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[a]
- An exhaustive presentation
- Advanced research, more a general view on the topic

[a] implying that I am not a specialist of all of the presented topics!

Outline

Outline

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.

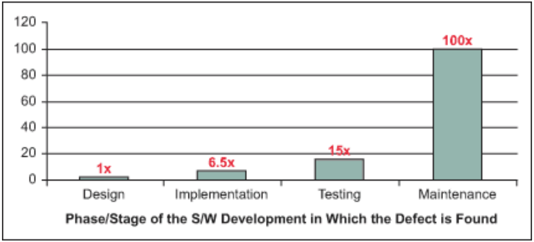
Why testing?

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)



Source : IBM Systems Sciences Institute

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)

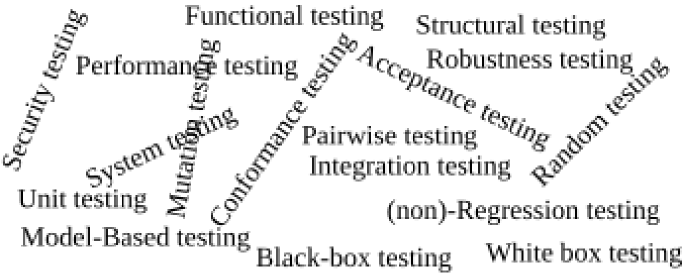
($\neq$ Static testing)

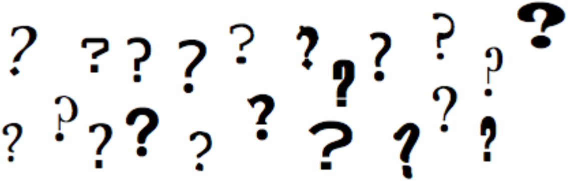
$\rightarrow$ this presentation will focus mainly on dynamic testing techniques

[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"[a]

[a] http://www.informationweek.com/story/IWK20020517S0011

$\Longrightarrow$ 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 $\rightarrow$ 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

$\Longrightarrow$ 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 $\rightarrow$ 583000 years...

Difficulty(ies) of testing (2)

Dijkstra :

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

- $\Longrightarrow$ 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)

Difficulty(ies) of testing (2)

Dijkstra :

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

- $\Longrightarrow$ 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, ...

$\rightarrow$ 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 $\rightarrow$ 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

Three dimensions of testing (Tretmans)

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

$\rightarrow$ 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 $\rightarrow$ Model Based Testing
- Structural Testing : TD is generated using the "structure" of the SUT, generally the Source Code (Code Based Testing)

$\rightarrow$ 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 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, ...

The previous classification will be used in this presentation

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

(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

Outline

References

Part essentially based on :

- [BH09] S. Bardin and P. Herrman, "Pruning the Search Space in Path-based Test Generation", *ICST 09*
- [Got10] A. Gotlieb, "Constraint-Based testing", presentation at Uppsala University, May 2010.
- [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 :

$$\textit{all statements} < \ldots < \textit{all executions}$$

Control Flow Graph

CFG

Directed Graph representing the possible paths of the program

- 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 :
  $\rightarrow$ risk of adding unfeasible paths

Control Flow Graph

CFG

Directed Graph representing the possible paths of the program

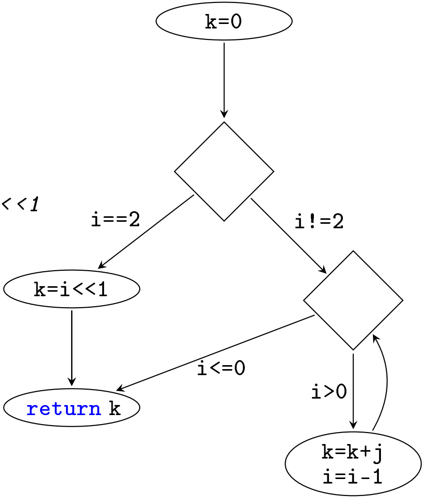
- Built from the source code
- A test may be seen as a path in the CFG
- Direct link between code coverage and CFG coverage
- Easy to obtain, e.g. with gcc :

```
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 → 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 → 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 φ, such that solving φ provides a TD

$\longrightarrow$ Does not solve the oracle problem.

$\longrightarrow$ 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 φ, such that solving φ provides a TD

$\rightarrow$ Does not solve the oracle problem.

$\rightarrow$ Pointers may lead to difficult problems

Path Predicate

Path predicate

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

$\rightarrow$ Find a solution (if any) to φ_Π in order to activate Π

- Need to remember the values of variables along the path
  $\rightarrow$ need to rename each variable in case of assignment
  $\rightarrow$ SSA[1] form

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

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

[1](Single Static Assignment)

Path predicate example

```
1  read (y,z);        ->   y0, z0 as inputs
2  y = y + 2;         ->   y1 = y0 + 2
3  x = y + 4;         ->   x1 = y1 + 4
4  if (x > 2 * z)     ->   x1 > 2 * z0
                      or x1 <= 2 * z0 depending on path
5      x = y + 2;     ->   x2 = y1 + 2
```

For the path : $1 \rightarrow 2 \rightarrow 3 \rightarrow (4, true) \rightarrow 5$,
the corresponding predicate is :
$y_1 = y_0 + 2 \wedge x_1 = y_1 + 4 \wedge x_1 > 2 * z_0 \wedge x_2 = y_1 + 2.$
Considering the inputs, we have $y_0 + 6 > 2 * z_0$
Example of TD : $y = 0, z = 0$

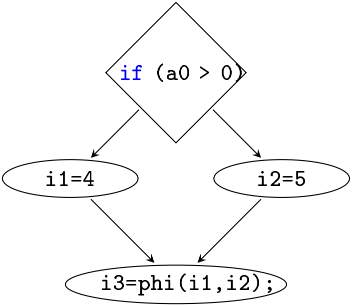
Path predicate example

