forall\, i, i \in \{1, 2, \dots, n\}\ END.$
     $ELSE\ find\ M', where\ \#\ of\ uncovered\ elements\ is\ max$
  3. $Add\ M'\ to\ L\ and\ replace\ elements\ in\ M_i\ by\ M_i - M'$
  4. $Goto\ Step\ 2$

- **Worst Case:** M contains only subsets with different elements
- Best solution not guaranteed
- Adaptation for pairwise CA generation is easy!

Adaptation to FMs and Improvements by the ICPL

```
input : arbitrary FM
output: t-wise covering array

S ← all t-tuples
while S ≠ ∅ do
    k ← new and empty configuration
    counter ← 0
    foreach tuple p in S do
        if FM is satisfiable with k ∪ p then
            k ← k ∪ p
            S ← S\ {p}
            counter ← counter + 1
        end
    end
    if counter > 0 then
        L ← L ∪ (FM satisfy with {k})
    end
    if counter < # of features in FM then
        foreach tuple p in S do
            if FM not satisfiable with p then
                S ← S\ {p}
            end
        end
    end
end
```

- **Adaptation is still slow in computation!**
- **(Selected) Improvements**
  - Finding core and dead features quickly
  - Early identification of invalid t-sets
  - Parallelization
  - and several more

Vending Machine and ICPL runtimes

- VM has 12 valid variants
- t = 2, ICPL calculates CA of size 6
- 50% testing time saved

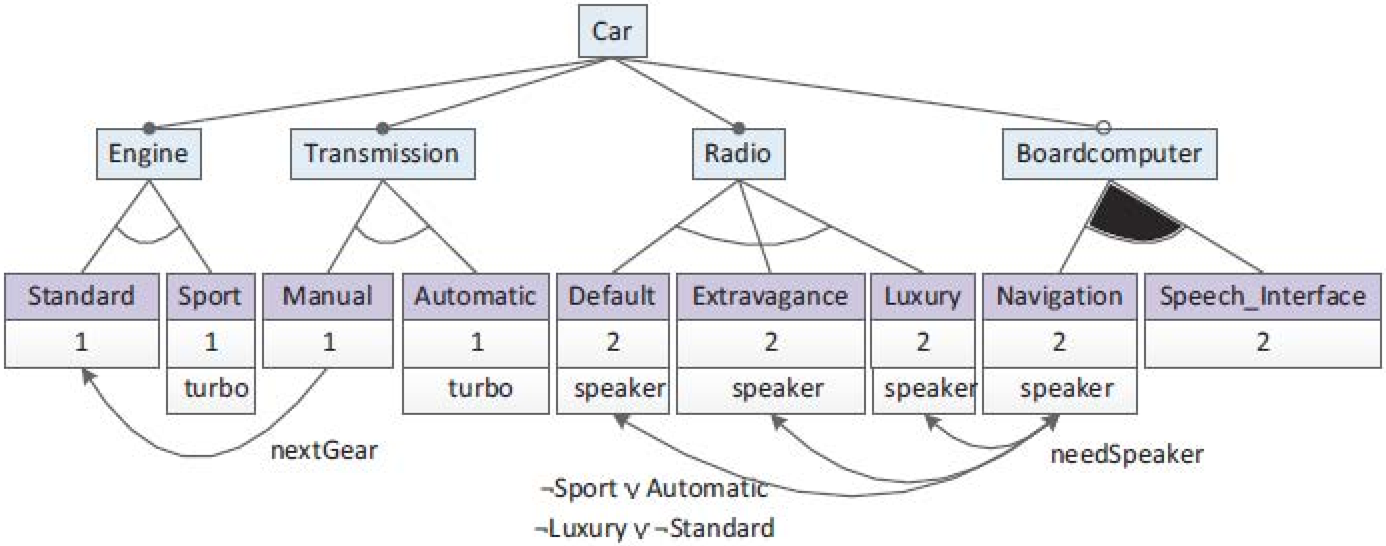
Feature\Product	0	1	2	3	4	5
Coffee	X	X	X	X	X	X
Beverage	X	X	X	X	X	X
2€		X	X	X	X	
Change		X			X	
Tea		X	X			X
Restock Cups		X		X		X
1€	X	X	X	X	X	X
Coins	X	X	X	X	X	X
Vending Machine	X	X	X	X	X	X

- ICPL can handle large-scale SPLs
- 2-wise with „normal“ hardware possible
- Easily over 90% variant reduction
- Even with ICPL: Calculation time can be several hours

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

Feature Annotations for More Efficient Combinatorics

- Annotate features with shared resources, communication links, testing priorities
- Use additional information for combinatorial testing
- **Consequence**: Even lesser variants to test and shorter computation time

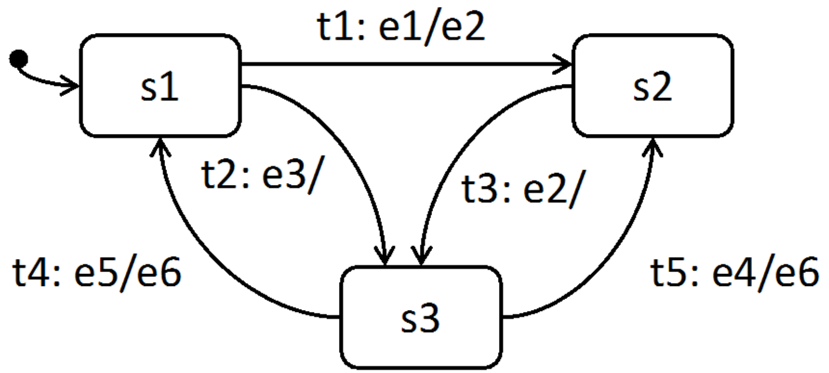


Kowal, M., Schulze, S., Schaefer, I.: Towards Efficient SPL Testing by Variant Reduction. In: VariComp. pp. 1–6. ACM (2013)

