Model-based Testing – Part 1

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


- 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

Specification → Test Case Design → System under Test (SUT) → Test Case Execution

Tester

Platform → Environment

I → O
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

- IEEE: Standard Glossary of Software Engineering Technology 610.121990
- IEEE: Standard for Software Test Documentation Std. 829-2008
Contents


- 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
Model-Based Testing

Specification

Test Case Design

Test Case Execution

Tester

Platform

Environment

SUT

I

O
Model-Based Testing

- **test model**
- **model-based test generation**
- **system model**

**SUT**

**Platform**

**Environment**

**test execution**

**pass/fail**
Model-based testing is the automation of black box tests.

- centered around test model
- abstract test case selection
- test case concretization
- basis for test case selection and coverage measure
- result verdict w.r.t. expected behavior
- dynamic observations

**SUT**

Platform

Environment

Blackbox

**Test execution**

Pass/fail

**Model-based test generation**

**System model**

June 18th, 2014 | Ina Schaefer | Model-based Testing – Part 1 | Seite 14
Contents


- 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
MBT from a Theoretical Point of View

implementation relation: \( i \simeq s \) with implementation \( i \) and formal behavioral specification \( s \)

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

intentional conformance: \( i \text{ conforms } s : \iff \llbracket i \rrbracket \subseteq \llbracket s \rrbracket \)

where \( \llbracket \cdot \rrbracket \) defines sets of all observable behaviors

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

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 Tretman
University of Twente *
Faculty of Computer Science, Formal Methods and Tools research group
P.O. Box 217, 7500 AE Enschede, The Netherlands
tretman@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: C\textsuperscript{3}OE de Reboot - Conformance Testing of REactive SYSTEMs; URL: http://tmt.cs.utwente.nl/CR3.

Joe C.M. Bostoen, Sybke Mooi (Eds.), CONCUR '90, LNCS 1664, pp. 401–413, 1999.
© Springer-Verlag Berlin Heidelberg 1999
Running Example

Beverage vending machine

- Input actions
  - \( I = \{1\text{€}, 2\text{€}\} \)
  - Transitions labels prefixed with “?”

- Output actions
  - \( U = \{\text{coffee}, \text{tea}\} \)
  - Transition labels prefixed with “!”
I/O-Labeled Transition Systems

I/O Labeled Transition System: \((Q, q_0, I, U, \rightarrow)\), 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, and
- \(\rightarrow \subseteq Q \times \text{act} \times Q\) is a labeled transition relation.
### LTS - Examples

\[ T_r(q_1) = \{ ?1\text{€}, ?1\text{€} \cdot !\text{coffee}, ?1\text{€} \cdot !\text{tea} \} \]

\[ T_r(q_5) = \{ ?1\text{€}, ?2\text{€} \} \]
Each computation 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

\[ \text{trace } \sigma = \mu_1 \mu_2 \cdots \mu_n \in \text{act}^* \]
\[ T_r(q_3) = \{ ?1\€, ?2\€, ?1\€ \cdot !\text{coffee}, ?2\€ \cdot !\text{coffee} \} \]

\[ T_r(q_2) = \{ ?1\€, ?2\€, ?1\€ \cdot !\text{coffee}, ?1\€ \cdot !\text{tea}, ?2\€ \cdot !\text{coffee}, ?2\€ \cdot !\text{tea} \} \]
LTS Trace Notations

Let \( s \) be an I/O LTS, \( \mu_i \in I \cup U \cup \{ \tau \} \) and \( a_i \in I \cup U \)

\[
\begin{align*}
S \mu_1 \ldots \mu_n \rightarrow S' & := \exists s_0, \ldots, s_n : s = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_n = s' \\
S \rightarrow & := \exists s' : s = S \rightarrow s' \\
\neg S \rightarrow & := \forall s' : s = \neg S \rightarrow s'
\end{align*}
\]
LTS Trace Notations

