Tutorial on testing techniques

A research point of view

Antoine Rollet

INP Bordeaux - Université de Bordeaux - LaBRI (UMR CNRS 5800), France
rollet@labri.fr
http://www.labri.fr/~rollet
Context of this presentation

This talk should:
- Provide some basics in the domain of testing
- Prepare the audience for following presentations
- Provide some historical research results on testing

This talk is not:
- My personal research results
- An exhaustive presentation
- Advanced research, more a general view on the topic

^implying that I am not a specialist of all of the presented topics!
Outline

1. Generalities on testing

2. Source Code Based Testing (SCBT)

3. Functional testing - Model Based Testing
   - "Historical" approaches of MBT: based on Mealy Machines
   - "Historical" approaches of MBT: based on Labelled Transition Systems

4. Conclusion
Outline

1. Generalities on testing

2. Source Code Based Testing (SCBT)

3. Functional testing - Model Based Testing
   - "Historical" approaches of MBT: based on Mealy Machines
   - "Historical" approaches of MBT: based on Labelled Transition Systems

4. Conclusion
Why testing?

Famous bugs (non exhaustive panorama)

- Ariane 5.01 (1996)
- Patriot missile (1991)
- First Pentium ® Chip (1994)
- Therac 25 (1985-1987)
- ... (long long list)
- Urban legends also (F16 fighter jet bug)

Wondering what the cost of software bugs? → $ 312 Billions per year according to Cambridge University (2013).

In fact it depends on how late you find it.
Why testing?

Famous bugs (non exhaustive panorama)

- Ariane 5.01 (1996)
- Patriot missile (1991)
- First Pentium® Chip (1994)
- Therac 25 (1985-1987)
- ... (long long list)
- Urban legends also (F16 fighter jet bug)

Wondering what the cost of software bugs? → $312 Billions per year according to Cambridge University (2013).

In fact it depends on how late you find it.
Why testing? (2)

Consequences:

- Product recall (Pentium® Chip, Toyota brake system bug (2009))
- Sometimes loss of human lifes (Therac 25, Patriot), loss of expensive system (Ariane 5.01)
- Space domain: send patches (NASA Curiosity Probe on Mars)
What is testing?

Dynamic testing

**Dynamic testing**: the software (IUT)\(^a\) is executed in order

- To ensure a “correct” behaviour
- To find bugs and defaults (Myers)

(\(\neq\) Static testing)

→ this presentation will focus mainly on dynamic testing techniques

\(^a\)Implementation Under Test

but not so simple ...
What is testing? (2)
Difficulty(ies) of testing

- Testing is a difficult/expensive task

B. Gates:

“50% of the people at Microsoft® are testers, and the programmers spend 50% of their time testing, thus Microsoft is more of a testing than a development organization”

http://www.informationweek.com/story/IWK20020517S0011

⇒ Important research domain

- Ideally a test should be exhaustive, but not possible in practice...

A simple function

```
int product(int i, int j);
```

$2^{64}$ possibilities. Considering one test per micro-second → 583000 years...
Difficulty(ies) of testing

- **Testing is a difficult/expensive task**

B. Gates:

“50% of the people at Microsoft® are testers, and the programmers spend 50% of their time testing, thus Microsoft is more of a testing than a development organization”\(^a\)

\(^a\)http://www.informationweek.com/story/IWK20020517S0011

⇒ **Important research domain**

- Ideally a test should be exhaustive, but not possible in practice...

A simple function

```c
int product(int i, int j);
```

\(2^{64}\) possibilities. Considering one test per micro-second \(\rightarrow\) 583000 years...
Difficulty(ies) of testing (2)

Dijkstra:

"Testing shows the presence, not the absence of bugs"

- The objective of testing is to increase confidence in the system (IUT)

Main problems:
- Find a “representative” sample of data (Test Data (TD)), providing “enough” confidence
- Automatically generate this sample of data
- Automatically provide a verdict (oracle problem)
Dijkstra:

“Testing shows the presence, not the absence of bugs “

- The objective of testing is to increase confidence in the system (IUT)

Main problems:
- Find a “representative” sample of data (Test Data (TD)), providing “enough” confidence
- Automatically generate this sample of data
- Automatically provide a verdict (oracle problem)
Testing point of view

Three dimensions of testing (Tretmans)

- **Level of detail**: Unit, Module, Integration ...
- **Accessibility**: White-box, black-box.
- **Characteristics**: Conformance, Robustness, Performance, ...

→ this presentation will focus mainly on (unit) conformance testing

Two (complementary) main approaches of conformance testing:

- **Functional Testing**: TD is generated using the specification of the System Under Test (SUT).
  If specification = Model → Model Based Testing
- **Structural Testing**: TD is generated using the “structure” of the SUT, generally the Source Code (Code Based Testing)
Testing point of view

Three dimensions of testing (Tretmans)

- **Level of detail**: Unit, Module, Integration ...
- **Accessibility**: White-box, black-box.
- **Characteristics**: Conformance, Robustness, Performance, ...

→ this presentation will focus mainly on (unit) conformance testing

Two (complementary) main approaches of conformance testing:

- **Functional Testing**: TD is generated using the specification of the System Under Test (SUT).
  If specification = **Model** → **Model Based Testing**
- **Structural Testing**: TD is generated using the “structure” of the SUT, generally the Source Code (**Code Based Testing**)

→ in many books, functional = black-box ; structural = white-box
Testing point of view

<table>
<thead>
<tr>
<th>Three dimensions of testing (Tretmans)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level of detail</strong> : Unit, Module, Integration ...</td>
</tr>
<tr>
<td><strong>Accessibility</strong> : White-box, black-box.</td>
</tr>
<tr>
<td><strong>Characteristics</strong> : Conformance, Robustness, Performance, ...</td>
</tr>
</tbody>
</table>

→ this presentation will focus mainly on (unit) conformance testing

<table>
<thead>
<tr>
<th>Two (complementary) main approaches of conformance testing :</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional Testing</strong> : TD is generated using the specification of the System Under Test (SUT). If specification = Model → Model Based Testing</td>
</tr>
<tr>
<td><strong>Structural Testing</strong> : TD is generated using the “structure” of the SUT, generally the Source Code (Code Based Testing)</td>
</tr>
</tbody>
</table>

→ in any case, TD are applied on the IUT and result is compared to the specification
Some words about integration testing

**Integration**: combining already tested components.

⚠️ Even if each component is working fine → integration may reveal new bugs

### Main (functional-decomposition based) strategies

- **Big-bang**
  → integrate all components together, then test the whole

- **Bottom-up**
  → from leaves to root of the functional decomposition tree

- **Top-down**
  → from root to leaves of the functional decomposition tree
  → need to use stubs

- **Sandwich**
  → combining Bottom-up and Top-Down

### Other approaches may be used (Call-graph based, Path based)
Another way to classify/point of view

**Functional Testing**

Checking that the IUT meets the *functional requirements*. Divided into four components:
Unit, Integration, System, Acceptance

**Non-Functional Testing**

Testing the application against *non-functional requirements*:
Performance, Load, Stress, Security, ...
Another way to classify/point of view

**Functional Testing**
Checking that the IUT meets the *functional requirements*. Divided into four components:
Unit, Integration, System, Acceptance

**Non-Functional Testing**
Testing the application against *non-functional requirements*:
Performance, Load, Stress, Security, ...

