Model Based Testing: principles and applications in the context of timed systems

Antoine Rollet

Université de Bordeaux - LaBRI (UMR CNRS 5800), France
rollet@labri.fr
http://www.labri.fr/~rollet
Outline

1. Model Based Testing
2. Conformance Testing with IOLTS
3. Testing Timed Systems
4. Conclusion and further work
Outline

1. Model Based Testing
2. Conformance Testing with IOLTS
3. Testing Timed Systems
4. Conclusion and further work
Introduction on testing

### Why testing?

- Systems getting more and more complex
  → potentially more bugs
- A failure may cost a lot (human and financial)
  → earlier detection implies weaker consequences

### Limitations

- “Testing can only be used to show the presence of bugs, but never to show their absence” (Dijkstra)
  → need to make some assumptions
  → Objective: increase the confidence in the system
Different kinds of testing

black box / white box

- **white box**: most elements of the system are known, especially source code (structural testing)
- **black box**: implementation is considered as an unknown black box; only interfaces are known
  → test generation based on the specification (functional testing)

What do we intend to test

User testing, performance testing, conformance testing, interoperability testing, robustness testing, etc...
Different kinds of testing

**black box / white box**

- **white box**: most elements of the system are known, especially source code (structural testing)
- **black box**: implementation is considered as an unknown black box; only interfaces are known
  → test generation based on the specification (functional testing)

**What do we intend to test**

User testing, performance testing, *conformance testing*, interoperability testing, robustness testing, etc...

→ Testing that a **black-box implementation** (IUT) of a system behaves correctly wrt. its functional specification Spec.
Conformance testing of reactive systems

Reactive system

System which **reacts** to its **environment** through its **interfaces**.

- **Environment**: human, software, hardware
- **Necessary to think about**:
  - **Controllability**: “how the tester can lead the test”
  - **Observability**: “how the tester can get information”

→ definition of **Points of Control and Observation (PCO)**.
→ definition of a **test architecture**
Model Based Testing

Industrial practice: manual design of test suites from informal specifications

⇒ possible automation for test generation, test execution, test evaluation (verdict)
⇒ Formal Methods
Test cases are generated from the **Model**

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 to define a **conformance relation** between the **Specification** and the **Implementation**
Model Based Testing (2)

- Test cases are generated from the Model
- 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 to define a conformance relation between the Specification and the Implementation

At the beginning...

Two main approaches of MBT:
- Finite State Machines
- Labeled Transition Systems
General schema

Property P

S ⊨ P ?

Specification S

I conf S ?

Implementation I
General schema

- Property P
- Specification S
- Implementation I

VERIFICATION

\[ S \models P \]

\[ I \text{ conf } S \]
General schema

- Property P
- Specification S
- Implementation I

\[ S \models P \]

\[ I \text{ conf } S \]

\[ \text{TEST} \]
General schema

- Property P
- Specification S
- Implementation I
- Test cases

Symbols:
- $S \models P$?
- I conf S ?
- ? observation
- ! control
- Verdict
General schema

- Property P
- Specification S
- Implementation I
- Test Generation
- Test cases
- Verdict

Diagram shows the relationship between property, specification, implementation, test generation, test cases, and verdict.
General schema

- Property $P$
- Specification $S$
- Implementation $I$
- Test Purpose
- Test Generation
- Test cases
- Verdict
Main ingredients of a testing theory

**Specification, implementation and conformance**

**Specification:** model of requested behaviors,

**Implementations:** model of *observable* real behavior (unknown)

**Conformance relation:** formalizes “IUT conforms to Spec”

**Tests cases and their executions**

**Test cases, test suites:** model of tests (control/observation)

**Test execution:** interaction test $\leftrightarrow$ IUT, produced *observations*, associated *verdicts* (e.g. pass, fail)

**Test suite properties:** “IUT passes TS” $\leftrightarrow$ “IUT conf S”

**Test generation**

**Algorithms:** $\text{tests} = \text{testgen} \left( \text{Spec} (+ \text{TestPurpose}) \right)$
Outline

1. Model Based Testing
2. Conformance Testing with IOLTS
3. Testing Timed Systems
4. Conclusion and further work
Part essentially based on:


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,
  - \( A^M_I \) input alphabet (with ?)
  - \( A^M_O \) output alphabet (with !)
  - \( I^M \) internal actions (\( \tau_k \))

\[ \longrightarrow_M \subseteq Q^M \times A^M \times Q^M \] 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:

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

**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 after σ**

\[ M \text{ after } \sigma \triangleq \{q_0\} \text{ after } \sigma \]
Non-determinism

$\begin{align*}
\text{Not to be confused with uncontrolled choice}
\end{align*}$

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

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

$Traces(M) = Traces(det(M))$
Non-determinism

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

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

\[ \text{Traces}(M) = \text{Traces}(det(M)) \]
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.

**Notation:**

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

In testing practice, one can observe traces of the \textit{IUT}, but also its \textbf{quiescences with timers}.

Only quiescences of \textit{IUT} unspecified in \textit{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:
  $\exists \tau_1, \ldots \tau_n : q \xrightarrow{\tau_1 \ldots \tau_n} q$. 
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 $\delta$

The suspension ioLTS of $M = (Q, A, \rightarrow, q_0)$ is an ioLTS $\Delta(M) = (Q, A \cup \{\delta\}, \rightarrow_{\Delta(M)}, q_0)$ where $\rightarrow_{\Delta(M)} = \rightarrow \cup \{q \rightarrow \delta q \mid q \in \text{quiescent}(M)\}$. 
Suspension traces

\[ \Delta(S) \]

\[ \text{det}(\Delta(S)) \]

\[ STraces(M) \triangleq \text{Traces}(\Delta(M)) = \text{Traces}(\text{det}(\Delta(M))) \]

\( STraces(S) \) and \( STraces(I) \) represent visible behaviors of \( S \) and \( I \) for testing \( \Rightarrow \) a base for the definition of conformance.
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 $S$.

**Conformance**

$IUT \ ioco \ S \ \triangleq \ \forall \sigma \in STraces(S), \ Out(\Delta(IUT) \ after \ \sigma) \subseteq Out(\Delta(S) \ after \ \sigma)$

with $Out(P) \ \triangleq \ \Gamma(P) \cap A^\delta_O$ \quad $^a$: set of outputs $\wedge$ quiescences in $P$.

$^a A^\delta_O$ 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
From $S$ (more precisely from $\text{det}(\Delta(S)) = (Q^d, A^d, \rightarrow_d, q_0^d)$), build an ioLTS $Can(S) = (Q^c, A^c, \rightarrow_c, q_0^c) \rightarrow$ the most general ioLTS permitting to detect 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 $\text{Can}(S') = (Q^c, A^c, \rightarrow_c, q^c_0) \rightarrow$ the most general ioLTS permitting to detect non-conformance of implementation $IUT$.

- $Q^c = Q^d \cup \{\text{Fail}\}$ and $q^c_0 = q^d_0$
- $A^c = A^c_I \cup A^c_O$ where $A^c_I = A^d_O$ and $A^c_O = A^d_I$ input / output inversion
- $\rightarrow_c = \rightarrow_d \cup \{q \xrightarrow{a} \text{Fail} \mid q \in Q^d, a \in A^c_I \land \neg(q \xrightarrow{a} d)\}$, 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 $\text{Can}(S) = (Q^c, A^c, \rightarrow_c, q_0^c) \rightarrow$ the most general ioLTS permitting to detect 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 permitting to detect 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 $\text{Can}(S) = (Q^c, A^c, \rightarrow_c, q_0^c) \rightarrow$ the most general ioLTS permitting to detect non-conformance of implementation $IUT$.

$IUT \ioco S \iff \text{St} \text{Traces}(IUT) \cap \text{Traces}_{\text{Fail}}(\text{Can}(S)) = \emptyset$
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.