```
1  read (y,z);        ->   y0, z0 as inputs
2  y = y + 2;         ->   y1 = y0 + 2
3  x = y + 4;         ->   x1 = y1 + 4
4  if (x > 2 * z)     ->   x1 > 2 * z0
                      or x1 <= 2 * z0 depending on path
5      x = y + 2;     ->   x2 = y1 + 2
```

For the path : $1 \rightarrow 2 \rightarrow 3 \rightarrow (4, true) \rightarrow 5$,
the corresponding predicate is :
$y_1 = y_0 + 2 \wedge x_1 = y_1 + 4 \wedge x_1 > 2 * z_0 \wedge x_2 = y_1 + 2$.
Considering the inputs, we have $y_0 + 6 > 2 * z_0$
Example of TD : $y = 0, z = 0$

if/then ; Φ-expressions, SSA form

```
if (a>0)     ->     if (a0 > 0)
  i=4;       ->       i1=4;
else         ->     else
  i=5;       ->       i2=5;
             ->     i3=phi(i1,i2);
```



Φ-expr $\rightarrow$ decide the value of $i3$ according to the path used to reach it

- $a_0 > 0 \implies i_1 = 4 \wedge i_3 = i_1$
- $\neg(a_0 > 0) \implies i_2 = 5 \wedge i_3 = i_2$

Choice of path (join operator)

$$\text{join}(a_0 > 0 \wedge i_1 = 4 \wedge i_3 = i_1, \neg(a_0 > 0) \wedge i_2 = 5 \wedge i_3 = i_2)$$

While; Φ-expressions, SSA form

`While` : Φ-expr added just before the decision

Initial code	SSA code
`x = 1;`	`x1 = 1;`
	`x2 = phi(x1, x3);`
`while ( x != 10 ) {`	`while (x2 != 10) {`
`  c = x;`	`  c = x2;`
`  x = x + 1;`	`  x3 = x2 + 1;`
`}`	`}`
`print(x);`	`print(x2);`

→ but need to solve a constraint according to the number of loops desired

→ each loop turn $\Longrightarrow$ new recursive constraints

Symbolic Depth First Search (DFS)

Path Based TD generation

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

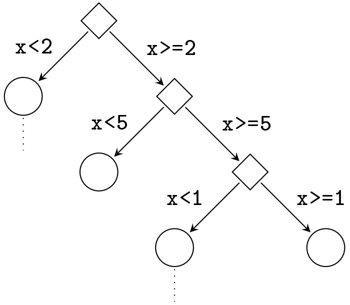
A strategy for the coverage criterion All paths :

- 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, ...).

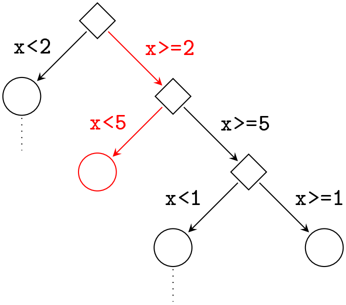
Concolic approach

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



Concolic approach

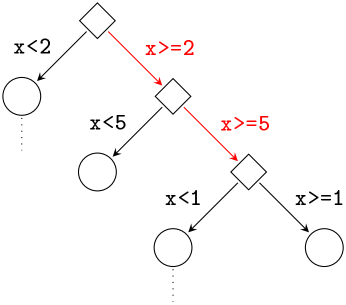
Idea : accelerate symbolic execution by using concrete execution at the same time. → Permits to select feasible paths.



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

Concolic approach

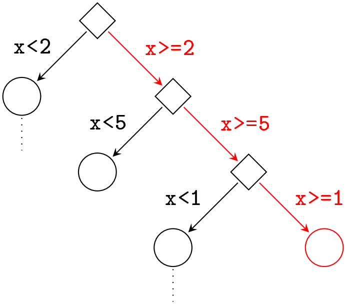
Idea : accelerate symbolic execution by using concrete execution at the same time. $\rightarrow$ Permits to select feasible paths.



Backtrack + resolution : $x \geq 2 \wedge x \geq 5$; possible solution : $x = 8$

Concolic approach

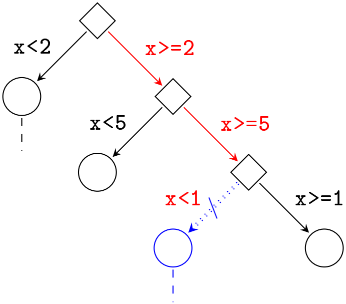
Idea : accelerate symbolic execution by using concrete execution at the same time. → Permits to select feasible paths.



x=8 (concrete)

Concolic approach

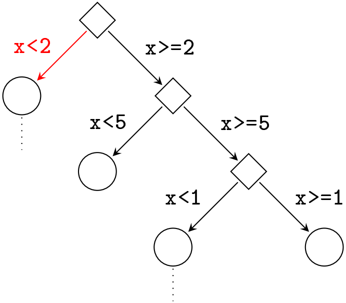
Idea : accelerate symbolic execution by using concrete execution at the same time. $\rightarrow$ Permits to select feasible paths.



Backtrack + resolution : $x \geq 2 \wedge x \geq 5 \wedge x < 1$; 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 $\Longrightarrow$ 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

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 :

V1	V2	V3	
0	0	0	
0	0	1	← redundant, remove test case
0	1	0	← redundant, remove test case
...			

[a] www.pairwise.org

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

Functional testing

TD is generated using the specification of the SUT

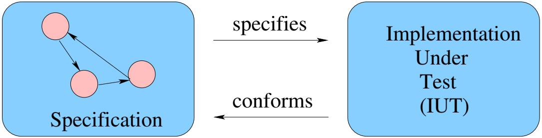
Example of known methods :