The previous classification will be used in this presentation
And beyond this classification, in a loose way

**Mutation Testing**

“Testing the Tester”

- Apply *tiny mutations* on the SUT (usually on the source code)
- Check that the test cases “*kill the mutants*” → *mutation score*
- **Difficulty**: apply significant mutations; equivalent mutants

**(non-)Regression Testing**

Verifying that an update of the SUT does not affect other parts

- Check that older test cases *still pass*
- Usually based on functional test cases
- **Difficulty**: costly, find a subset of test cases suited for regression testing
- **Remark**: sometimes, a distinction is made between regression and non-regression testing
And beyond this classification, in a loose way

**Mutation Testing**

“Testing the Tester”
- Apply tiny mutations on the SUT (usually on the source code)
- Check that the test cases “kill the mutants” → mutation score
- **Difficulty**: apply significant mutations; equivalent mutants

**Regression Testing**

Verifying that an update of the SUT does not affect other parts
- Check that older test cases still pass
- Usually based on functional test cases
- **Difficulty**: costly, find a subset of test cases suited for regression testing
- **Remark**: sometimes, a distinction is made between regression and non-regression testing
Outline

1 Generalities on testing

2 Source Code Based Testing (SCBT)

3 Functional testing - Model Based Testing
   - "Historical" approaches of MBT: based on Mealy Machines
   - "Historical" approaches of MBT: based on Labelled Transition Systems

4 Conclusion
References

Part essentially based on:

- [BH09] S. Bardin and P. Herrman, “Pruning the Search Space in Path-based Test Generation”, ICST 09
- [Rue08] M. Rueher, Software testing courses.
General Principle - Code coverage

Source Code Based Testing

TD is generated using the Source Code of the IUT

Ideally, the best TD would cover all possible executions. But usually not possible in practice.

- The more we cover code, the more confident we are, but
- The more we cover code, the more TD we need to generate and apply
- Notion of coverage criterion

There exists an ordering between coverage criteria:

\[ \text{all statements} < \ldots < \text{all executions} \]
Control Flow Graph

- Built from the source code
- A test may be seen as a path in the CFG
- Direct link between code coverage and CFG coverage
- Easy to obtain, e.g. with gcc:
  
  ```
  gcc -fdump-tree-cfg ...
  ```

- Not an equivalent representation:
  → risk of adding unfeasible paths
Control Flow Graph

- Built from the source code
- A test may be seen as a path in the CFG
- Direct link between code coverage and CFG coverage
- Easy to obtain, e.g. with gcc:
  
  ```
  gcc -fdump-tree-cfg ...
  ```

- **Not an equivalent representation**:  
  → risk of adding **unfeasible paths**
Control Flow Graph: example

```
// product of two int
int prod(int i, int j) {
    int k = 0;
    if (i == 2)
        k = i << 1;
    // error here; should be j << 1
    else {
        while (i > 0) {
            k = k + j;
            i--;
        }
    }
    return k;
}
```
Classical coverage criteria

Coverage criteria hierarchy (not exhaustive):

- **All statements (TER1)** = All nodes of the CFG
- **All decisions (TER2)** = All branches of the CFG
- ...  
- **All conditions (BCCC, Branch Condition Combination Coverage)**:
  each atomic predicate (i.e. condition) is tested with a true value and a false value
- **MCDC** (modified condition / decision coverage)
  Check “the role” of each condition in the decision
- ...  
- **All i-paths**
  (When feasible,) loop $j$ times in each loop ($0 \leq j \leq i$).
- **All executions** = All (feasible) paths $\rightarrow$ Usually infinite

Other possible approaches: e.g. Data Flow based.
Classical coverage criteria

Coverage criteria hierarchy (not exhaustive):

- **All statements (TER1)** = All nodes of the CFG
- **All decisions (TER2)** = All branches of the CFG
- **All conditions (BCCC**, Branch Condition Combination Coverage**): each atomic predicate (i.e. condition) is tested with a true value and a false value
- **MCDC** (modified condition / decision coverage)
  Check “the role” of each condition in the decision
- **All i-paths**
  (When feasible,) loop $j$ times in each loop ($0 \leq j \leq i$).
- **All executions** = All (feasible) paths $\rightarrow$ Usually infinite

Other possible approaches: e.g. **Data Flow** based
Constraint Based Testing

Constraint-Based Testing (CBT)

CBT is the process of generating test cases against a testing objective by using constraint solving techniques (Gotlieb)

Principle of Test Generation

- Given a location in the program under test, automatically generate a TD that reaches this location
- Transform (part of) the program into a logical formula \( \varphi \), such that solving \( \varphi \) provides a TD

\[ \rightarrow \] Does not solve the oracle problem.

\[ \rightarrow \] Pointers may lead to difficult problems
Constraint Based Testing

**Constraint-Based Testing (CBT)**

CBT is the process of generating test cases against a testing objective by using *constraint solving techniques* (Gotlieb)

**Principle of Test Generation**

- Given a location in the program under test, *automatically generate a TD that reaches this location*
- *Transform (part of) the program into a logical formula* $\varphi$, such that *solving* $\varphi$ *provides a TD*

→ Does not solve the *oracle* problem.

→ Pointers may lead to difficult problems
Path Predicate

Path predicate

Given a path $\Pi$ of a program, a formula $\varphi_\Pi$ is a path predicate of $\Pi$ if for a given set of values $V$, $V \models \varphi_\Pi \implies$ the execution of the program on $V$ follows $\Pi$

→ Find a solution (if any) to $\varphi_\Pi$ in order to activate $\Pi$

- Need to remember the values of variables along the path
  → need to rename each variable in case of assignment
  → SSA\textsuperscript{1} form

Remark : Using gcc, SSA form can be easily obtained :

```
gcc -S -fdump-tree-ssa ...
```

\textsuperscript{1}(Single Static Assignment)
Path predicate example

1. \textit{read} \ (y, z); \quad \rightarrow \quad y_0, z_0 \text{ as inputs}
2. \quad y = y + 2; \quad \rightarrow \quad y_1 = y_0 + 2
3. \quad x = y + 4; \quad \rightarrow \quad x_1 = y_1 + 4
4. \quad \textit{if} \ (x > 2 \times z) \quad \rightarrow \quad x_1 > 2 \times z_0 \quad \text{or} \quad x_1 \leq 2 \times z_0 \text{ depending on path}
5. \quad x = y + 2; \quad \rightarrow \quad x_2 = y_1 + 2

For the \textbf{path} : 1 \rightarrow 2 \rightarrow 3 \rightarrow (4, \text{true}) \rightarrow 5, \\
the corresponding \textbf{predicate} is : \\
y_1 = y_0 + 2 \land x_1 = y_1 + 4 \land x_1 > 2 \times z_0 \land x_2 = y_1 + 2.