Let \( s \) be an I/O LTS, \( \mu_i \in I \cup U \cup \{ \tau \} \) and \( a_i \in I \cup U \)

\[
\begin{align*}
    \epsilon & \quad \Rightarrow \quad s' := s = s' \text{ or } s \xrightarrow{\tau} s' \\
    a & \quad \Rightarrow \quad s' := \exists s_1, s_2 : s \Rightarrow s_1 \rightarrow s_2 \Rightarrow s'
\end{align*}
\]

\[
\begin{align*}
    a_1 \cdots a_n & \quad \Rightarrow \quad s' := \exists s_0, \ldots, s_n : s = s_0 \Rightarrow s_1 \Rightarrow \ldots \Rightarrow s_n = s'
\end{align*}
\]
Let $s$ be an I/O LTS, $\sigma \in (I \cup U)^*$

\[
s \xrightarrow{\sigma} := \exists s' : = \exists s' : s \xrightarrow{\sigma} s'
\]

\[
\neg s \xrightarrow{\sigma} := \forall s' : s \xrightarrow{\sigma} s'
\]
The set of traces in an LTS is defined as

\[ Tr(s) := \{ \sigma \in (I \cup U)^* | \exists s' \in Q : q_0 \xrightarrow{\sigma} s' \}. \]
LTS - Examples

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

\[ Tr(q_6) = \{ ?1€, ?1€ \cdot !coffee \} \]
LTS - Examples

\[ \text{Tr}(q_7) = \{? 1\text{€}, ? 1\text{€} \cdot ! \text{coffee}\} \]

\[ \text{Tr}(q_8) = \{\} \]
An LTS is (weak) input-enabled iff for every state $s \in Q$ with $q_0 \Rightarrow^* s$ and for all $a \in I$ it holds that $s \xrightarrow{a}$. 
Input Completion - Example

Not input-enabled LTS

Input-enabled LTS
A First Attempt: Conformance as Trace Inclusion

\[ i \text{ conforms } s :\iff Tr(i) \subseteq Tr(s) \]

- Fails to refuse trivial implementations
- Fails to take the asymmetric nature of \( \mathcal{LT}S \) traces with I/O actions into account

Solution: explicit notion of quiescent behavior

Solution: distinguish input and output behaviors in traces
Some Auxiliary Definitions: Init Sets

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

\[
\text{init}(p) := \{ u \in (I \cup U) \mid p \xrightarrow{u} \}
\]

- $\text{init}(p_1) := \{ ? 1 \epsilon \}$
- $\text{init}(p_2) := \{ ! \text{coffee}, ! \text{tea} \}$
- $\text{init}(p_3) := \{ \}$
- $\text{init}(p_4) := \{ \}$
Some Auxiliary Definitions: Quiescent States

Let $s$ be an LTS, $p \in Q, P \subseteq Q$ and $\sigma \in (I \cup U)^*$. $p$ is **quiescent**, denoted $\delta(p)$, iff $\text{init}(p) \subseteq I$

\[ \begin{align*}
\text{init}(p_1) &:= \{ ? 1€ \} \\
\text{init}(p_2) &:= \{ !\text{coffee}, !\text{tea} \} \\
\text{init}(p_3) &:= \{ \} \\
\text{init}(p_4) &:= \{ \}
\end{align*} \]

\[ I = \{ ? 1€, ? 2€ \} \]
Let $s$ be an LTS, $p \in Q$, $P \subseteq Q$ and $\sigma \in (I \cup U)^*$. Then:

$$p \text{ after } \sigma := \{ q \in U \mid p \xrightarrow{\sigma} q \}$$

Diagram:
- $p_1$ with transitions $\delta$ and $?1\epsilon$
- $p_2$ with transitions $!\text{coffee}$ and $!\text{tea}$
- $p_3$ with transition $\delta$
- $p_4$ with transition $\delta$

- $U = \{!\text{coffee},!\text{tea}\}$
- $p_2 \text{ after } !\text{coffee} = \{p_3\}$
- $p_2 \text{ after } !\text{tea} = \{p_4\}$
- $p_1 \text{ after } ?2\epsilon = \{\}$
- $p_4 \text{ after } !\text{tea} = \{\}$
Some Auxiliary Definitions: Out Sets

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

\[
out(P) := \{ \mu \in U \mid \exists p \in P : p \xrightarrow{\mu} \} \cup \{ \delta \mid \exists p \in P : \delta(p) \}
\]

$P = p$ after $\sigma$

\[
P_1 = \{ p_1 \text{ after } \delta \} = \{ p_1 \}
\]
\[
P_2 = \{ p_2 \text{ after } !\text{tea}, p_2 \text{ after } !\text{coffee} \} = \{ p_3, p_4 \}
\]
\[
P_3 = \{ p_3 \text{ after } \delta \} = \{ p_3 \}
\]
\[
P_4 = \{ p_4 \text{ after } \delta \} = \{ p_4 \}
\]
Some Auxiliary Definitions: After-Out Sets