- \(A^{TC}_O = A^S_I\) and \(A^{TC}_I = A^S_O \cup \{\delta\}\) (**input / output inversion**)
- \(TC\) is **controllable**, i.e. never have to choose btw. several outputs or btw. inputs and outputs:
  \(\forall q \in Q^{TC}, (\exists a \in A^{TC}_O, q \xrightarrow{a}^{TC} \Rightarrow \forall b \in A^{TC}, (b \neq a \Rightarrow q \nrightarrow^{TC} b))\)
- All states permitting an input, are **input-complete**, except verdict states.
Test execution

Modelled by the parallel composition $TC \parallel \Delta(IUT)$ synchronizing on common visible actions

$TC1 \parallel \Delta(IUT)$
Properties of test suites

**TC** fails **IUT** iff an execution of **TC**∥Δ(**IUT**) reaches **Fail**

Expresses a *possibility* for rejection.

Due to non-controllable choices of **IUT**, a single test case applied on a single Implementation can produce all different verdicts!

**Soundness, Exhaustiveness, Completeness**

A set of test cases **TS** is

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

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

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

A test case $TC$ fails an implementation $IUT$ iff an execution of $TC \parallel \Delta(IUT)$ reaches Fail.

Expresses a possibility for rejection.

Due to non-controllable choices of $IUT$, a single test case applied on a single implementation can produce all different verdicts!

**Soundness, Exhaustiveness, Completeness**

A set of test cases $TS$ is

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

- **Exhaustive** $\triangleq \forall IUT : (\neg (IUT \text{ ioco } S) \implies \exists TC \in TS : TC \text{ fails } IUT)$,
  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 (à la TorX)
2. Selection guided by a test purpose (à la TGV)
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**

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

$TS$ 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

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

$T_S$ is sound and limit-exhaustive
Examples

\[ Can(S) \]

TC1
Examples

Can(S)

TC2
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,
- off-line selection, a posteriori execution.
Test Purpose definition

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

$A^{TP} = A^{S_{VIS}} \cup \{\delta\}$
Selection principle
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:

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

- Keep the first *Accept* state in a path
- 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) 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.
Conclusion

- Testing theory for ioLTS

- Test generation for \texttt{finite} ioLTS

  - Non-deterministic selection: unfolding of $\text{Can}(S)$

  - Selection by test purpose: for finite ioLTS based on co-reachability analysis.

- Soundness and exhaustiveness.
Outline

1. Model Based Testing
2. Conformance Testing with IOLTS
3. Testing Timed Systems
4. Conclusion and further work
Part essentially based on:


Main lines

- Need a “new” model to describe real-time aspects: *Timed Automata with Inputs and Outputs*... and semantics.

- Need a “new” conformance relation: *rtioco*

- Non-deterministic online test generation

- Discussion about offline test generation
“Uppaal-like” approach

Explicit and separate model of the environment

Input

Real Env. [input]

IUT

Output

\[\mathcal{E}\] [input]

\[\mathcal{S}\] [output]

- test generation tool can synthesize only relevant scenario
- designer can lead the test to specific situations
Timed Automaton

Semantics defined in terms of **TIOTS**.
Possibly **non-deterministic**
Timed Input Output Transition System (TIOTS)

Given a set of actions $A$, divided in $A_{out}$ and $A_{in}$, and $\tau \not\in A$. 

$$(A_{\tau} \triangleq A \cup \{\tau\})$$

if no precision is given, in the following $a_{[k]}$ is an action, $d_{[k]}$ is a delay

**TIOTS definition**

$$S = (S, s_0, A_{in}, A_{out}, \rightarrow)$$  
where:

- $S$ set of states, $s_0 \in S$ the *initial state*
- $\rightarrow \subseteq S \times (A_{\tau} \cup \mathbb{R}_{\geq 0}) \times S$ transition relation with
  - time determinism : $(s \xrightarrow{d} s' \land s \xrightarrow{d} s'') \Rightarrow s' = s''$
  - time additivity : $(s \xrightarrow{d_1} s' \land s' \xrightarrow{d_2} s'') \Rightarrow s \xrightarrow{d_1+d_2} s''$
  - zero-delay : $\forall s, s \xrightarrow{0} s$