Considering the inputs, we have \( y_0 + 6 > 2 \times z_0 \)

Example of TD : \( y = 0, z = 0 \)
Path predicate example

1. \( \text{read} \ (y,z); \) \quad \rightarrow \quad y_0, z_0 \text{ as inputs} \\
2. \( y = y + 2; \) \quad \rightarrow \quad y_1 = y_0 + 2 \\
3. \( x = y + 4; \) \quad \rightarrow \quad x_1 = y_1 + 4 \\
4. \( \text{if} \ (x > 2 \times z) \) \quad \rightarrow \quad x_1 > 2 \times z_0 \\
    \quad \text{or} \quad x_1 \leq 2 \times z_0 \text{ depending on path} \\
5. \( x = y + 2; \) \quad \rightarrow \quad x_2 = y_1 + 2 \\

For the path : \( 1 \rightarrow 2 \rightarrow 3 \rightarrow (4, \text{true}) \rightarrow 5, \)
the corresponding predicate is :
\( y_1 = y_0 + 2 \land x_1 = y_1 + 4 \land x_1 > 2 \times z_0 \land x_2 = y_1 + 2. \)
Considering the inputs, we have \( y_0 + 6 > 2 \times z_0 \)
Example of TD : \( y = 0, z = 0 \)
if/then; Φ-expressions, SSA form

\[
\begin{align*}
\text{if } (a > 0) & \quad \Rightarrow \quad \text{if } (a_0 > 0) \\
i = 4; & \quad \Rightarrow \quad i_1 = 4; \\
\text{else} & \quad \Rightarrow \quad \text{else} \\
i = 5; & \quad \Rightarrow \quad i_2 = 5; \\
& \quad \Rightarrow \quad i_3 = \phi(i_1, i_2);
\end{align*}
\]

\(\Phi\)-expr \(\rightarrow\) decide the value of \(i_3\) according to the path used to reach it

- \(a_0 > 0\) \(\Rightarrow\) \(i_1 = 4 \land i_3 = i_1\)
- \(\neg(a_0 > 0)\) \(\Rightarrow\) \(i_2 = 5 \land i_3 = i_2\)

Choice of path (join operator)

\[
\text{join}(a_0 > 0 \land i_1 = 4 \land i_3 = i_1, \neg(a_0 > 0) \land i_2 = 5 \land i_3 = i_2)
\]
While; $\Phi$-expressions, SSA form

While : $\Phi$-expr added just before the decision

<table>
<thead>
<tr>
<th>Initial code</th>
<th>SSA code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1;$</td>
<td>$x_1 = 1;$</td>
</tr>
<tr>
<td>while ( x != 10 ) {</td>
<td>$x_2 = \phi(x_1, x_3);$</td>
</tr>
<tr>
<td>$c = x;$</td>
<td>$c = x_2;$</td>
</tr>
<tr>
<td>$x = x + 1;$</td>
<td>$x_3 = x_2 + 1;$</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>print(x);</td>
<td>print($x_2$);</td>
</tr>
</tbody>
</table>

$\rightarrow$ but need to solve a constraint according to the number of loops desired

$\rightarrow$ each loop turn $\Rightarrow$ new recursive constraints
Symbolic Depth First Search (DFS)

**Path Based TD generation**

1. Select (another) path $\Pi$ of the CFG
2. Build the corresponding predicate $\varphi_{\Pi}$
3. Solve $\varphi_{\Pi}$ (if possible); keep an input solution as a TD (if any)
4. Back to (1)

A strategy for the coverage criterion **All paths**:
- The CFG is unwound providing an execution tree
- The execution tree is explored using a **DFS** approach

😊 Constraint solving, even on a single path, may be costly (unwinding, unfeasible paths, ...).
Symbolic Depth First Search (DFS)

Path Based TD generation

1. Select (another) path $\Pi$ of the CFG
2. Build the corresponding predicate $\varphi_\Pi$
3. Solve $\varphi_\Pi$ (if possible); keep an input solution as a TD (if any)
4. Back to (1)

A strategy for the coverage criterion All paths :

- The CFG is unwound providing an execution tree
- The execution tree is explored using a DFS approach

Constraint solving, even on a single path, may be costly (unwinding, unfeasible paths, ...).
Symbolic Depth First Search (DFS)

### Path Based TD generation

1. Select (another) path $\Pi$ of the CFG
2. Build the corresponding predicate $\varphi_\Pi$
3. Solve $\varphi_\Pi$ (if possible); keep an input solution as a TD (if any)
4. Back to (1)

A strategy for the coverage criterion **All paths**:  
- The CFG is unwound providing an execution tree  
- The execution tree is explored using a DFS approach

.Constraint solving, even on a single path, may be costly (unwinding, unfeasible paths, ...).
Concolic approach

Idea: accelerate symbolic execution by using concrete execution at the same time. → Permits to select feasible paths.
Concolic approach

Idea: accelerate *symbolic* execution by using *concrete* execution at the same time. → Permits to select *feasible paths*.

Random choice: e.g. $x=3$ (concrete)
Concolic approach

Idea: accelerate symbolic execution by using concrete execution at the same time. → Permits to select feasible paths.

Backtrack + resolution: $x \geq 2 \land x \geq 5$; possible solution: $x = 8$
**Concolic approach**

**Idea**: accelerate *symbolic* execution by using *concrete* execution at the same time. → Permits to select *feasible paths*.

![Concolic approach diagram]

\[x=8\text{ (concrete)}\]
Concolic approach

Idea: accelerate **symbolic** execution by using **concrete** execution at the same time. → Permits to select **feasible paths**.

Backtrack + resolution: $x \geq 2 \land x \geq 5 \land x < 1$; **unfeasible**
Concolic approach

Idea: accelerate symbolic execution by using concrete execution at the same time. → Permits to select feasible paths.

Backtrack + resolution: $x < 2$; etc ...
Some known tools of CBT (non exhaustive ...)

- **C - C++**:
  - Cute (University of Illinois, Berkeley)
  - Crest (Berkeley)
  - Dart (Bell Labs)
  - EXE (University of Stanford)
  - Inka (INRIA, France)
  - PathCrawler (CEA)

- **Java, C#**:
  - CATG (NTT Labs, Berkeley)
  - CPBPV (I3S, Sophia Antipolis, France)
  - JCute (University of Illinois, Berkeley)
  - Java Path Finder (NASA)
  - Pex (Microsoft)
  - Pet (University of Madrid)

- **Binaries**:
  - Osmose (CEA)
  - Sage (Microsoft)
  - Triton (Bordeaux University, Quarkslab)
Outline

1. Generalities on testing
2. Source Code Based Testing (SCBT)
3. Functional testing - Model Based Testing
   - "Historical" approaches of MBT: based on Mealy Machines
   - "Historical" approaches of MBT: based on Labelled Transition Systems
4. Conclusion
Generalities

**Functional testing**

TD is generated using the *specification* of the SUT

Example of known methods:

- **Equivalent classes analysis** - **Boundary values analysis**
  - Divide the global (multi-dimensional) set of inputs into equivalent classes
  - **One value** of the class tested $\Rightarrow$ **all values** of the class tested
  - Sometimes add tests for the **boundaries**, often source of bugs
  - Decreases the number of TD in theory, sometimes not easy in practice
Generalities on testing

Source Code Based Testing (SCBT)

Functional testing - Model Based Testing

Conclusion

Functional testing

TD is generated using the *specification* of the SUT

Example of known methods:

- **Combinatory testing - Pairwise testing**
  
  When more than 2 params, TD checks *only pairs* of values, not all possible combinations\(^a\)