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

\[
\text{out}(P) := \{\mu \in U \mid \exists p \in P : p \xrightarrow{\mu}\} \cup \{\delta \mid \exists p \in P : \delta(p)\}
\]

\[
P_1 = \{p_1 \text{ after } \delta\} = \{p_1\}
P_2 = \{p_2 \text{ after } !\text{tea}, p_2 \text{ after } !\text{coffee}\} = \{p_3, p_4\}
P_3 = \{p_3 \text{ after } \delta\} = \{p_3\}
P_4 = \{p_4 \text{ after } \delta\} = \{p_4\}
\]

\[
\text{Out}(P_1) = \{\delta\}
\]

\[
\text{Out}(P_2) = \{!\text{tea}, !\text{coffee}\}
\]

\[
\text{Out}(P_3) = \{\delta\}
\]

\[
\text{Out}(P_4) = \{\delta\}
\]
Some Auxiliary Definitions: Suspension Traces

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

$\text{Straces}(p) := \{ \sigma' \in (I_S \cup U_S \cup \{\delta\})^* | p \xrightarrow{\sigma'} q \} \text{ where } q \xrightarrow{\delta} q \text{ iff } \delta(p)$

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

$\text{Straces}(p_2) = \{ !\text{coffee}, !\text{tea}, !\text{coffee} \cdot \delta, \text{tea} \cdot \delta \}$

$\text{Straces}(p_3) = \{ \delta \}$

$\text{Straces}(p_4) = \{ \delta \}$
Quiescent Behaviors

Trace \( Tr(q_1) \)

\[
\begin{align*}
    Straces(q_1) &= \{ \delta, ?1€, \delta \cdot ?1€, ?1€ \cdot !coffee, ?1€ \cdot !coffee \cdot \delta, \ldots \} 
\end{align*}
\]

Allows to discriminate (non-)behaviors

- \(?1€ \cdot \delta \not\in Straces(q_6), \text{ whereas } ?1€ \cdot \delta \in Straces(q_7)\)
- \(\ldots\)
Let $s \in \mathcal{LTS}(I \cup U)$ and $i \in \mathcal{IOTS}(I, U)$.

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

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

- Assume $\delta$-transitions in the $\mathcal{LTS}$ example

- Possible environmental stimulations are $\sigma = ? 1\€$ and $\sigma' = ? 2\€$

- Investigate the observable behavior
Example

\[ \text{out}(q_1 \text{ after } \sigma) = \{\text{coffee, tea}\}, \text{out}(q_1 \text{ after } \sigma') = \{\} \]

\[ \text{out}(q_2 \text{ after } \sigma) = \{\text{coffee, tea}\}, \text{out}(q_2 \text{ after } \sigma') = \{\text{coffee, tea}\} \]
Example

\[ \text{out}(q_3 \text{ after } \sigma) = \{\text{coffee}\}, \text{out}(q_3 \text{ after } \sigma') = \{\text{coffee}\} \]

\[ \text{out}(q_4 \text{ after } \sigma) = \{\text{coffee}\}, \text{out}(q_4 \text{ after } \sigma') = \{\text{tea}\} \]
Example

\[
\text{out}(q_5 \text{ after } \sigma) = \{\delta\}, \quad \text{out}(q_5 \text{ after } \sigma') = \{\delta\}
\]

\[
\text{out}(q_6 \text{ after } \sigma) = \{\text{coffee}\}, \quad \text{out}(q_6 \text{ after } \sigma') = \{\}
\]
Example

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

out(q_8 \text{ after } \sigma) = \{\}, \text{out}(q_8 \text{ after } \sigma') = \{\}
Second Attempt: I/O Conformance (IOR)

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

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

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

Problem: this is quite a lot!
Third Attempt: IOCO

Let \( s \in LTS(I \cup U) \) and \( i \in IOTS(I, U) \).

Focus on specified behaviors only

\[
i \text{ioco} s \iff \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)
\]

\( ior \subseteq \text{ioco} \)
Example [Tretmans, 1999]
Third Attempt: IOCO

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

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

$ior \subseteq ioco$

Still infinite in case of loops
The set of suspension traces under consideration is restricted to sub sets $\mathcal{F} \subseteq \text{act}^*$.

The restricted ioco relation is denoted as

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

where $\text{ior} = \text{ioco}_{\text{act}^*}$ and $\text{ior} = \text{ioco}_{\text{straces}(s)}$ holds.