- Model Based Testing
  - Powerful technique
  - Particularly adapted for testing reactive systems, communication protocols
  - Not easy to have a (formal) model in practice
    ☹ $\leftarrow$ 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

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

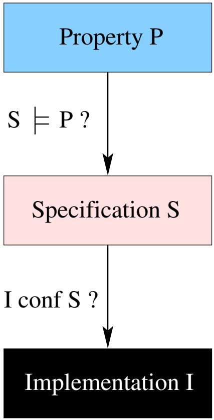
What about Model Based Testing? (2)

- 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

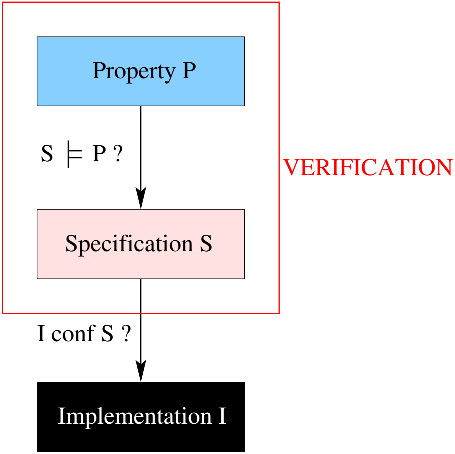
Two historical approaches of MBT of reactive systems :

- Finite State Machines
- Labeled Transition Systems

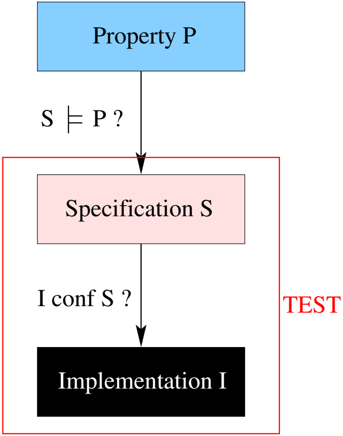
General schema



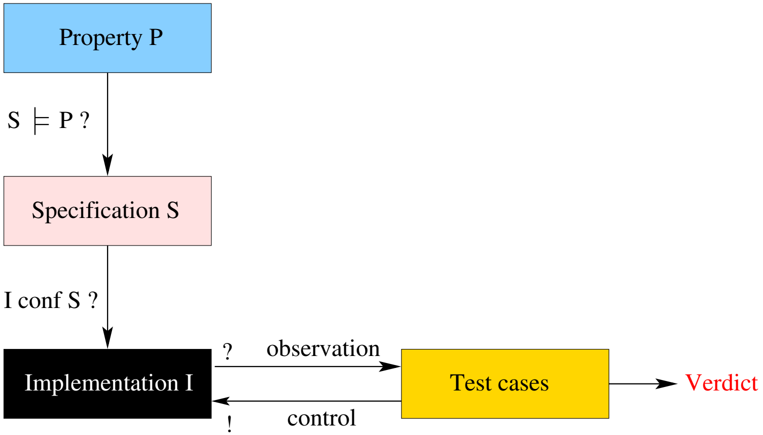
General schema



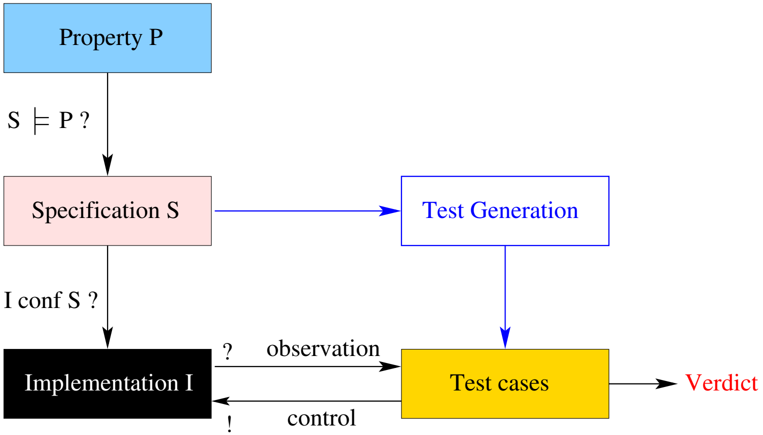
General schema



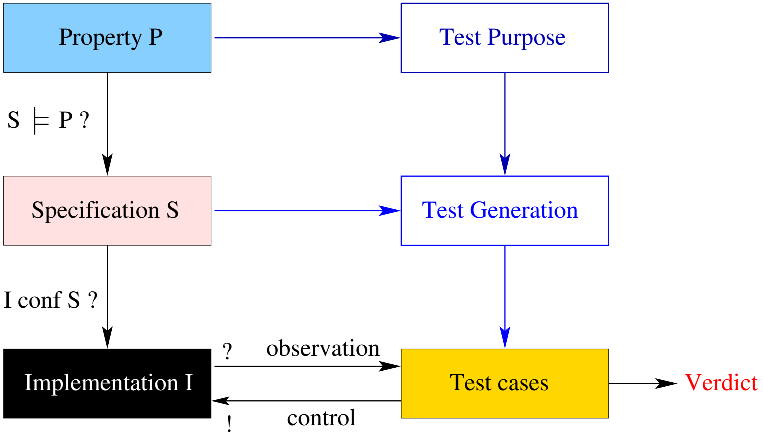
General schema



General schema



General schema



Modelling the system (1)

Find a "good" model

- Necessity to find a formal model adapted to the description of the specification
- Too abstract $\Longrightarrow$ not realistic, no interest
- Too detailed $\Longrightarrow$ 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

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 "!")

Possibility to add time (TFSM), data variables (EFSM), or both (TEFSM)

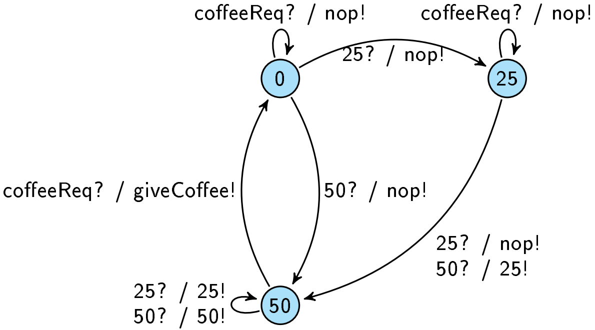
Modelling the system (2)

Example of Model : Mealy machine (FSM with outputs)

Simple coffee machine controller giving change, managing 2 coins

Inputs : { coffeeReq?, 25?, 50? } (labelled with "?")

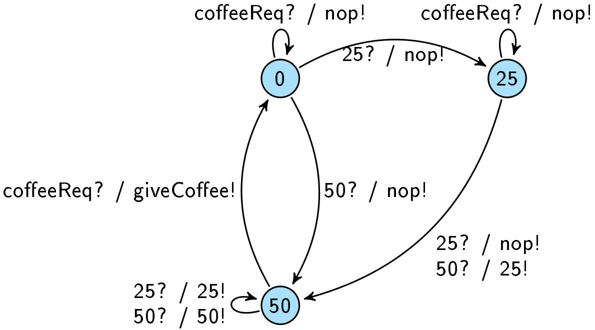
Outputs : { nop!, 25!, 50!, giveCoffee! } (labelled with "!")



Possibility to add time (TFSM), data variables (EFSM), or both (TEFSM)

Modelling the system (2)

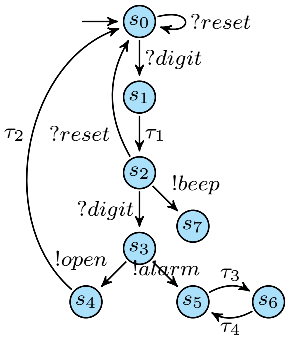
Example of Model : Mealy machine (FSM with outputs)
Simple coffee machine controller giving change, managing 2 coins
Inputs : { coffeeReq?, 25?, 50? } (labelled with "?")
Outputs : { nop!, 25!, 50!, giveCoffee! } (labelled with "!")



Possibility to add time (TFSM), data variables (EFSM), or both (TEFSM)

Modelling the system (3)

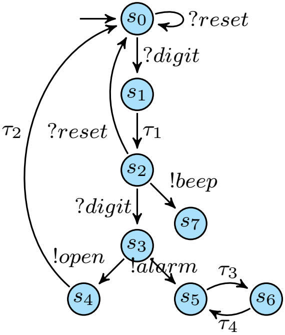
Example of Model : (IO)LTS (LTS with inputs and outputs)
A (very very) simplified digicode.
inputs (resp. outputs) labelled with “?” (resp. “!”)



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. "!")



Possibility to add time (TIOTS, TAIO), data variables (IOSTS)

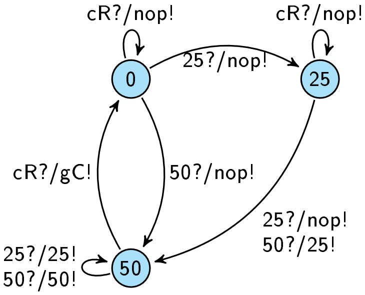
Outline

References

Part essentially based on :

- [BJK+05] Broy, M.; Jonsson, B.; Katoen, J.-P.; Leucker, M., Pretschner, A. (Eds.), "Model-Based Testing of Reactive Systems", *Springer, LNCS, volume 3472*
- [BP94] G. v. Bochmann and Alexandre Petrenko, "Protocol Testing : Review of Methods and Relevance for Software Testing" In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, Seattle, Washington, United States, p 109 - 124, 1994
- [Jer04] T. Jéron, "Contribution à la génération automatique de tests pour les systèmes réactifs," 2004, habilitation à Diriger des Recherches - Université de Rennes 1.

Mealy Machine

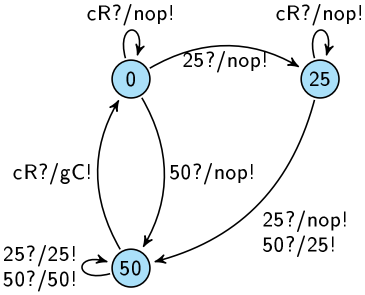


$\mathcal{M} = (I, O, S, \delta, \lambda)$ where

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

$\to$ Deterministic since here δ and λ are functions

Mealy Machine

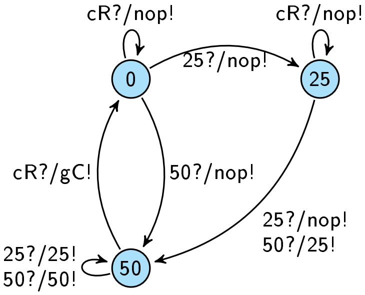