Regression-based SPL Testing

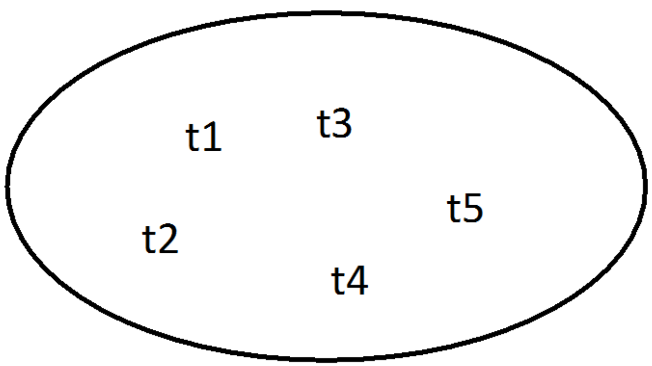
Model-based Testing - Procedure

Test Model **TM**



Coverage Criterion:
e.g., all transitions **C1**

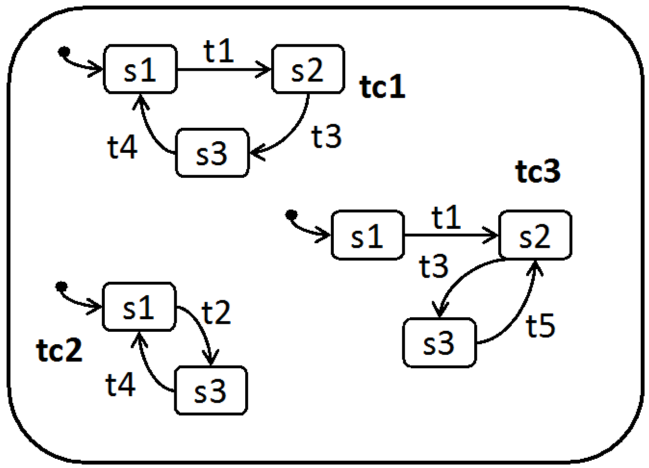
Test Goals **TG**



Model-based Testing – Procedure (2)

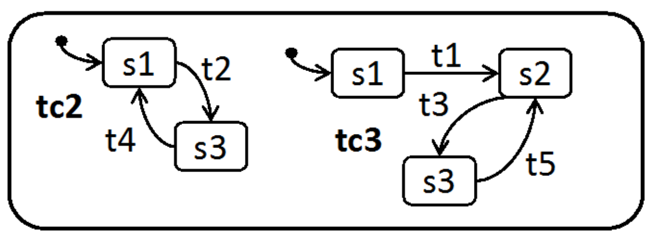
Test Case Generation

Test Suite **TS**

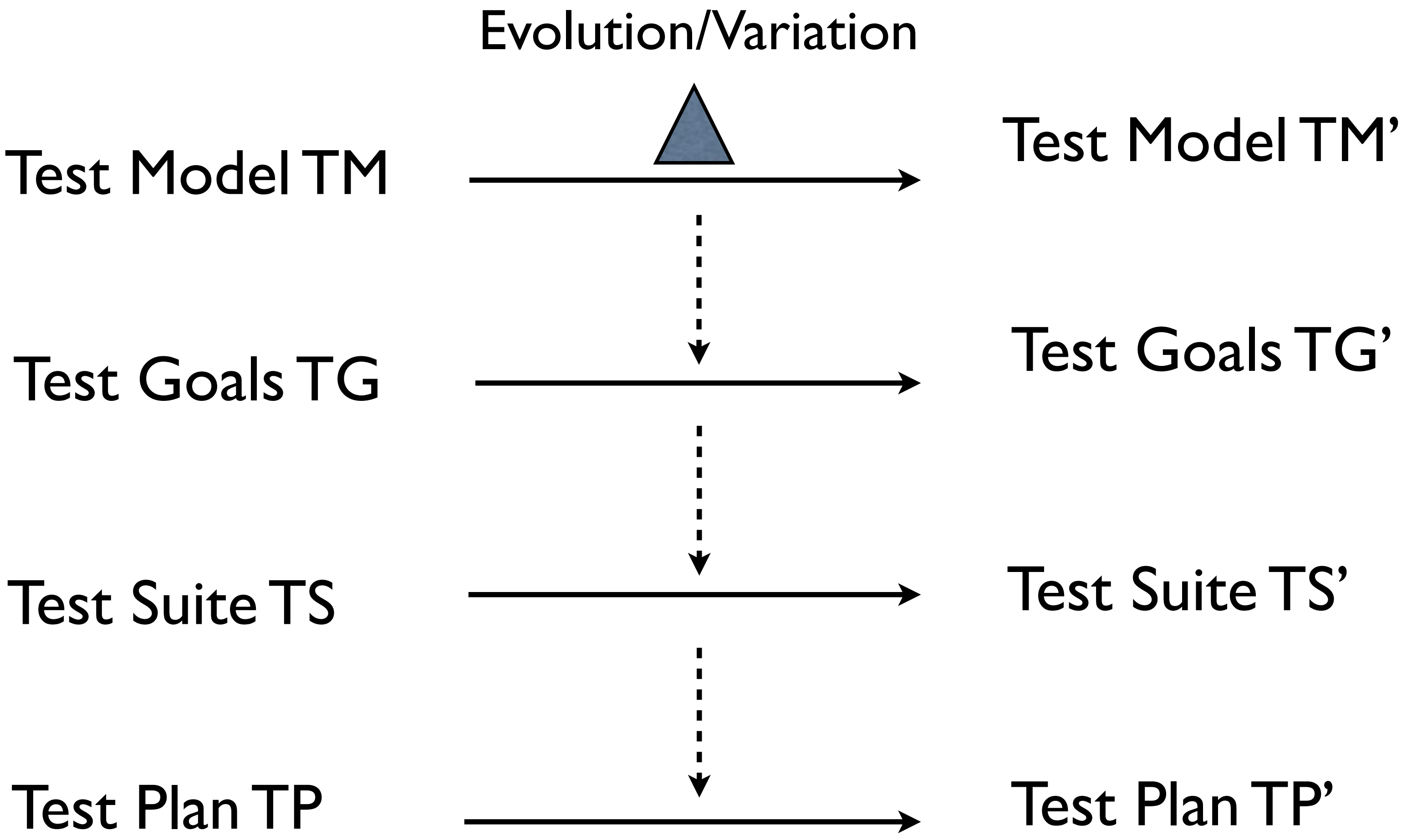


Test Selection

Test Plan **TP**



Incremental Model-based Testing



Delta-Modeling of Variant-Rich Systems

Core Product

Product Delta$_1$

[...]