  Example: 3 boolean variables:

  \[
  \begin{array}{ccc}
  V1 & V2 & V3 \\
  0  & 0  & 0  \\
  0  & 0  & 1  \leftarrow \text{redundant, remove test case} \\
  0  & 1  & 0  \leftarrow \text{redundant, remove test case} \\
  \ldots \\
  \end{array}
  \]

  \(^a\)www.pairwise.org
Generalities on testing Source Code Based Testing (SCBT)  

Functional testing - Model Based Testing

**Generalities**

**Functional testing**

TD is generated using the *specification* of the SUT

Example of known methods:

- **Random testing**
  - Quick feedback for coarse testing
  - In case of bug on few values, low probability to find it
  - May provide an important number of test cases $\rightarrow$ oracle?
Generalities on testing

Source Code Based Testing (SCBT)

Functional testing - Model Based Testing

Conclusion

Functional testing

TD is generated using the specification of the SUT

Example of known methods:

- **Model Based Testing**
  - Powerful technique
  - Particularly adapted for testing reactive systems, communication protocols
  - Not easy to have a (formal) model in practice
    😞 ← requires an important modelling effort (costly, but generally profitable)
  - Requires sometimes a *mapping* between abstract test cases and concrete test cases
What about Model Based Testing?

Model Based Testing (MBT) → testing with the ability to detect faults which do not conform to a model called specification.

→ possible automation for test generation, test execution, test evaluation (verdict)
→ Formal Methods
What about Model Based Testing? (2)

- Test cases are generated from the **Model**
- As usual, TD are applied on the **Implementation**, and results are compared with the **specification**
- Problems:
  - Need to find a “good” model of the specification
  - What does **specify** mean?
  - What does **conform** mean?
- **Implementation** is supposed to be equivalent to a formal model (but Implementation is unknown)
- Need a **conformance relation** between the **Specification** and the **Implementation**
What about Model Based Testing? (2)

- Test cases are generated from the Model
- As usual, TD are applied on the Implementation, and results are compared with the specification
- Problems:
  - Need to find a “good” model of the specification
  - What does specify mean?
  - What does conform mean?
- Implementation is supposed to be equivalent to a formal model (but Implementation is unknown)
- Need a conformance relation between the Specification and the Implementation

Two historical approaches of MBT of reactive systems:
- Finite State Machines
- Labeled Transition Systems
General schema

Property P

\[ S \models P \]

Specification S

I conf S ?

Implementation I
General schema

Property P

\( S \models \varphi \)?

Specification S

I conf S ?

Implementation I

VERIFICATION
Generalities on testing
Source Code Based Testing (SCBT)

Functional testing - Model Based Testing

Conclusion

General schema

Property P

S \models P ?

Specification S

I \text{conf } S ?

TEST

Implementation I
General schema

Property P

Specification S

Implementation I

Test cases

Verdict

S \models P ?

I conf S ?

? observation

! control
General schema

Property $P$

Specification $S$

Implementation $I$

Test Generation

Test cases

Verdict

$S \models P$?

$I \text{conf} S$?

? observation

! control
General schema

Property $P$

Specification $S$

Implementation $I$

Test Generation

Test Purpose

Verdict

Test cases

control

observation
Modelling the system (1)

Find a “good” model

- Necessity to find a formal model adapted to the description of the specification
- Too abstract $\implies$ not realistic, no interest
- Too detailed $\implies$ difficult to model, too many cases to check
- Usually, the choice of the model has an impact on choice of the testing method (and conversely)?

Many possibilities of models, more or less formal
Example of Model: Mealy machine (FSM with outputs)

Simple coffee machine controller giving change, managing 2 coins

Inputs: \{ coffeeReq?, 25?, 50? \} (labelled with “?”)

Outputs: \{ nop!, 25!, 50!, giveCoffee! \} (labelled with “!”)

Possibility to add time (TFSM), data variables (EFSM), or both (TEFSM)
Modelling the system (2)

Example of Model: **Mealy machine** (FSM with outputs)
Simple coffee machine controller giving change, managing 2 coins

**Inputs**: \{ coffeeReq?, 25?, 50? \} (labelled with “?”)
**Outputs**: \{ nop!, 25!, 50!, giveCoffee! \} (labelled with “!”)

![Mealy machine model]

Possibility to add time (TFSM), data variables (EFSM), or both (TEFSM)
Modelling the system (2)

Example of Model: **Mealy machine** (FSM with outputs)
Simple coffee machine controller giving change, managing 2 coins

**Inputs:** \{ coffeeReq?, 25?, 50? \} (labelled with “?”)

**Outputs:** \{ nop!, 25!, 50!, giveCoffee! \} (labelled with “!”)

Possibility to add time (TFSM), data variables (EFSM), or both (TEFSM)
Modelling the system (2)

Example of Model: **Mealy machine** (FSM with outputs)
Simple coffee machine controller giving change, managing 2 coins
Inputs: \{ coffeeReq?, 25?, 50? \} (labelled with “?”)
Outputs: \{ nop!, 25!, 50!, giveCoffee! \} (labelled with “!”)

![FSM Diagram]

Possibility to add time (TFSM), data variables (EFSM), or both (TEFSM)
Modelling the system (3)

Example of Model: (IO)LTS (LTS with inputs and outputs)
A (very very) simplified digicode.
inputs (resp. outputs) labelled with “?” (resp. “!”)

Possibility to add time (TIOTS, TAIO), data variables (IOSTS)
Modelling the system (3)

Example of Model: *(IO)LTS* (LTS with inputs and outputs)
A (very very) simplified digicode.
inputs (resp. outputs) labelled with “?” (resp. “!”)

Posibility to add time (T|OTS, TA|O), data variables (I|OSTS)
Outline

1. Generalities on testing

2. Source Code Based Testing (SCBT)

3. Functional testing - Model Based Testing
   - "Historical" approaches of MBT: based on Mealy Machines
   - "Historical" approaches of MBT: based on Labelled Transition Systems

4. Conclusion
References

Part essentially based on:


Mealy Machine

\[ M = (I, O, S, \delta, \lambda) \]

where

- \( I \) and \( O \) are finite sets of inputs and outputs symbols
- \( S \) is a finite set of states,
- \( \delta : S \times I \rightarrow S \) is the state transition function, extend to input sequences with \( \delta^* : S \times I^* \rightarrow S^* \)
- \( \lambda : S \times I \rightarrow O \) is the output function, extend to output sequences with \( \lambda^* : S \times I^* \rightarrow O^* \)

→ Deterministic since here \( \delta \) and \( \lambda \) are functions
Mealy Machine

\[ \mathcal{M} = (I, O, S, \delta, \lambda) \]

- \( I \) and \( O \) are finite sets of inputs and outputs symbols.
- \( S \) is a finite set of states,
- \( \delta : S \times I \rightarrow S \) is the state transition function, extend to input sequences with \( \delta^* : S \times I^* \rightarrow S^* \)
- \( \lambda : S \times I \rightarrow O \) is the output function, extend to output sequences with \( \lambda^* : S \times I^* \rightarrow O^* \)

→ Usually complete: \( \delta \) and \( \lambda \) defined for any input
Mealy Machine

\[ M = (I, O, S, \delta, \lambda) \]

- \( I \) and \( O \) are finite sets of inputs and outputs symbols
- \( S \) is a finite set of states,
- \( \delta : S \times I \rightarrow S \) is the state transition function, extend to input sequences with \( \delta^* : S \times I^* \rightarrow S^* \)
- \( \lambda : S \times I \rightarrow O \) is the output function, extend to output sequences with \( \lambda^* : S \times I^* \rightarrow O^* \)

→ Equivalent states: two states \( s \) and \( t \) are equivalent if
\[ \forall x \in I^*, \lambda(s, x) = \lambda(t, x) \]
Mealy Machine

\[ M = (I, O, S, \delta, \lambda) \]

- **I** and **O** are finite sets of inputs and outputs symbols.
- **S** is a finite set of states.
- \( \delta : S \times I \rightarrow S \) is the state transition function, extend to input sequences with \( \delta^* : S \times I^* \rightarrow S^* \).
- \( \lambda : S \times I \rightarrow O \) is the output function, extend to output sequences with \( \lambda^* : S \times I^* \rightarrow O^* \).