$\mathcal{M} = (I, O, S, \delta, \lambda)$ where

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

$\to$ Usually complete : δ and λ defined for any input

Mealy Machine

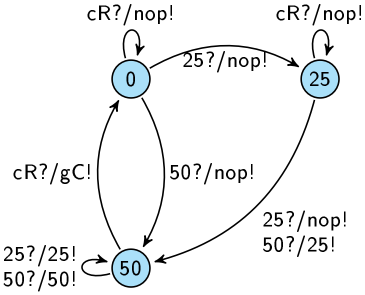


$\mathcal{M} = (I, O, S, \delta, \lambda)$ where

- 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 $\lambda^* : S \times I^* \rightarrow O^*$

$\rightarrow$ Equivalent states : two states s and t are equivalent if $\forall x \in I^*, \lambda(s, x) = \lambda(t, x)$

Mealy Machine



$\mathcal{M} = (I, O, S, \delta, \lambda)$ where

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

$\rightarrow$ Minimal machine : no pair of distinct equivalent states (possible to build a minimal machine from a non minimal one)

Conformance testing

Problem description

- Specification M_S, Mealy machine, known
- Implementation M_I Mealy machine, unknown, only inputs/outputs are observable
- Aim : Use test sequences to check if M_I is equivalent [a] to M_S, i.e. M_I *conforms to* M_S

[a] 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 $\Longrightarrow$ 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 $\Longrightarrow$ 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[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

[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

[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

[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

[2] Unique Input Output

Test generation (1)

Aim : reduce the length of the test case

- TT [NT81]
  Find a minimal sequence running through all transitions
  $\rightarrow$ 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
    $\rightarrow$ 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
  $\rightarrow$ 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 $s_i \overset{i/o}{\rightarrow} s_j \overset{UIO_{s_j}}{\rightarrow} s_k$, find a minimal circuit running once through these transitions
    $\rightarrow$ 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] [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 $\{reset\} \cdot P \cdot W$
- W_p, UIO_p, UIO_v, DS without resets, Adaptative DS, HSI, ...
- Possibility to be more efficient by using Adaptative Sequences

[3] 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 $\{reset\} \cdot P \cdot W$
- W_p, UIO_p, UIO_v, DS without resets, Adaptative DS, HSI, ...
- Possibility to be more efficient by using Adaptative Sequences

[3] 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 $\{reset\} \cdot P \cdot W$
- W_p, UIO_p, UIO_v, DS without resets, Adaptative DS, HSI, ...
- Possibility to be more efficient by using Adaptative Sequences

[3] 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 $\{reset\} \cdot P \cdot W$
- W_p, UIO_p, UIO_v, DS without resets, Adaptative DS, HSI, ...
- Possibility to be more efficient by using Adaptative Sequences

[3] case where M_I and M_S have the same number of states

Outline

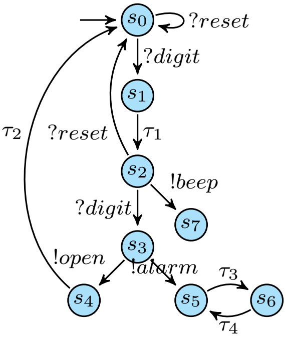
1. Generalities on testing
2. Source Code Based Testing (SCBT)
3. Functional testing - Model Based Testing
   - "Historical" approaches of MBT : based on Mealy Machines
   - "Historical" approaches of MBT : based on Labelled Transition Systems
4. Conclusion

References

Part essentially based on :