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

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

holds.

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

denotes observation of quiescence
Example

Test case for specification $q_1$

Stimulated input
- $!1€$

Expected output
- *either coffee*
- *or tea*

Observable errors
- No output occurs: $\Theta$
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 \( s \in \mathcal{LTS}(I \cup U) \), \( i \in \mathcal{IOTS}(I \cup U) \) and \( F \subseteq \text{Straces}(s) \)

Then it holds that

1. the set \( \mathcal{TTEST} \) of all derivable test cases is **sound** and
2. the set \( \mathcal{TTEST} \) of all derivable test cases is **exhaustive**.
Contents


- 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
Some Further Readings


- ....
Model-based Testing – Part 2

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


- 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
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 their 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
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 grows 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
  - 3-wise CA: 95% of all errors

Trade-Off
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 $M_i = \emptyset, \forall i, i \in \{1, 2, ..., n\}$ 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

- Adaptation is still slow in computation!

- (Selected) Improvements
  - Finding core and dead features quickly
  - Early identification of invalid t-sets
  - Parallelization
  - and several more

```plaintext
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
```
Vending Machine and ICPL runtimes

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

<table>
<thead>
<tr>
<th>Feature \ Product</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffee</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Beverage</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2€</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tea</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Restock Cups</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1€</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Coins</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Vending Machine</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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

Regression-based SPL Testing
Model-based Testing - Procedure

Test Model $\text{TM}$

Coverage Criterion: e.g., all transitions $\text{C1}$

Test Goals $\text{TG}$
Model-based Testing – Procedure (2)

Test Case Generation

Test Suite TS

Test Selection

Test Plan TP
Incremental Model-based Testing

Evolution/Variation

Test Model TM  →  Test Model TM’

Test Goals TG  →  Test Goals TG’

Test Suite TS  →  Test Suite TS’

Test Plan TP  →  Test Plan TP’
Delta-Modeling of Variant-Rich Systems

- 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
Delta-oriented Test Models (Examples)

Adding a state to a State Machine:

Changing the transition labels:
Delta-oriented Test Modeling

Feature-Konfiguration

Feature-Modell

Kernprodukt

T1: e1/e2
T2: e3/
T3: e2/
T4: e5/e6
T5: e4/e6

S1

S2

S3

Delta 1 Add

<table>
<thead>
<tr>
<th>S1</th>
<th>S4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T7: e1/</td>
<td>T6: e3/</td>
</tr>
</tbody>
</table>

Delta 2 Rem

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: e1/e2</td>
<td></td>
</tr>
</tbody>
</table>

Delta 3 Mod

<table>
<thead>
<tr>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5: e5/e2; e3; e4</td>
</tr>
</tbody>
</table>

Product

T7: e1/
T6: e3/
T4: e5/e6
T5: e4/e6

S4

S1

S2

S3

S4

S1

S2

S3

Delta 1 + Delta 2

Kern + Delta 1 + Delta 2 + Delta 3

Kern + Delta 1
Classification of Test Cases by Delta-Analysis

- Variant 1
- Variant 2
- Variant n

- Test cases V1
- Test cases V2
- Test cases Vn

Invalid V2

Retest V2

[...]

\[ \Delta \]
Delta Testing - Procedure

0. Fully test first product variant

1. Generate test cases for subsequent variants
   • Still valid and reusable test cases?
   • Invalid test cases?
   • New test cases?

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

2. Optionally minimize resulting test suite by redundancy elimination
Delta-Testing Strategy

- Core Product
  - Variant 1
    - Variant 3
      - Variant 2
      - Variant 4
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

**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_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_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_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_R3V1**
without CLS

**BCS_R3V2**
with CLS

Test cases

**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_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_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_TC4**
Precondition: CLS installed and deactivated
Action: Press move down button
Expected Result: Power window moves to the bottom position and stops

**BCS_TC5**
Precondition: Power window is at bottom position
Action: Press move up button for less than 1 second
Expected Result: Power window moves to the top position and stops

...
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


- 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

Test Case Design

Product Line Specification

Test Case Execution

150% SPL implementation

Product Line Implementation Under Test

Platform → Environment

Tester

I

O

P_1

Specification

P_2

Specification

P_n

Specification

P_1

SUT

P_2

SUT

P_n

SUT
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/output behaviors
- SPL specification with explicit transition **modality**
Modal I/O Transition Systems

(a)
(b)

may transition
must transition
Contents


- 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
Literature