→ **Minimal machine**: no pair of distinct equivalent states (possible to build a minimal machine from a non minimal one)
Conformance testing

<table>
<thead>
<tr>
<th>Problem description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specification</strong> $M_S$, Mealy machine, known</td>
</tr>
<tr>
<td><strong>Implementation</strong> $M_I$, Mealy machine, unknown, only inputs/outputs are observable</td>
</tr>
<tr>
<td><strong>Aim</strong>: Use test sequences to check if $M_I$ is equivalent to $M_S$, i.e. $M_I$ conforms to $M_S$</td>
</tr>
</tbody>
</table>

"Here “equivalent” means isomorphic"

→ generally, $M_S$ and $M_I$ supposed to be minimal and strongly connected (usually existence of a reset action)

→ $M_S$ and $M_I$ have the same number of states (usually not necessary)
Test synthesis

Fault model

Checking equivalence between $M_I$ and $M_S$ means checking if $M_I$ has no:

- **Output fault**: not the expected output for a given (state, input)
- **Transfer fault**: not the expected arrival state for a given (state, input)

Exhaustivity ⇒ any transition should be checked

Elementary test, general algorithm

For any state $s$ and any input $i$ (of the specification)

- Go to $s$
- Apply $i$, verify output $o$ (compare to the specification)
- Identify arrival state with a sequence
Test synthesis

Fault model

Checking equivalence between $M_I$ and $M_S$ means checking if $M_I$ has no:

- **Output fault**: not the expected output for a given (state, input)
- **Transfer fault**: not the expected arrival state for a given (state, input)

Exhaustivity $\implies$ any transition should be checked

Elementary test, general algorithm

For any state $s$ and any input $i$ (of the specification)

- Go to $s$
- Apply $i$, verify output $o$ (compare to the specification)
- Identify arrival state with a sequence
Identification sequences

For a given Mealy Machine $\mathcal{M} = (I, O, S, \delta, \lambda)$,

- **Distinguishing Sequence**:
  $$\exists DS \in I^*, \forall s, s' \in S : s \neq s' \Rightarrow \lambda^*(s, DS) \neq \lambda^*(s', DS)$$  (DS method [Gon70])

- **UIO\textsuperscript{2} Sequence**:
  $$\forall s \in S, \exists UIO_s \in I^*, \forall s' \in S \setminus \{s\}, \lambda^*(s, UIO_s) \neq \lambda^*(s', UIO_s)$$  (UIO method [SD88])

- **W set of $x_{ij}$ sequences**:
  $$\forall s_i, s_j \in S, \exists x_{ij} \in I^*, \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij})$$  (W method [Cho78])

$\rightarrow$ **W set** always exists in case of minimal machine, others may not exist

\textsuperscript{2}Unique Input Output
Identification sequences

For a given Mealy Machine $\mathcal{M} = (I, O, S, \delta, \lambda)$,

- **Distinguishing Sequence**: 
  $\exists DS \in I^*, \forall s, s' \in S : s \neq s' \Rightarrow \lambda^*(s, DS) \neq \lambda^*(s', DS)$ (DS method [Gon70])

- **UIO$^2$ Sequence**: 
  $\forall s \in S, \exists UIO_s \in I^*, \forall s' \in S \setminus \{s\}, \lambda^*(s, UIO_s) \neq \lambda^*(s', UIO_s)$ (UIO method [SD88])

- **W set of $x_{ij}$ sequences**: 
  $\forall s_i, s_j \in S, \exists x_{ij} \in I^*, \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij})$ (W method [Cho78])

$\rightarrow$ W set always exists in case of minimal machine, others may not exist
Identification sequences

For a given Mealy Machine $\mathcal{M} = (I, O, S, \delta, \lambda)$,

- **Distinguishing Sequence**: 
  $\exists DS \in I^*, \forall s, s' \in S : s \neq s' \Rightarrow \lambda^*(s, DS) \neq \lambda^*(s', DS)$ (DS method [Gon70])

- **UIO$^2$ Sequence**: 
  $\forall s \in S, \exists UIO_s \in I^*, \forall s' \in S \setminus \{s\}, \lambda^*(s, UIO_s) \neq \lambda^*(s', UIO_s)$ (UIO method [SD88])

- **W set** of $x_{ij}$ sequences: 
  $\forall s_i, s_j \in S, \exists x_{ij} \in I^*, \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij})$ (W method [Cho78])

$\rightarrow$ W set always exists in case of minimal machine, others may not exist
Identification sequences

For a given Mealy Machine $M = (I, O, S, \delta, \lambda)$,

- **Distinguishing Sequence** :
  $$\exists DS \in I^*, \forall s, s' \in S : s \neq s' \Rightarrow \lambda^*(s, DS) \neq \lambda^*(s', DS)$$ (DS method [Gon70])

- **UIO$^2$ Sequence** :
  $$\forall s \in S, \exists UIO_s \in I^*, \forall s' \in S \setminus \{s\}, \lambda^*(s, UIO_s) \neq \lambda^*(s', UIO_s)$$ (UIO method [SD88])

- **W set of $x_{ij}$ sequences** :
  $$\forall s_i, s_j \in S, \exists x_{ij} \in I^*, \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij})$$ (W method [Cho78])

$\rightarrow$ **W set** always exists in case of minimal machine, others may not exist
Test generation (1)

Aim: reduce the length of the test case

- **TT** [NT81]
  - Find a minimal sequence running through all transitions
  - Chinese Postman problem:
    - Transform the graph into a symmetric one in a minimal way
    - Find an Eulerian circuit

- **UIO** [SD88]
  - Find a UIO sequence for each state
  - Considering the transitions $s_i \xrightarrow{i/o} s_j \xrightarrow{UIO_{s_j}} s_k$, find a minimal circuit running once through these transitions
  - Rural Chinese Postman problem
Test generation (1)

Aim: reduce the length of the test case

- **TT** [NT81]
  - Find a minimal sequence running through all transitions
  - Chinese Postman problem:
    - Transform the graph into a symmetric one in a minimal way
    - Find an Eulerian circuit

- **UIO** [SD88]
  - Find a UIO sequence for each state
  - Considering the transitions $s_i \xrightarrow{i/o} s_j \xrightarrow{UIO} s_j \rightarrow s_k$, find a minimal circuit running once through these transitions
  - Rural Chinese Postman problem
Test generation (2)

- **DS** [Gon70]
  Could be seen as a special case of the UIO method, with the same UIO for each state

- **W** [Cho78]
  1. Find a **Transition Cover Set** $P$: set of input sequences s.t. for each state $s \in S$ and each input $a \in I$, there exists an input sequence in $P$ starting from the initial state and ending with the transition that applies $a$ to $s$.
  2. Find a **Characterising Set** $W$: set of input sequences s.t. $\forall s_i, s_j \in S, \exists x_{ij} \in W, \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij})$
  3. Noting $X \cdot Y$ the concatenation of all elements of $X$ with all elements of $Y$, generate $\{\text{reset}\} \cdot P \cdot W$

- $W_p$, $UIO_p$, $UIO_v$, $DS$ without resets, Adaptative $DS$, $HSI$, ...