- [Tre96] J. Tretmans, "Test generation with inputs, outputs, and repetitive quiescence," *Software–Concepts and Tools*, vol. 17, pp. 103–120, 1996.
- [JJ04] C. Jard and T. Jéron, "Tgv: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Software Tools for Technology Transfer (STTT)*, 10 2004.
- [Jer04] T. Jéron, "Contribution à la génération automatique de tests pour les systèmes réactifs," 2004, habilitation à Diriger des Recherches - Université de Rennes 1.
- [Jer12] T. Jéron, Model Based Testing courses.

Input Output Labelled Transition System (IOLTS)

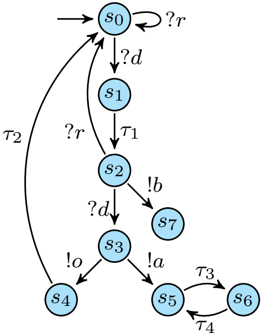


$M = (Q^M, A^M, \longrightarrow_M, q_0^M)$ with :

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

$A_{VIS}^M = A_I^M \cup A_O^M$ set of visible actions

Input Output Labelled Transition System (IOLTS)

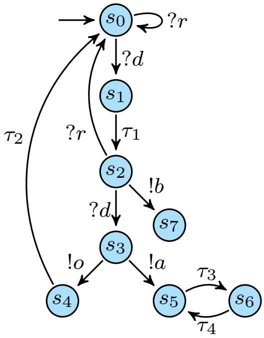


$M = (Q, A, \longrightarrow, q_0)$ with :

- 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 (τ_k)
- $\longrightarrow \subseteq Q \times A \times Q$ 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

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

Traces: projections of $Runs$ on visible actions:

$Traces(M) = \{\varepsilon, ?d, ?r, ?d.?r, ?r.?d, ?d.!b, \ldots\}$

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

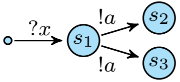
$\{s_2\}$ after $?d.!o = \{s_0, s_4\}$

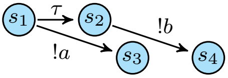
$\{s_0\}$ after $?d, !a = \emptyset$

M after $\sigma \triangleq \{q_0\}$ after σ

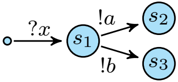
Non-determinism

M is deterministic if it has no internal action,
and $\forall q, q', q'' \in Q, \forall a \in A_{VIS}, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \Rightarrow q' = q''$





Not to be confused with uncontrolled choice

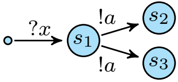


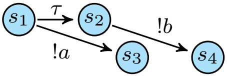
Determinization: $det(M) = (2^Q, A_{VIS}, \longrightarrow_{det}, q_0 \text{ after } \epsilon)$ with $P \xrightarrow{a}_{det} P' \Leftrightarrow P, P' \in 2^Q, a \in A_{VIS}$ and $P' = P$ after a.

$$Traces(M) = Traces(det(M))$$

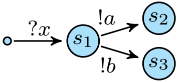
Non-determinism

M is deterministic if it has no internal action,
and $\forall q, q', q'' \in Q, \forall a \in A_{VIS}, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \Rightarrow q' = q''$





Not to be confused with uncontrolled choice

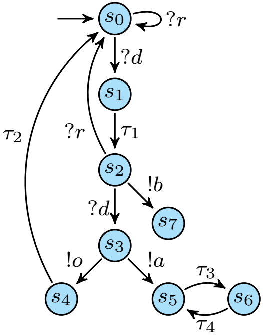


Determinization: $det(M) = (2^Q, A_{VIS}, \longrightarrow_{det}, q_0$ after $\epsilon)$ with
$P \xrightarrow{a}_{det} P' \Leftrightarrow P, P' \in 2^Q, a \in A_{VIS}$ and $P' = P$ after a.

$$Traces(M) = Traces(det(M))$$

Observation of quiescence

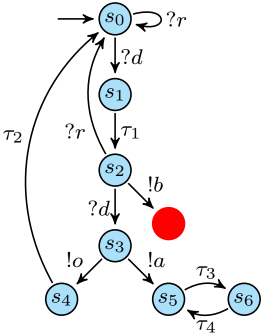
In testing practice, one can observe traces of the IUT, but also its quiescences with timers.
Only quiescences of IUT unspecified in S should be rejected.



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

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.

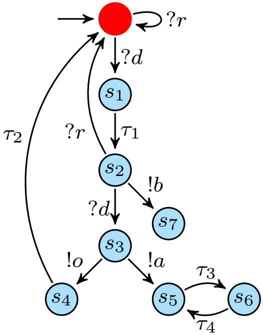


Notation : $\Gamma(q) \triangleq \{a \in A \mid q \xrightarrow{a}\}$
deadlock : no possible evolution :
$\Gamma(q) = \emptyset$.

.

Observation of quiescence

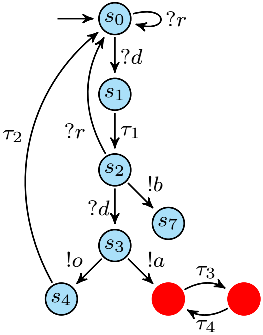
In testing practice, one can observe traces of the IUT, but also its quiescences with timers.
Only quiescences of IUT unspecified in S should be rejected.