Product Delta$_n$

- Product for valid feature configuration.
- Developed with Standard Techniques

- Modifications of Core Product.
- Application conditions over product features.
- Partial ordering for conflict resolution.

Delta-Modeling - Background

Instances of Delta-Languages:

- Software architectures (Delta-MontiArc)
- Programming languages (Delta-Java)
- Modeling languages (Delta-Simulink, Delta-State Machines, Deltarx)

Advantages of Delta-Modeling:

- Modular and flexible description of change
- Intuitively understandable and well-structured
- Traceability of changes and extensions
- Support for proactive, reactive and extractive SPLE

Delta-oriented Testing approaches

- Based on delta languages and modeling techniques, different testing approaches can be defined [Lity13]

- **Goal**: Reduce regression testing effort by only testing differences between products and not every product as a whole

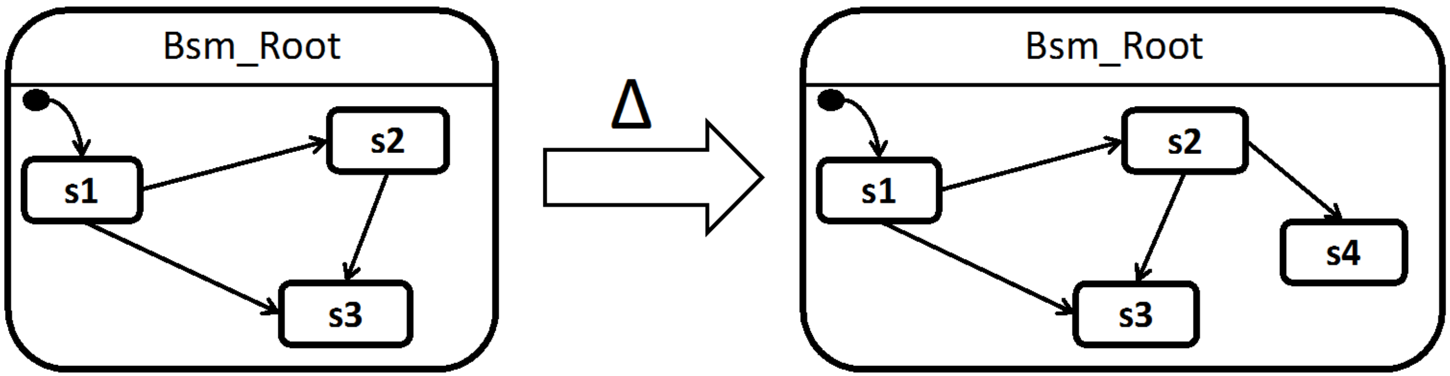
- **Deltas on variable test-models:**
  - Statemachines
  - Architectures
  - Activity Diagrams

- **Deltas on requirements** in natural language

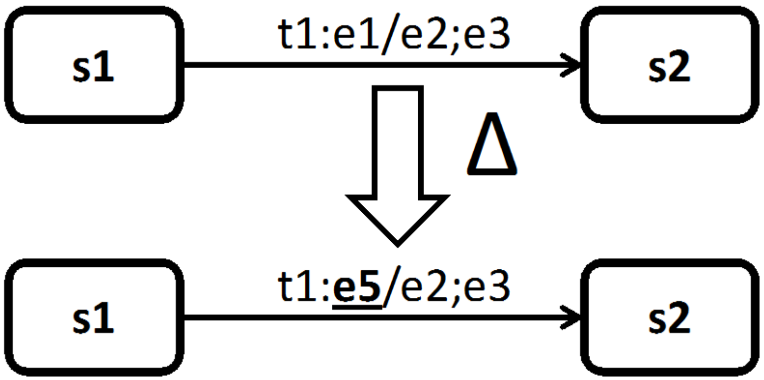
Technische Universität Braunschweig

Delta-oriented Test Models (Examples)

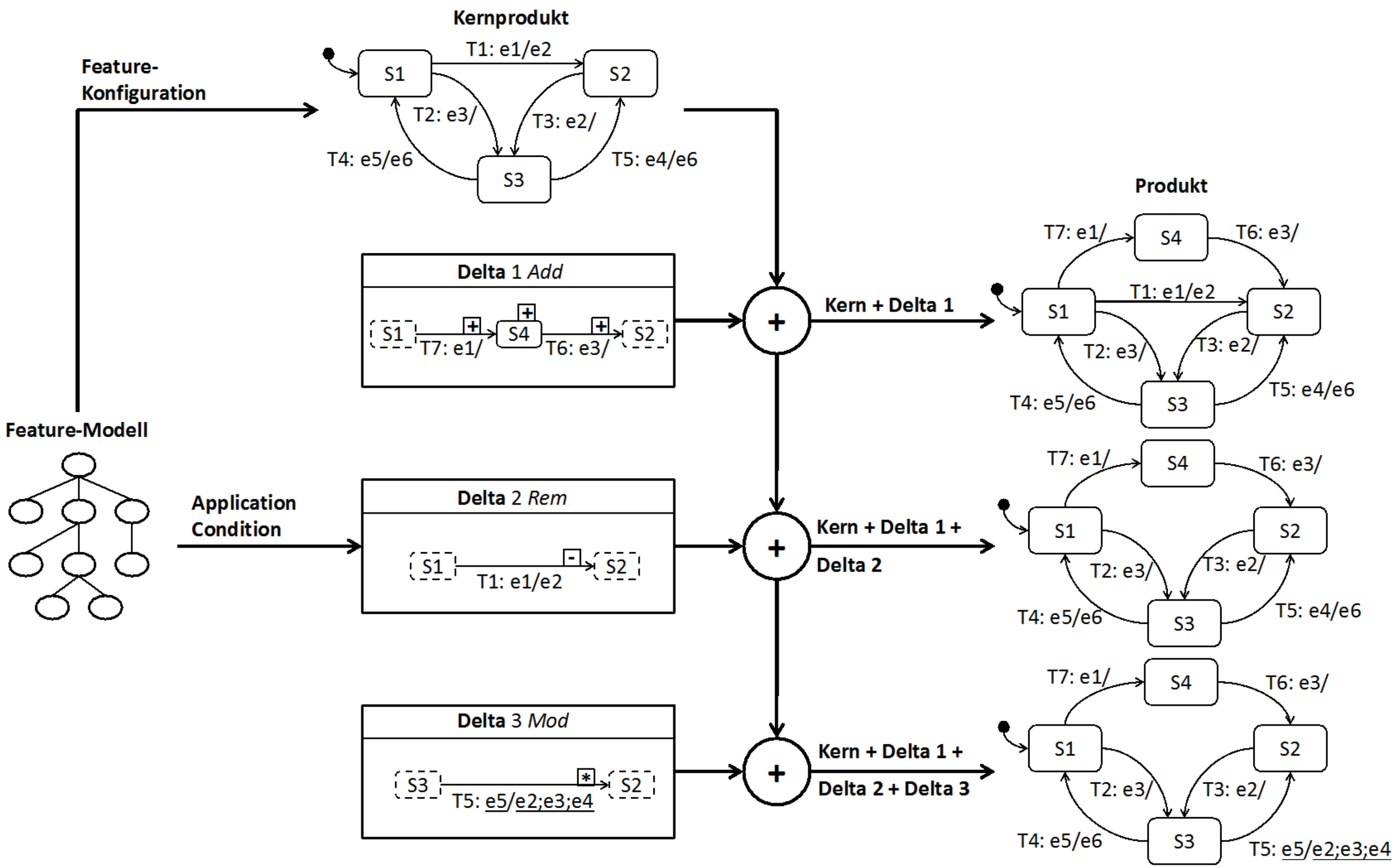
Adding a state to a State Machine:



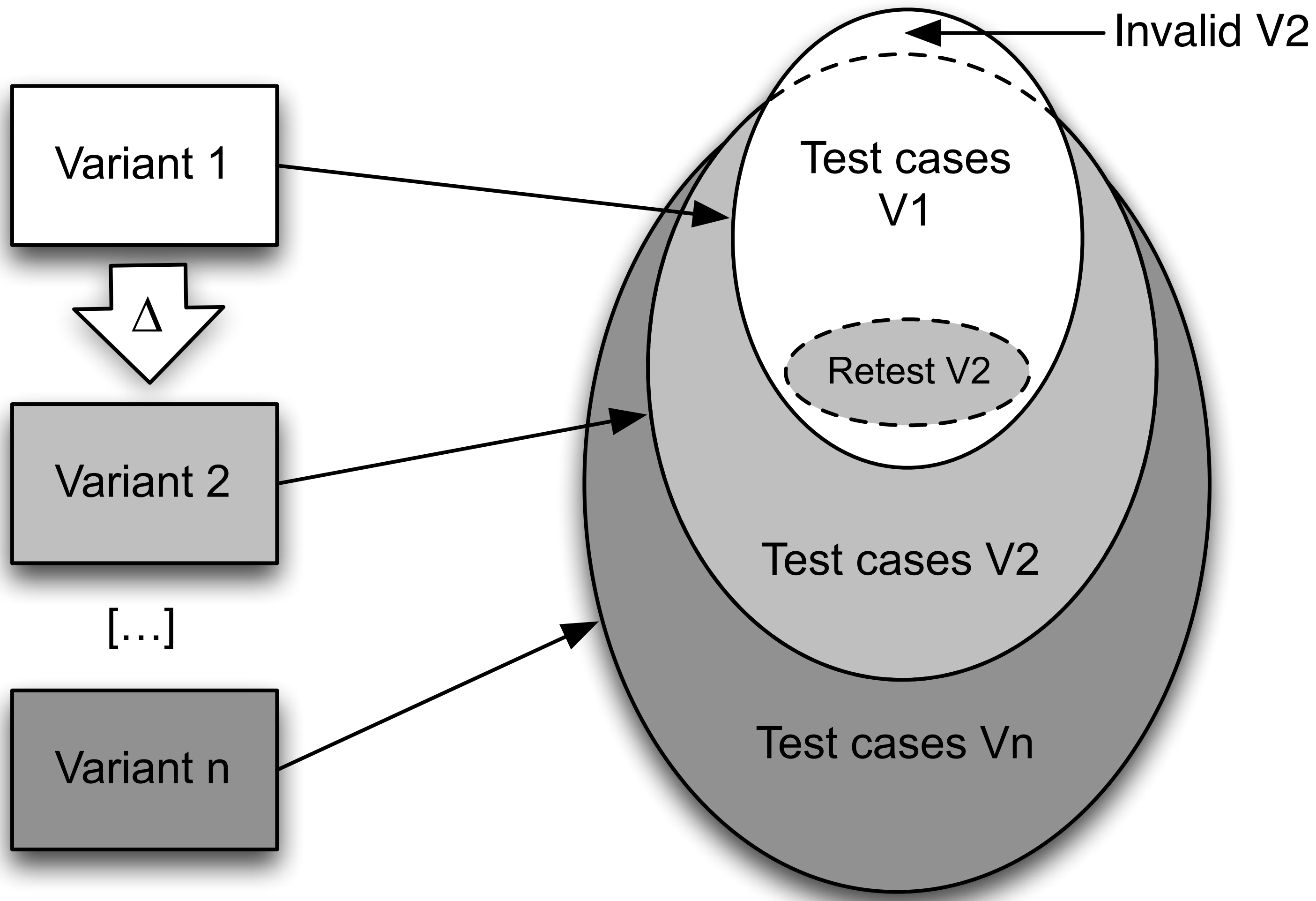
Changing the transition labels:



Delta-oriented Test Modeling



Classification of Test Cases by Delta-Analysis



Delta Testing - Procedure

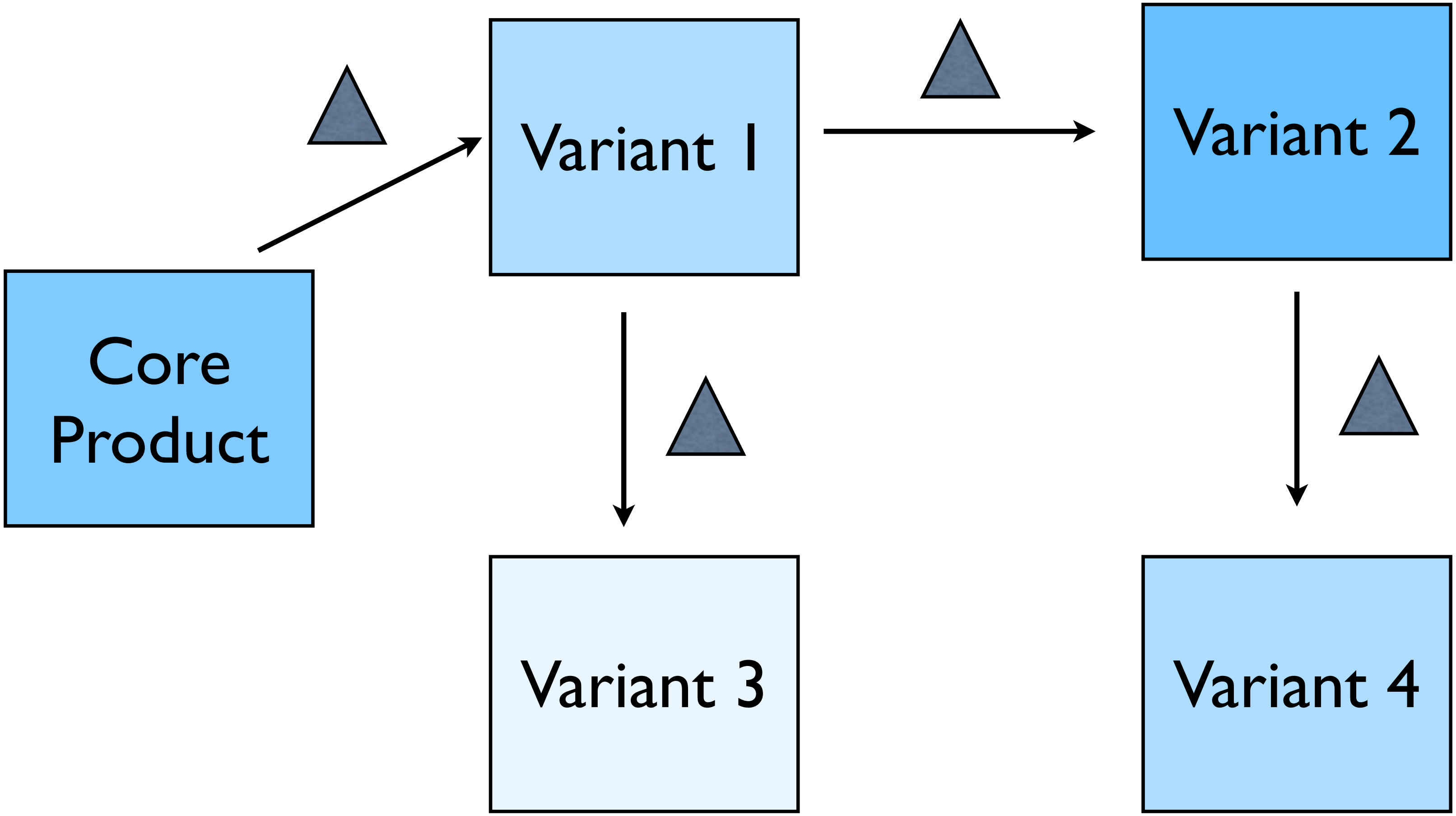
0. Fully test first product variant

1. Generate test cases for subsequent variants
   - Still valid and reuseable test cases?
   - Invalid test cases?
   - New test cases?

1. Selection of test cases by delta analysis:
   - Always test new test cases
   - Select subset of reuseable test cases for re-test