- Possibility to be more efficient by using **Adaptative Sequences**
Test generation (2)

- **DS** [Gon70]
  Could be seen as a special case of the UIO method, with the same UIO for each state

- **W** [Cho78]
  Find a **Transition Cover Set** $P$: set of input sequences s.t. for each state $s \in S$ and each input $a \in I$, there exists an input sequence in $P$ starting from the initial state and ending with the transition that applies $a$ to $s$.

  Find a **Characterising Set** $W$: set of input sequences s.t.
  \[ \forall s_i, s_j \in S, \exists x_{ij} \in W, \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij}) \]

  Noting $X \cdot Y$ the concatenation of all elements of $X$ with all elements of $Y$, generate \{reset\} · $P$ · $W$

- $W_p$, $UIO_p$, $UIO_v$, $DS$ without resets, Adaptative $DS$, $HSI$, ...

- Possibility to be more efficient by using **Adaptative Sequences**

\(^3\)case where $M_I$ and $M_S$ have the same number of states
Test generation (2)

- **DS** [Gon70]
  Could be seen as a special case of the UIO method, with the same UIO for each state

- **W** [Cho78] ³

  - Find a **Transition Cover Set** $P$: set of input sequences s.t. for each state $s \in S$ and each input $a \in I$, there exists an input sequence in $P$ starting from the initial state and ending with the transition that applies $a$ to $s$.

  - Find a **Characterising Set** $W$: set of input sequences s.t. $\forall s_i, s_j \in S, \exists x_{ij} \in W ; \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij})$

  - Noting $X \cdot Y$ the concatenation of all elements of $X$ with all elements of $Y$, generate $\{\text{reset}\} \cdot P \cdot W$

- $W_p, UIO_p, UIO_v, DS$ without resets, Adaptative $DS, HSI, ...$

- Possibility to be more efficient by using **Adaptative Sequences**

³Case where $M_I$ and $M_S$ have the same number of states
Test generation (2)

- **DS** [Gon70]
  Could be seen as a special case of the UIO method, with the same UIO for each state

- **W** [Cho78]
  - Find a **Transition Cover Set** $P$: set of input sequences s.t. for each state $s \in S$ and each input $a \in I$, there exists an input sequence in $P$ starting from the initial state and ending with the transition that applies $a$ to $s$.
  - Find a **Characterising Set** $W$: set of input sequences s.t.
    \[
    \forall s_i, s_j \in S, \exists x_{ij} \in W, \lambda^*(s_i, x_{ij}) \neq \lambda^*(s_j, x_{ij})
    \]
  - Noting $X \cdot Y$ the concatenation of all elements of $X$ with all elements of $Y$, generate $\{\text{reset}\} \cdot P \cdot W$

- $W_p$, $UIO_p$, $UIO_v$, $DS$ without resets, Adaptative $DS$, $HSI$, ...

- Possibility to be more efficient by using **Adaptative Sequences**

\[^3\text{case where } M_I \text{ and } M_S \text{ have the same number of states}\]
Outline

1 Generalities on testing

2 Source Code Based Testing (SCBT)

3 Functional testing - Model Based Testing
   - "Historical" approaches of MBT: based on Mealy Machines
   - "Historical" approaches of MBT: based on Labelled Transition Systems

4 Conclusion
References

Part essentially based on:


Input Output Labelled Transition System (IOLTS)

\[ M = (Q^M, A^M, \rightarrow_M, q_0^M) \] with:

- \( Q^M \) set of states
- \( q_0^M \in Q^M \) initial state
- \( A^M \) action alphabet,
  - \( A^M_I \) input alphabet (with ?)
  - \( A^M_O \) output alphabet (with !)
  - \( I^M \) internal actions (\( \tau_k \))
- \( \rightarrow_M \subseteq Q^M \times A^M \times Q^M \) transition relation

\[ A^M_{VIS} = A^M_I \cup A^M_O \] set of visible actions
Input Output Labelled Transition System (IOLTS)

$$M = (Q, A, \rightarrow, q_0)$$ with:
- $Q$ set of states
- $q_0 \in Q$ initial state
- $A$ action alphabet,
  - $A_I$ input alphabet (with ?)
  - $A_O$ output alphabet (with !)
  - $I$ internal actions ($\tau_k$)
- $\rightarrow \subseteq Q \times A \times Q$ transition relation

$$A_{VIS} = A_I \cup A_O$$ set of visible actions
Runs / Traces

**Runs**: alternate sequences of states and actions fireable btw those states

\[ s_0 \xrightarrow{?d} s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{?d} s_3 \xrightarrow{!o} s_4 \in \text{Runs}(M) \]

**Traces**: projections of Runs on visible actions:

\[ \text{Traces}(M) = \{ \varepsilon, ?d, ?r, ?d.?r, ?r.?d, ?d.?b, ... \} \]

**P after σ**: set of states reachable from P after observation σ:

\[ \{ s_2 \} \text{ after } ?d.!o = \{ s_0, s_4 \} \]
\[ \{ s_0 \} \text{ after } ?d,!a = \emptyset \]
\[ M \text{ after } \sigma \triangleq \{ q_0 \} \text{ after } \sigma \]
Non-determinism

$M$ is deterministic if it has no internal action, and

$\forall q, q', q'' \in Q, \forall a \in A_{VIS}, (q \xrightarrow{a} q' \land q \xrightarrow{a} q'') \Rightarrow q' = q''$

Not to be confused with uncontrolled choice

$\text{Determinization: } \det(M) = (2^Q, A_{VIS}, \xrightarrow{\det}, q_0 \text{ after } \epsilon) \text{ with } P \xrightarrow{a}_{\det} P' \iff P, P' \in 2^Q, a \in A_{VIS} \text{ and } P' = P \text{ after } a.$

$\text{Traces}(M) = \text{Traces}(\det(M))$
Non-determinism

\( M \) is **deterministic** if it has no internal action, and 
\[ \forall q, q', q'' \in Q, \forall a \in AVIS, (q \xrightarrow{a} q' \land q \xrightarrow{a} q'') \Rightarrow q' = q'' \]

Not to be confused with **uncontrolled choice**

**Determinization:** 
\[ det(M) = (2^Q, AVIS, \rightarrow_{det}, q_0 \text{ after } \epsilon) \text{ with } P \xrightarrow{a}_{det} P' \iff P, P' \in 2^Q, a \in AVIS \text{ and } P' = P \text{ after } a. \]

\[ \text{Traces}(M) = \text{Traces}(det(M)) \]
Observation of quiescence

In testing practice, one can observe traces of the IUT, but also its quiescences with timers. Only quiescences of IUT unspecified in $S$ should be rejected.

\begin{equation}
\Gamma(q) \triangleq \{ a \in A \mid q \xrightarrow{a}\}
\end{equation}
Observation of quiescence

In testing practice, one can observe traces of the IUT, but also its quiescences with timers.
Only quiescences of IUT unspecified in $S$ should be rejected.