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

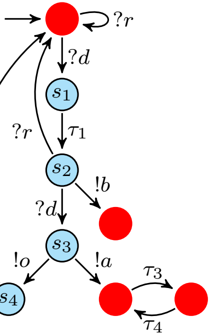
In testing practice, one can observe traces of the IUT, but also its quiescences with timers.
Only quiescences of IUT unspecified in S should be rejected.



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 :
$\exists \tau_1, \dots \tau_n : q \xrightarrow{\tau_1 \dots \tau_n} q$.

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 :
$\exists \tau_1, \ldots \tau_n : q \xrightarrow{\tau_1 \ldots \tau_n} q$.

$$quiescent(M) = deadlock(M) \cup livelock(M) \cup outputlock(M)$$

Suspension automaton

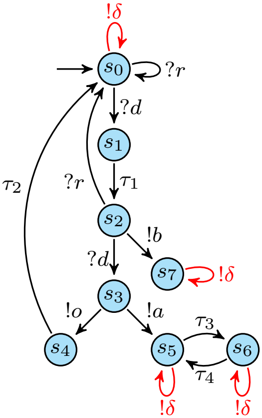
Quiescence : special output δ

Suspension automaton $\Delta(M)$

$\Delta(M)$ = Specification M + δ-transitions on quiescent states

Suspension traces

$STraces(M) \triangleq Traces(\Delta(M)) = Traces(det(\Delta(M)))$



Testing framework

Specification : ioLTS $S = (Q^{S}, A^{S}, \longrightarrow_{S}, s_0^{S})$

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

Conformance

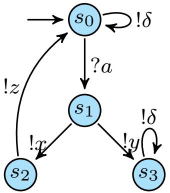
$$IUT \textbf{ ioco } S \triangleq \begin{array}{l} \forall \sigma \in STraces(S), \\ Out(\Delta(IUT) \text{ after } \sigma) \subseteq Out(\Delta(S) \text{ after } \sigma) \end{array}$$

with $Out(P) \triangleq \Gamma(P) \cap A_O^\delta$ [a]: set of outputs $\wedge$ quiescences in P.

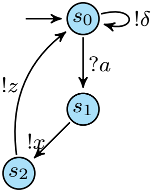
[a] A_O^δ is equivalent notation for A_O since δ is an output of $\Delta(S)$ and $\Delta(IUT)$

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

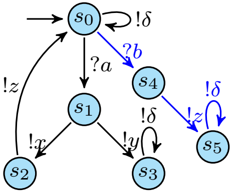
ioco: example



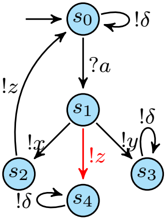
specification $\Delta(S)$



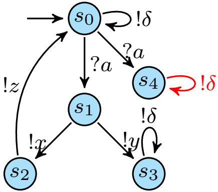
I_1: Implem. choice



I_2: Implem. of a partial spec.



I_3: Unspec. output



I_4 : Unspec. quiescence

Canonical Tester

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

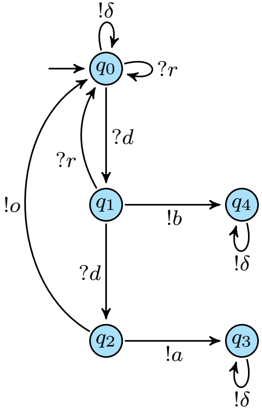
Canonical Tester

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

- Invert inputs and outputs (tester point of view)
- All non-specified outputs lead to **Fail**.

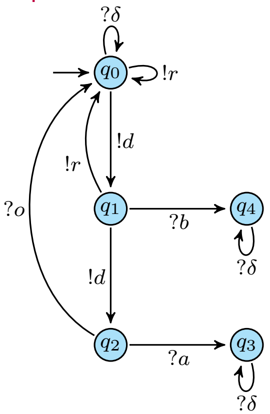
Canonical Tester

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



Canonical Tester

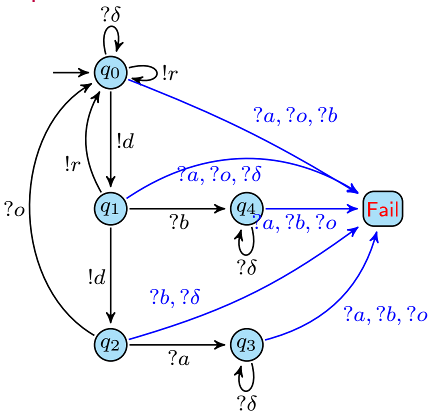
From S (more precisely from $det(\Delta(S)) = (Q^{\mathsf{d}}, A^{\mathsf{d}}, \longrightarrow_{\mathsf{d}}, q_0^{\mathsf{d}})$),
build an ioLTS $Can(S) = (Q^{\mathsf{c}}, A^{\mathsf{c}}, \longrightarrow_{\mathsf{c}}, q_0^{\mathsf{c}})$
$\rightarrow$ the most general ioLTS detecting non-conformance of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



Canonical Tester

From S (more precisely from $det(\Delta(S)) = (Q^{\mathsf{d}}, A^{\mathsf{d}}, \longrightarrow_{\mathsf{d}}, q_0^{\mathsf{d}})$), build an ioLTS $Can(S) = (Q^{\mathsf{c}}, A^{\mathsf{c}}, \longrightarrow_{\mathsf{c}}, q_0^{\mathsf{c}})$

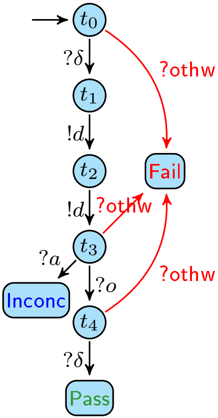
$\rightarrow$ the most general ioLTS detecting non-conformance of implementation IUT.



Test cases

A test case is a deterministic ioLTS $(Q^{TC}, A^{TC}, \longrightarrow_{TC}, t_0^{TC})$, 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.

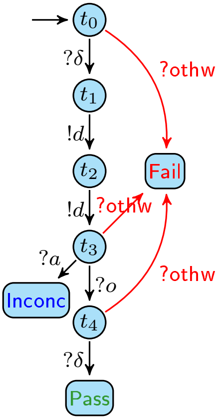


Test execution = parallel composition $TC \| \Delta(IUT)$ synchronizing on common visible actions