2. Optionally minimize resulting test suite by redundancy elimination

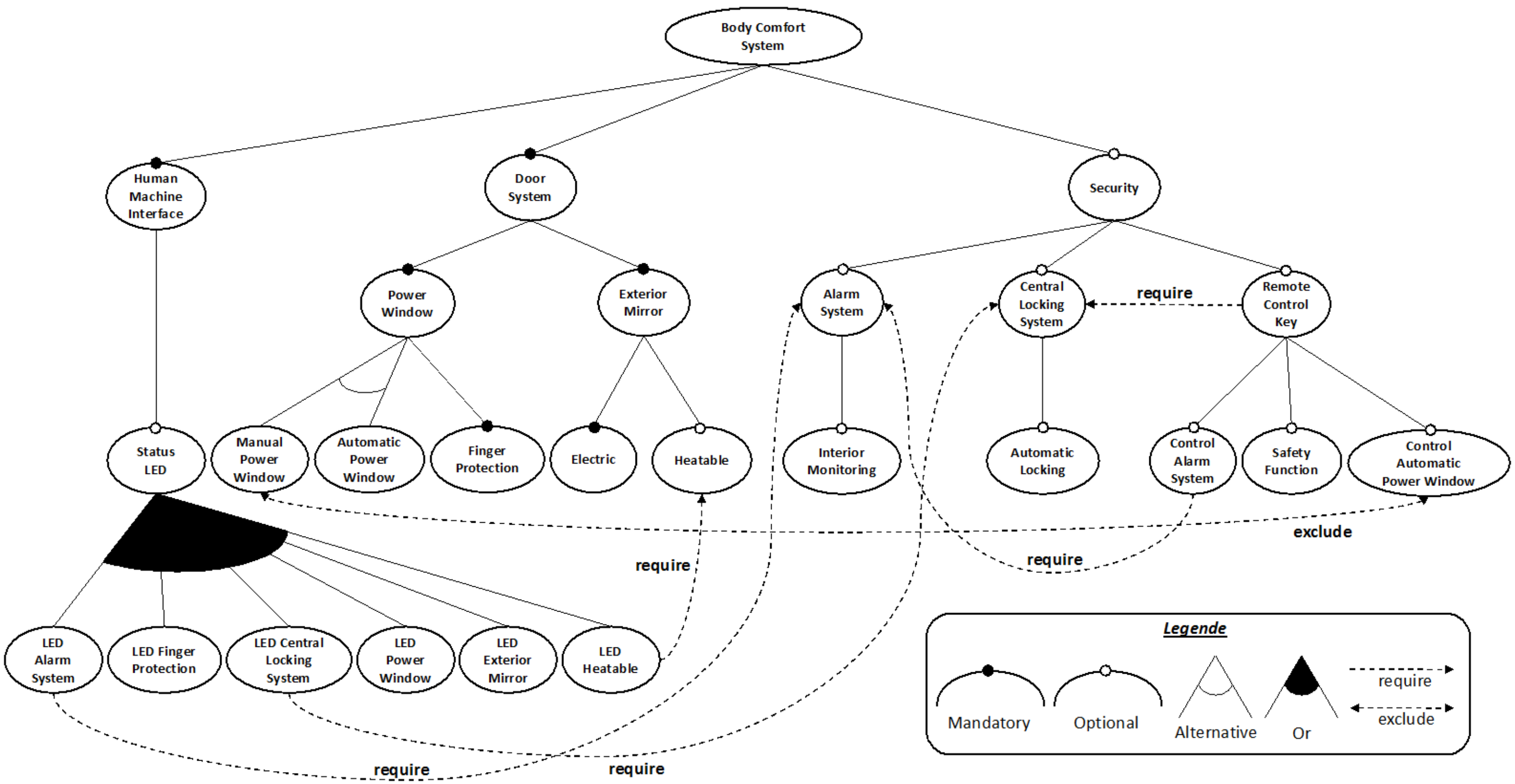
Delta-Testing Strategy



Case Study – Body Comfort System 2

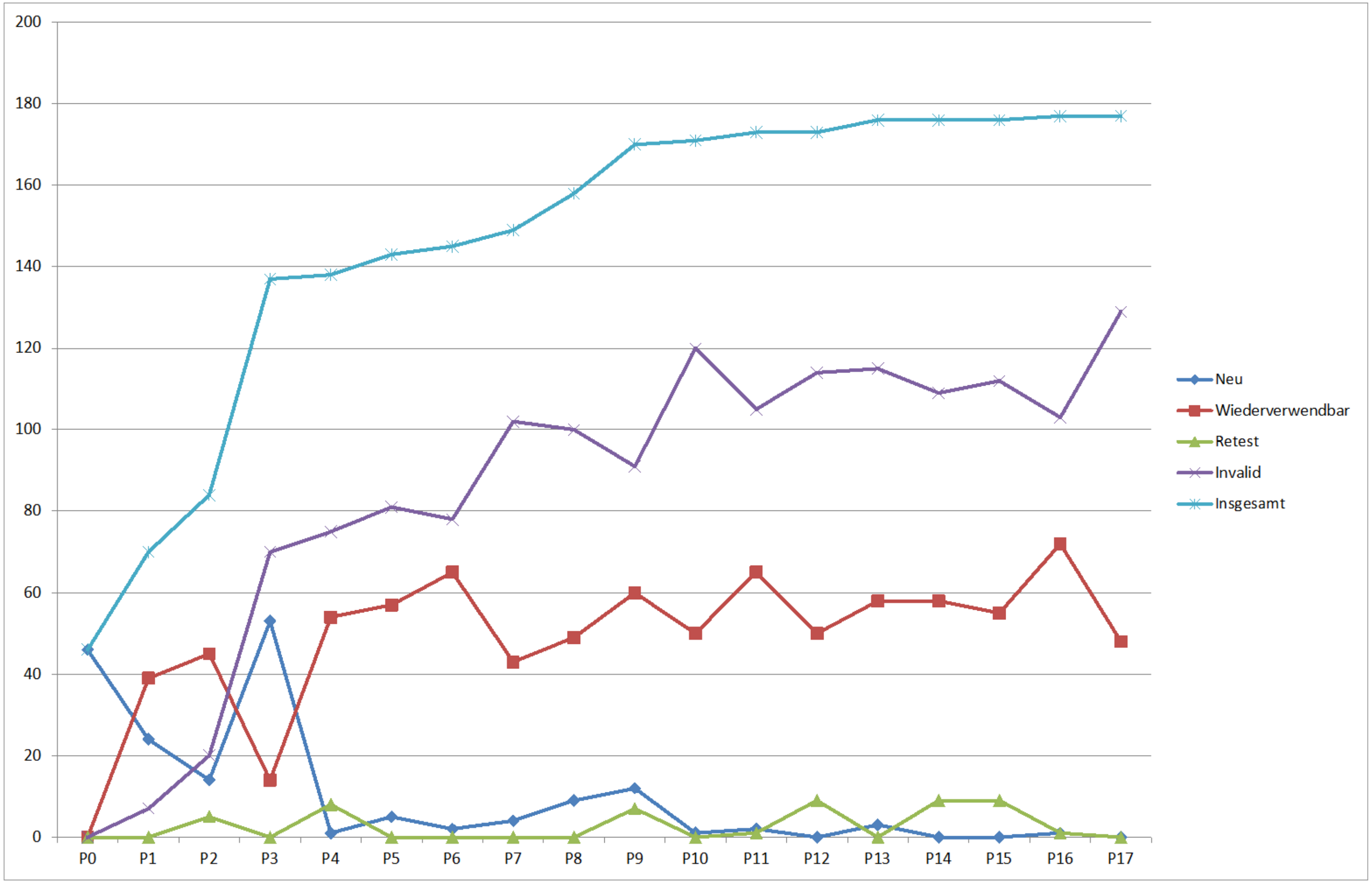
28 Features, 11616 Product Variants, 1 Core Product, 40 Deltas