Notation: $\Gamma(q) \triangleq \{ a \in A \mid q \xrightarrow{a} \}$

- **deadlock**: no possible evolution:
  \[ \Gamma(q) = \emptyset. \]

- **outputlock**: systems waiting for an action:
  \[ \Gamma(q) \subseteq A_I. \]

- **livelock**: internal actions loop:
  \[ \exists \tau_1, \ldots, \tau_n: q \tau_1 \ldots \tau_n \rightarrow q. \]

- **quiescent** of $(M) = \text{deadlock}(M) \cup \text{livelock}(M) \cup \text{outputlock}(M)$. 

---

A. Rollet - TAROT2016 - Paris (France) - July 2016
Observation of quiescence

In testing practice, one can observe traces of the *IUT*, but also its quiescences with timers. Only quiescences of *IUT* unspecified in *S* should be rejected.

**Notation**: \( \Gamma(q) \triangleq \{ a \in A \mid q \xrightarrow{a} \} \)

*deadlock* : no possible evolution :
\[ \Gamma(q) = \emptyset. \]

*outputlock* : system waiting for an action :
\[ \Gamma(q) \subseteq A_I. \]
Observation of quiescence

In testing practice, one can observe traces of the IUT, but also its quiescences with timers. Only quiescences of IUT unspecified in $S$ should be rejected.

**Notation:**

$$\Gamma(q) \triangleq \{ a \in A \mid q \xrightarrow{a} \}$$

- **deadlock:** no possible evolution:
  $$\Gamma(q) = \emptyset.$$

- **outputlock:** system waiting for an action:
  $$\Gamma(q) \subseteq A_I.$$

- **livelock:** internal actions loop:
  $$\exists \tau_1, \ldots, \tau_n : q \xrightarrow{\tau_1 \ldots \tau_n} q.$$
Observation of quiescence

In testing practice, one can observe traces of the *IUT*, but also its **quiescences** with timers.
Only quiescences of *IUT* unspecified in *S* should be rejected.

### Notation:

\[
\Gamma(q) \triangleq \{ a \in A \mid q \xrightarrow{a} \}
\]

- **deadlock**: no possible evolution:
  \[\Gamma(q) = \emptyset.\]
- **outputlock**: system waiting for an action:
  \[\Gamma(q) \subseteq A_I.\]
- **livelock**: internal actions loop:
  \[\exists \tau_1, \ldots, \tau_n : q \xrightarrow{\tau_1 \ldots \tau_n} q.\]

\[
\text{quiescent}(M) = \text{deadlock}(M) \cup \text{livelock}(M) \cup \text{outputlock}(M)
\]
Suspension automaton

**Quiescence**: special output $\delta$

**Suspension automaton** $\Delta(M)$

$\Delta(M) = \text{Specification } M + \delta$-transitions on quiescent states

**Suspension traces**

$S\text{Traces}(M) \triangleq \text{Traces}(\Delta(M)) = \text{Traces}(\text{det}(\Delta(M)))$
Testing framework

**Specification** : ioLTS $S = (Q^S, A^S, \rightarrow_S, s_0^S)$

**Implementation** : ioLTS $IUT = (Q^{IUT}, A^{IUT}, \rightarrow_{IUT}, s_0^{IUT})$

Unknown implementation, except for its interface, identical to $S$’s

**Hyp.** : $IUT$ is input-complete : In any state, $IUT$ accepts any input, possibly after internal actions.
Conformance relation

The conformance relation defines the set of implementations \( IUT \) conforming to \( S \).

\[
IUT \; \text{ioco} \; S \; \triangleq \; \forall \sigma \in S\text{Traces}(S), \; \text{Out}(\Delta(IUT) \; \text{after} \; \sigma) \subseteq \text{Out}(\Delta(S) \; \text{after} \; \sigma)
\]

with \( \text{Out}(P) \; \triangleq \; \Gamma(P) \cap A_{\delta} \) \( ^{a} \): set of outputs \( \wedge \) quiescences in \( P \).

\( ^{a} \) \( A_{\delta} \) is equivalent notation for \( A_{O} \) since \( \delta \) is an output of \( \Delta(S) \) and \( \Delta(IUT) \).

Intuition: \( IUT \) conforms to \( S \) iff after any suspension trace of \( S \) and \( IUT \), all outputs and quiescences of \( IUT \) are specified by \( S \).
ioco: example

specification $\Delta(S)$

$I_1$: Implem. choice

$I_2$: Implem. of a partial spec.

$I_3$: Unspec. output

$I_4$: Unspec. quiescence
Canonical Tester

From \( S \) (more precisely from \( \text{det}(\Delta(S)) = (Q^d, A^d, \rightarrow_d, q_0^d) \)), build an ioLTS \( \text{Can}(S) = (Q^c, A^c, \rightarrow_c, q_0^c) \) → the most general ioLTS detecting non-conformance of implementation \( IUT \).
Canonical Tester

From $S$ (more precisely from $\text{det}(\Delta(S)) = (Q^d, A^d, \rightarrow_d, q_0^d)$), build an ioLTS $\text{Can}(S) = (Q^c, A^c, \rightarrow_c, q_0^c)$
→ the most general ioLTS detecting non-conformance of implementation $IUT$.

From $\text{det}(\Delta(S))$:
- Invert inputs and outputs (tester point of view)
- All non-specified outputs lead to Fail.
Canonical Tester

From $S$ (more precisely from $\text{det}(\Delta(S)) = (Q^d, A^d, \rightarrow_d, q_0^d)$), build an ioLTS $\text{Can}(S) = (Q^c, A^c, \rightarrow_c, q_0^c)$ → the most general ioLTS detecting non-conformance of implementation $IUT$.
From $S$ (more precisely from $\text{det}(\Delta(S)) = (Q^d, A^d, \rightarrow_d, q_0^d)$), build an ioLTS $\text{Can}(S) = (Q^c, A^c, \rightarrow_c, q_0^c)$ → the most general ioLTS detecting non-conformance of implementation $IUT$. 
Canonical Tester

From $S$ (more precisely from $\text{det}(\Delta(S)) = (Q^d, A^d, \rightarrow_d, q^d_0)$), build an ioLTS $\text{Can}(S) = (Q^c, A^c, \rightarrow_c, q^c_0)$ → the most general ioLTS detecting non-conformance of implementation $IUT$. 
Test cases

A test case is a deterministic ioLTS $(Q^{TC}, A^{TC}, \rightarrow^{TC}, t_0^{TC})$, equipped with **verdict** states: **Pass**, **Fail** and **Inconc** s.t.

- $TC$ follows the tester point of view (input / output inversion)
- $TC$ is **controllable**, i.e. never have to choose btw. several outputs or btw. inputs and outputs:
- All states with an input, are **input-complete**, except verdict states.