Test cases

A test case is a deterministic ioLTS $(Q^{\mathsf{TC}}, A^{\mathsf{TC}}, \longrightarrow_{\mathsf{TC}}, t_0^{\mathsf{TC}})$, 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 $TC \| \Delta(IUT)$ synchronizing on common visible actions

Properties of test suites

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

Expresses a *possibility* for rejection.

$\rightarrow$ a single test case may lead to several different verdicts

Soundness, Exhaustiveness, Completeness

A set of test cases TS is

- *Sound* $\triangleq$
  $\forall IUT : (IUT \textbf{ ioco } S \implies \forall TC \in TS : \neg(TC \ fails \ IUT))$,
  i.e. only non-conformant IUT may be rejected by a $TC \in TS$.
- *Exhaustive* $\triangleq$
  $\forall IUT : (\neg(IUT \textbf{ ioco } S) \implies \exists TC \in TS : TC \ fails \ IUT)$,
  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 **Fail**

Expresses a *possibility* for rejection.

$\rightarrow$ a single test case may lead to several different verdicts

Soundness, Exhaustiveness, Completeness

A set of test cases TS is

- *Sound* $\triangleq$
  $\forall IUT : (IUT \textbf{ ioco } S \implies \forall TC \in TS : \neg(TC \textit{ fails } IUT))$,
  i.e. only non-conformant IUT may be rejected by a $TC \in TS$.
- *Exhaustive* $\triangleq$
  $\forall IUT : (\neg(IUT \textbf{ ioco } S) \implies \exists TC \in TS : TC \textit{ fails } IUT)$,
  i.e. any non-conformant IUT may be rejected by a $TC \in TS$.
- 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 σ in $Can(S)$

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

Properties

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

Non-deterministic selection

Algorithm: partial unfolding of $Can(S)$

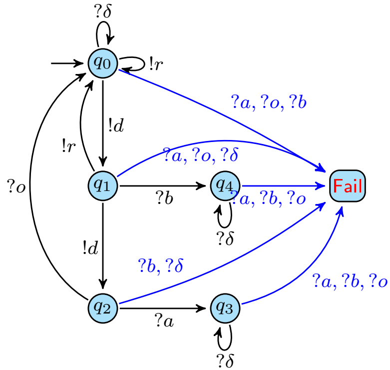
Start in q_0^c. After any trace σ in $Can(S)$

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

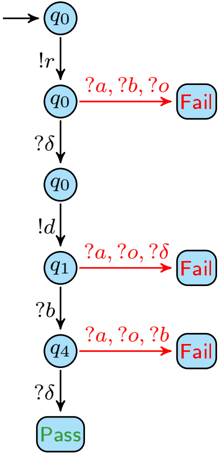
Properties

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...

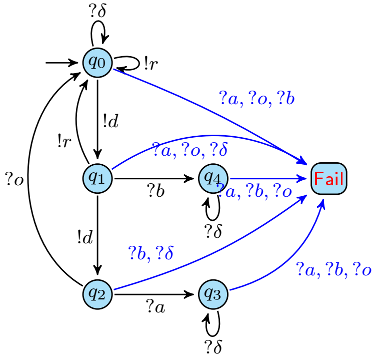
Main characteristics of Test Purpose Generation:

- Test selection by test purposes describing a set of behaviors to be tested, targeted by a test case

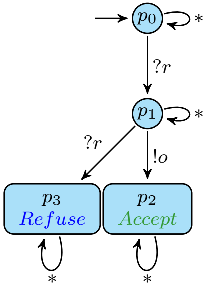
Test Purpose definition

Test Purpose

Deterministic and complete ioLTS $TP = (Q^{\mathsf{TP}}, A^{\mathsf{TP}}, \longrightarrow_{\mathsf{TP}}, q_0^{\mathsf{TP}})$ equipped with two sets $Accept^{\mathsf{TP}}$ and $Refuse^{\mathsf{TP}}$ of trap states

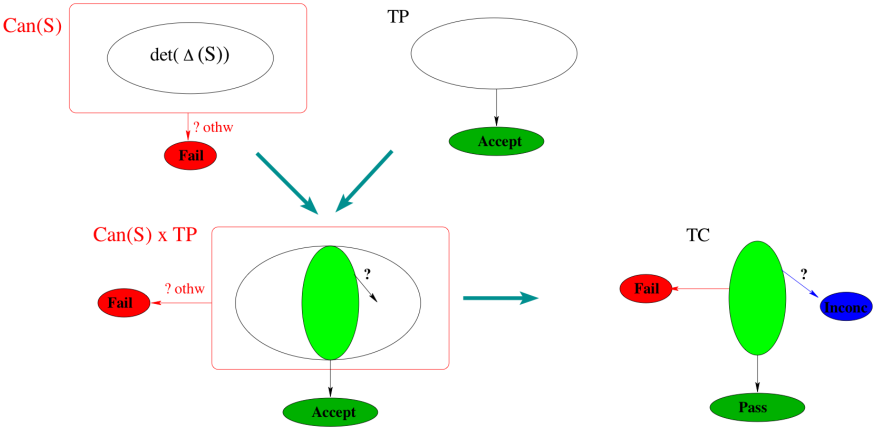


$Can(S)$



TP

Selection principle



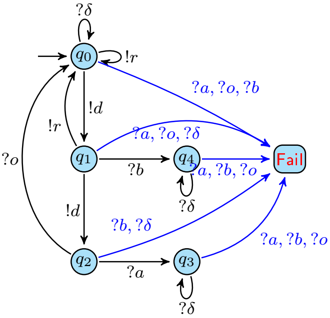
Synchronous Product : definition

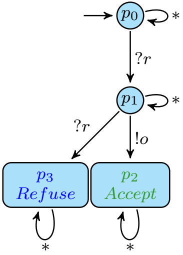
Definition of Synchronous Product

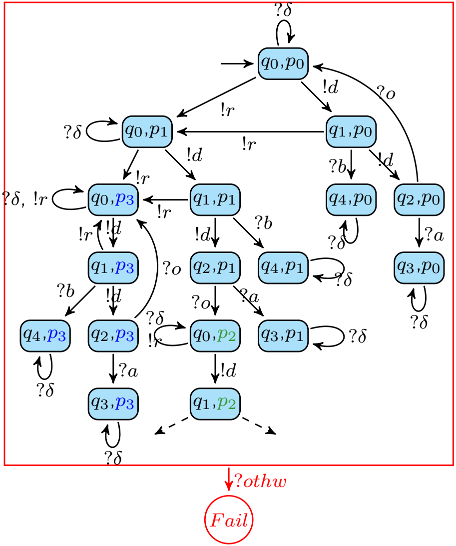
The Synchronous Product of two ioLTS $M_1 = (Q^{M1}, A, \longrightarrow_{M1}, q_0^{M1})$, and $M_2 = (Q^{M2}, A, \longrightarrow_{M2}, q_0^{M2})$ is the ioLTS $M_1 \times M_2 = (Q^{M1} \times Q^{M2}, A, \longrightarrow, q_0^{M1} \times q_0^{M2})$ where $\longrightarrow$ is defined by :

$$(q_{M1}, q_{M2}) \xrightarrow{a} (q'_{M1}, q'_{M2}) \Leftrightarrow (q_{M1} \xrightarrow{a}_{M1} q'_{M1}) \wedge (q_{M2} \xrightarrow{a}_{M2} q'_{M2})$$

The Synchronous Product $Can(S) \times TP$



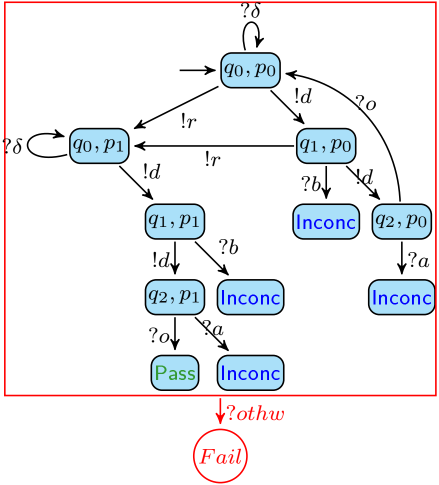




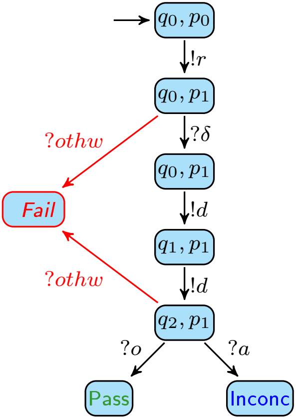