16 Products for Pair-Wise Feature Coverage

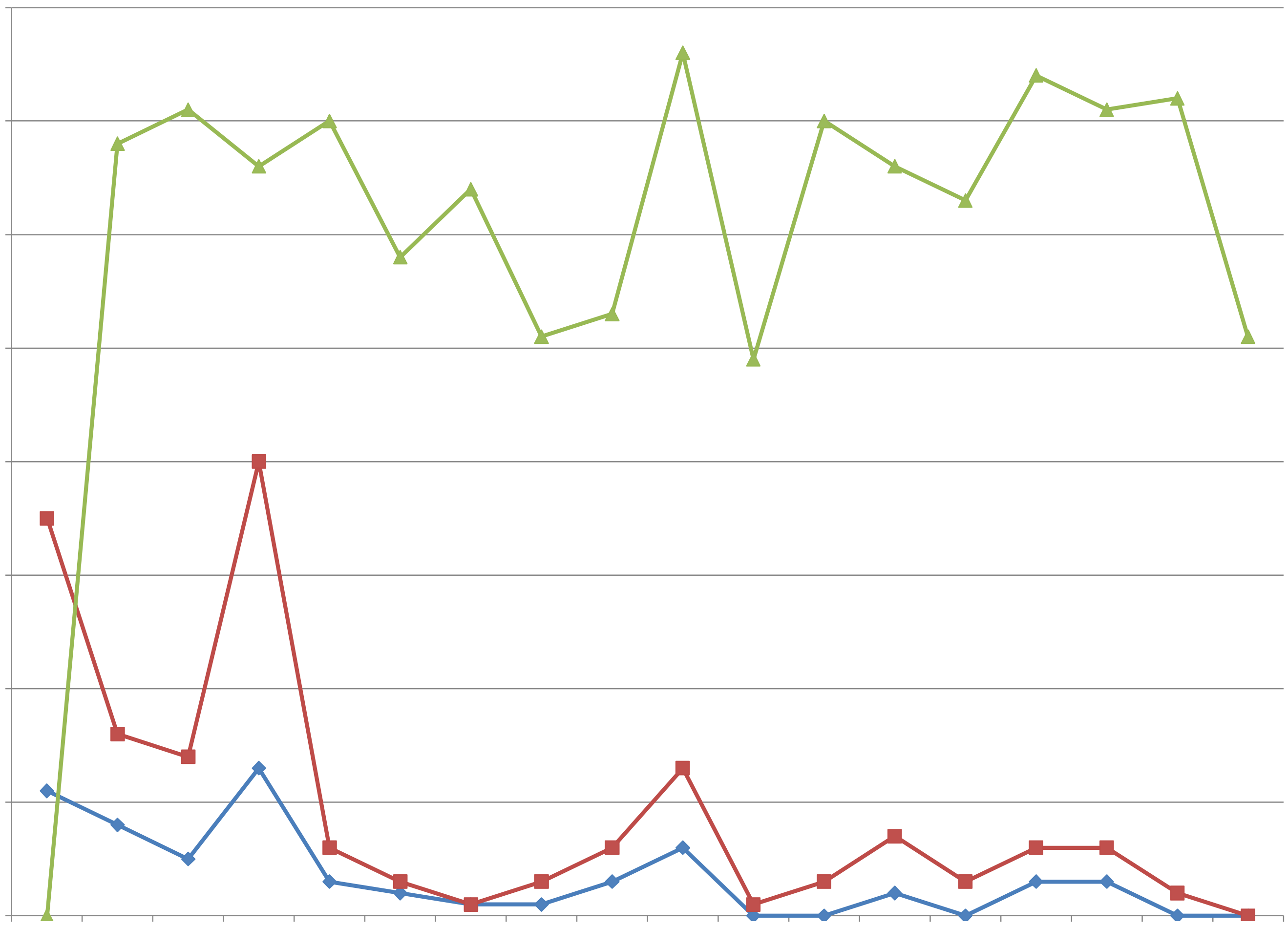


For more information see [BCS12]

Case Study BCM 2 – Delta-Testing Results



Case Study BCM 2 – Delta-Testing Results (2)



Requirements-Based Delta-oriented Testing

Requirements	*Test cases*
BCS_R1 If an object is detected in the window (window pressure P > threshold), activate the finger protection to prevent the power window from moving any further.	**BCS_TC1** Precondition: Window is open and an object is within the window Action: Press move up button Expected Result: Window moves up, until it reaches the objects and stops
BCS_R2 If the central locking system is activated and the power window is not in the top position, move the power window up, until it reaches the top position.	**BCS_TC2** Precondition: CLS is activated & power window is not in top position Action: Press move up button Expected Result: Power window moves to the top position and stops
BCS_R3 If the move down button for the power window is pressed and there is no Central locking system, move the power Window down. Otherwise, only move down if the central locking system is deactivated **BCS_R3V1** without CLS	**BCS_TC3** Precondition: No CLS installed Action: Press move down button Expected Result: Power window moves to the bottom position and stops
BCS_R3V2 with CLS	**BCS_TC4** Precondition: CLS installed and deactivated Action: Press move down button Expected Result: Power window moves to the bottom position and stops
BCS_R4 After the move up button has been tapped shortly (< 1 sec), the power window moves automatically up until it reaches the top position and then the movement stops.	**BCS_TC5** Precondition: Power window is at bottom position Action: Press move up button for less then 1 second Expected Result: Power window moves to the top position and stops
...	...
BCS_Rn	**BCS_TCn**

Technische Universität Braunschweig

ISF

Possible Strategies for Re-Test Selection

- Manually by test engineer
- (Semi-)Automatical classification of test cases into variants
- Formulation of requirements in delta-sets with linking of test cases to requirements
- Model-based impact analysis of changes by delta analysis