Test execution = parallel composition $TC \parallel \Delta(IUT)$ synchronizing on common visible actions
A test case is a deterministic iOLTS $(Q^{TC}, A^{TC}, \rightarrow_{TC}, t_0^{TC})$, equipped with verdict states: Pass, Fail and Inconc s.t.

- $TC$ follows the tester point of view (input / output inversion)
- $TC$ is controllable, i.e. never have to choose btw. several outputs or btw. inputs and outputs:
- All states with an input, are input-complete, except verdict states.

Test execution = parallel composition $TC \| \triangle(IUT)$ synchronizing on common visible actions
**Properties of test suites**

\(TC\) fails \(IUT\) iff an execution of \(TC \parallel \Delta(IUT)\) reaches **Fail**

Expresses a **possibility** for rejection.

→ a single test case may lead to several different verdicts

**Soundness, Exhaustiveness, Completeness**

A set of test cases \(TS\) is

- **Sound** ⇔
  \[
  \forall IUT : (IUT \ ioco \ S \implies \forall TC \in TS : \neg (TC \ fails \ IUT)),
  \]
  i.e. only non-conformant \(IUT\) may be rejected by a \(TC \in TS\).

- **Exhaustive** ⇔
  \[
  \forall IUT : (\neg (IUT \ ioco \ S) \implies \exists TC \in TS : TC \ fails \ IUT),
  \]
  i.e. any non-conformant \(IUT\) may be rejected by a \(TC \in TS\).

- **Complete** = Sound and Exhaustive
Properties of test suites

\( TC \) fails \( IUT \) iff an execution of \( TC \| \Delta(IUT) \) reaches \textbf{Fail}.

Expresses a \textit{possibility} for rejection.

→ a single test case may lead to several different verdicts.

Soundness, Exhaustiveness, Completeness

A set of test cases \( TS \) is

- \textit{Sound} \( \triangleq \)
  \[ \forall IUT : (IUT \ not\ conformed \ S ) \implies \forall TC \in TS : \neg(TC \ fails \ IUT) \],
  i.e. only non-conformant \( IUT \) may be rejected by a \( TC \in TS \).

- \textit{Exhaustive} \( \triangleq \)
  \[ \forall IUT : (\neg(IUT \ not\ conformed \ S)) \implies \exists TC \in TS : TC \ fails \ IUT \],
  i.e. any non-conformant \( IUT \) may be rejected by a \( TC \in TS \).

- \textit{Complete} = Sound and Exhaustive
Test selection

Objective: Find an algorithm taking as input a finite state ioLTS $S$, and satisfying the following properties:

- Produces only sound test suites
- Is limit-exhaustive i.e. the infinite suite of test cases that can be produced is exhaustive

Two techniques:
1. Non-deterministic selection (TorX)
2. Selection guided by a test purpose (TGV)
Non-deterministic selection

Algorithm: partial unfolding of $Can(S)$

Start in $q_0^c$. After any trace $\sigma$ in $Can(S)$

- If $Can(S)$ after $\sigma \subseteq \text{Fail}$, emit a Fail verdict
- Otherwise make a choice between
  - Produce a Pass verdict and stop,
  - Consider all inputs of $Can(S)$ after $\sigma$ and continue.
  - Choose one output in those of $Can(S)$ after $\sigma$ and continue.

Properties

$TS = \text{all possible Test cases generated with this algorithm :}$

$TS$ is sound and limit-exhaustive
Non-deterministic selection

Algorithm: partial unfolding of $\text{Can}(S)$

Start in $q_0^c$. After any trace $\sigma$ in $\text{Can}(S)$

- If $\text{Can}(S)$ after $\sigma \subseteq \text{Fail}$, emit a Fail verdict
- Otherwise make a choice between
  - Produce a Pass verdict and stop,
  - Consider all inputs of $\text{Can}(S)$ after $\sigma$ and continue.
  - Choose one output in those of $\text{Can}(S)$ after $\sigma$ and continue.

Properties

$TS =$ all possible Test cases generated with this algorithm:
$TS$ is sound and limit-exhaustive
Example

Can(S)

TC1
Test Purpose generation

Previous algorithm: maybe quite long if we intend to focus on a specific behavior...

Main characteristics of Test Purpose Generation:
- Test selection by test purposes describing a set of behaviors to be tested, targeted by a test case
Test Purpose definition

Test Purpose

Deterministic and complete ioLTS $TP = (Q^{TP}, A^{TP}, \rightarrow^{TP}, q_0^{TP})$

equipped with two sets $\text{Accept}^{TP}$ and $\text{Refuse}^{TP}$ of trap states

Can($S$)
Selection principle

Can(S) x TP

Fail

Can(S)

det(Δ(S))

? othw

Fail

TP

Accept

Accept

TC

Fail

Inconce

Pass
Synchronous Product: definition

Definition of Synchronous Product

The **Synchronous Product** of two ioLTS $M_1 = (Q^{M_1}, A, \rightarrow_{M_1}, q_{0}^{M_1})$, and $M_2 = (Q^{M_2}, A, \rightarrow_{M_2}, q_{0}^{M_2})$ is the ioLTS $M_1 \times M_2 = (Q^{M_1} \times Q^{M_2}, A, \rightarrow, q_{0}^{M_1} \times q_{0}^{M_2})$ where $\rightarrow$ is defined by:

$$(q_{M_1}, q_{M_2}) \xrightarrow{a} (q'_{M_1}, q'_{M_2}) \iff (q_{M_1} \xrightarrow{a}_{M_1} q'_{M_1}) \land (q_{M_2} \xrightarrow{a}_{M_2} q'_{M_2})$$
The Synchronous Product $Can(S) \times TP$
Complete Test Graph (CTG)

Co-reachability analysis:
- Keep the first *Accept* state in a path → *Pass*
- If $q \in \text{coreach}(\text{Pass})$ keep $q$
- If $q \in \{\text{Fail}\}$ keep $q$
- If $q \not\in \text{coreach}(\text{Pass})$ input (tester point of view) is successor of a state $q' \in \text{coreach}(\text{Pass})$ then *Inconc*
Ensuring controllability of test cases

Example of Test Case

The test suite composed of the set of test cases that the algorithm can produce is sound and limit-exhaustive.
If we summarize MBT...

- Two “historical” approaches of MBT: based on ioLTS and Mealy Machines theory

- Today, many other approaches exist, with various describing formats
  (e.g. extensions of FSM and LTS, UML, SysML, Markov chains, Simulink, Lustre, ...)

- Many tools are available.
  → A (non-exhaustive, but yet interesting) list may be found here:
  http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html
Outline

1. Generalities on testing
2. Source Code Based Testing (SCBT)
3. Functional testing - Model Based Testing
   - "Historical" approaches of MBT: based on Mealy Machines
   - "Historical" approaches of MBT: based on Labelled Transition Systems
4. Conclusion
Concluding remarks

- Very active domain
- Many issues, both theoretical and practical
- Still a lot to do
- Huge industrial needs

Perspectives

- Many!
- See the following presentations of this Summer School!
Concluding remarks

- Very active domain
- Many issues, both theoretical and practical
- Still a lot to do
- Huge industrial needs

Perspectives

- Many!
- See the following presentations of this Summer School!
Thank you for your attention

rollet@labri.fr
Other References (CBT)


Other References (MBT)