Complete Test Graph (CTG)

Co-reachability analysis :

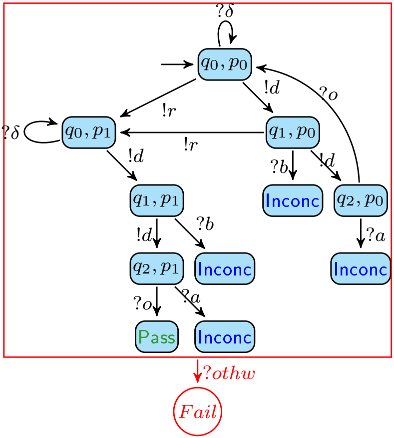
- Keep the first $Accept$ state in a path $\rightarrow$ *Pass*
- If $q \in coreach(Pass)$ keep q
- If $q \in \{Fail\}$ keep q
- If $q \notin coreach(Pass)$ input (tester point of view) is successor of a state $q' \in coreach(Pass)$ then $Inconc$



Ensuring controlabillity 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.
$\rightarrow$ 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)

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. "Dart: directed automated random testing", *SIGPLAN Not.*, 40(6):213–223, 2005.

[SMA05] K. Sen, D. Marinov, and G. Agha. "Cute: a concolic unit testing engine for c", In *Proceedings of the 10th European software engineering conference* , ESEC/FSE-13, pages 263–272. ACM, 2005.

[Got09] A. Gotlieb. "Euclide: A constraint-based testing platform for critical c programs", In *2th IEEE International Conference on Software Testing, Validation and Verification (ICST'09), Denver, CO*, page 10p, 04 2009.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided abstraction refinement", In E. Emerson and A. Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, Heidelberg, 2000.

[WMMR05] N .Williams, B. Marre, P. Mouy, and M. Roger. "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis", In *EDCC, volume 3463 of Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[BH08] S. Bardin and P. Herrmann. "Structural testing of executables." In *International Conference on Software Testing, Verification, and Validation, Los Alamitos, CA, USA*, pages 22–31. IEEE Computer Society, 2008.

Other References (MBT)

[LY96] D. Lee and M. Yannakakis."Principles and methods of testing finite state machines - a survey", *Proc. of the IEEE*, 84:1090–1123, 8 1996.

[Pet00b] A. Petrenko."Fault model-driven test derivation from finite state models: Annotated bibliography", In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes*, MOVEP '00, pages 196–205. Springer-Verlag, 2000.

[NT81] S. Naito and M. Tsunoyama. "Fault Detection for Sequential Machines by Transition-Tours", *Proceedings of Fault Tolerant Computer Systems*, pages 238–243, 1981.

[Gon70] G. Gonenc."A method for the design of fault detection experiment", *IEEE transactions on Computers*, C-19:551–558, 1970.

[SD88] K. Sabnani and A. Dahbura. "A protocol test generation procedure", *Computer Networks and ISDN Systems*, 15:285–297, 1988.

[Cho78] T.S. Chow. "Testing software design modeled by finite-state machines", *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[VCI89] S. Vuong, W. Chan, and M. Ito. "The UIOv-Method for Protocol Test Sequence Generation", In *2nd IWPTS International Workshop on Protocol Test Systems, Berlin*, 1989.

[FBK+91] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. "Test selection based on finite-state models", *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[CA92] W. Chung and P. Amer. "Improved on UIO Sequence Generation and Partial UIO Sequences", *In Protocol Specification, Testing, and Verification, XII*, Lake Buena Vista, Florida, USA. North-Holland, June 1992.

[BT00] E. Brinksma and J. Tretmans. "Testing Transition Systems: An Annotated Bibliography". *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, p. 44–50, Nantes, 2000.