Contents

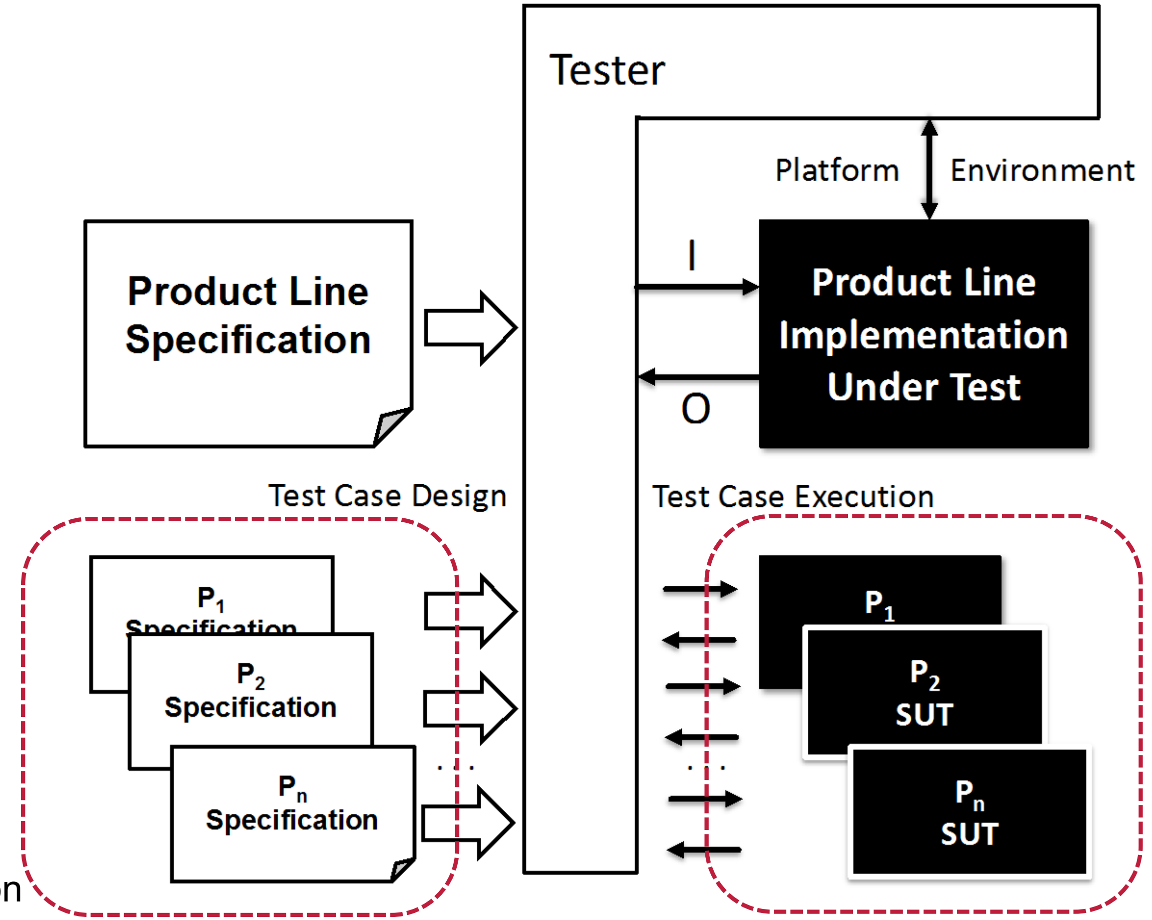
Part 1: Foundations of Testing and Model-based Testing

- Fundamental Notions and Concepts of Software Testing
- Model-based Testing
- A Theoretical Perspective on Model-based Testing

Part 2: Model-based Testing of Software Product Lines

- Sample-based Software Product Line Testing
- Regression-based Software Product Line Testing
- **Variability-Aware Software Product Line Testing**

Software Product Line Testing



150% SPL specification

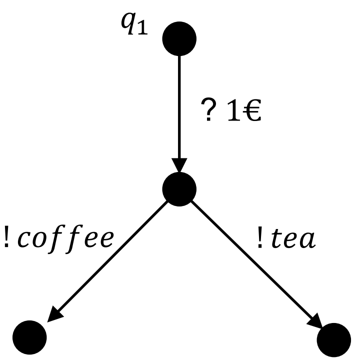
150% SPL implementation

Meaning of Specifications

Implementation freedom in single system IOCO testing

- The implementation must show **at least one** specified output behavior for specified input behaviors

- The implementation may show **arbitrary output behaviors** for unspecified input behaviors

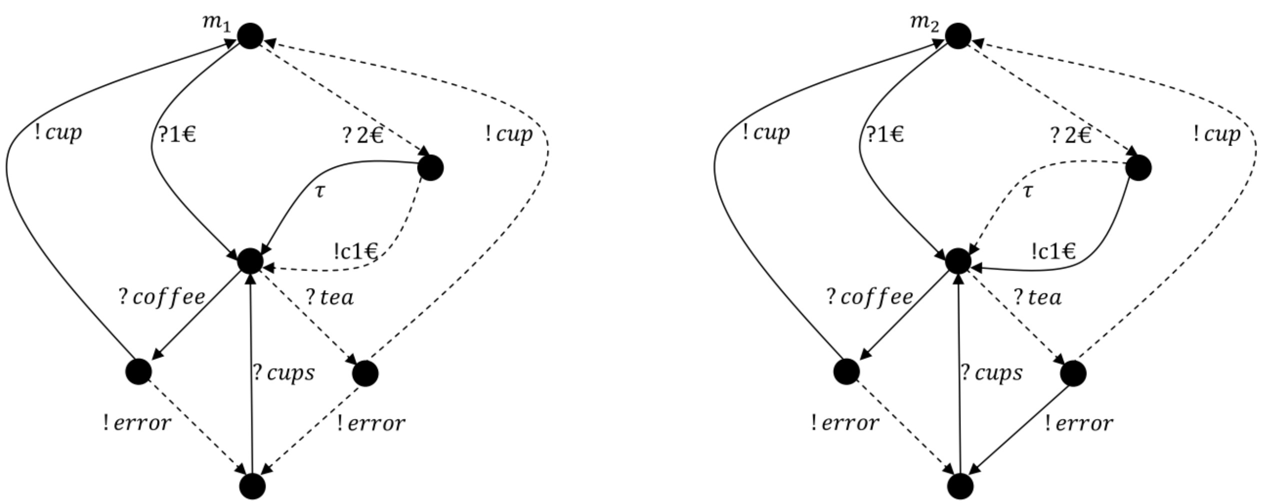


Implementation variability in SPL IOCO testing

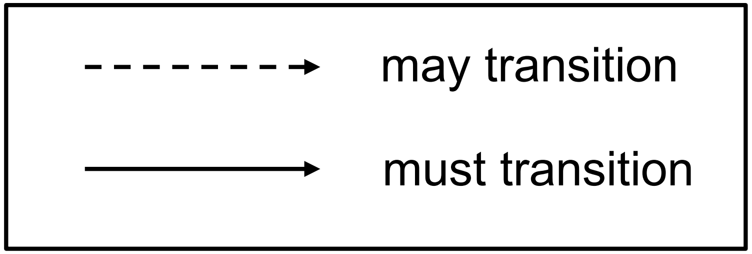
- Distinction between **mandatory** and **possible** input/ouput behaviors

- SPL specification with explicit transition **modality**

Modal I/O Transition Systems



(a) (b)



Contents

Literature

- [BCS12] - S. Lity, R. Lachmann, M. Lochau, I. Schaefer: *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*, Technische Universität Braunschweig, 2012

- [Kang90] - Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson - *Feature-Oriented Domain Analysis (FODA) Feasibility Study,* Technical Report*,* 1990

- [Lity13] - S. Lity, R. Lachmann, M. Lochau, M. Dukaczewski, I. Schaefer: *Delta-orientiertes Testen von variantenreichen Systemen*, ObjektSpektrum, 2013