Testing point of view : Timed Traces are considered, e.g.

$$\sigma = ?coin \cdot 1 \cdot ?req \cdot 2 \cdot \!wCoffee \cdot 9 \cdot ?coin$$
Notations / Definitions

- \( s \xrightarrow{a} s' \) iff \( s \xrightarrow{\tau} * \xrightarrow{a} \xrightarrow{\tau} * \xrightarrow{\tau} s' \)
- \( s \xrightarrow{d} s' \) iff \( s \xrightarrow{\tau} * \xrightarrow{d_1} \xrightarrow{\tau} * \xrightarrow{d_2} \xrightarrow{\tau} * \ldots \xrightarrow{\tau} * \xrightarrow{d_n} \xrightarrow{\tau} * \xrightarrow{\tau} s' \)
  where \( d = \sum_{k=1}^{n} d_k \)
- usually generalized to sequences

Observable Timed Traces \( TTr(s) \)

\[
TTr(s) = \{ \sigma \in (A \cup \mathbb{R}_{\geq 0})^* | s \xrightarrow{\sigma} \}
\]

Example: \( \sigma = ?\text{coin} \cdot 1 \cdot ?\text{req} \cdot 2 \cdot !\text{wCoffee} \cdot 9 \cdot ?\text{coin} \)

After

\( s \text{ After } \sigma = \{ s' | s \xrightarrow{\sigma} s' \} \), \( S' \text{ After } \sigma = \bigcup_{s \in S'} s \text{ After } \sigma \)

Out

\[
Out(s) = \{ a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \xrightarrow{a} \}
\]

\( Out(S') = \bigcup_{s \in S'} Out(s) \)
Notations / Definitions

- $s \xrightarrow{a} s'$ iff $s \xrightarrow{\tau} * \xrightarrow{a} \xrightarrow{\tau} * s'$
- $s \xrightarrow{d} s'$ iff $s \xrightarrow{\tau} * \xrightarrow{d_1} \xrightarrow{\tau} * \xrightarrow{d_2} \xrightarrow{\tau} * \ldots \xrightarrow{d_n} \xrightarrow{\tau} * s'$

where $d = \sum_{k=1}^{n} d_k$

- usually generalized to sequences

Observable Timed Traces $TTr(s)$

$TTr(s) = \{\sigma \in (A \cup \mathbb{R}_{\geq 0})^* | s \xrightarrow{\sigma}\}$

Example: $\sigma = ?coin \cdot 1 \cdot ?req \cdot 2 \cdot !wCoffee \cdot 9 \cdot ?coin$

After

$s$ After $\sigma = \{s'|s \xrightarrow{\sigma} s'\}$, $S'$ After $\sigma = \bigcup_{s \in S'} s$ After $\sigma$

Out

$Out(s) = \{a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \xrightarrow{a}\}$

$Out(S') = \bigcup_{s \in S'} Out(s)$
Notations / Definitions

- \( s \xrightarrow{a} s' \iff s \xrightarrow{\tau} s' \)
- \( s \xrightarrow{d} s' \iff s \xrightarrow{\tau} s' \)
  where \( d = \sum_{k=1}^{n} d_k \)
  usually generalized to sequences

Observable Timed Traces \( TTr(s) \)

\[ TTr(s) = \{ \sigma \in (A \cup \mathbb{R}_{\geq 0})^* | s \xrightarrow{\sigma} \} \]
Example : \( \sigma = ?coin \cdot 1 \cdot ?req \cdot 2 \cdot !wCoffee \cdot 9 \cdot ?coin \)

After

\[ s \text{ After } \sigma = \{ s' | s \xrightarrow{\sigma} s' \}, \quad S' \text{ After } \sigma = \bigcup_{s \in S'} s \text{ After } \sigma \]

Out

\[ Out(s) = \{ a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \xrightarrow{a} \} \]
\[ Out(S') = \bigcup_{s \in S'} Out(s) \]
Notations / Definitions

- \( s \xrightarrow{a} s' \) iff \( s \xrightarrow{\tau}^* \xrightarrow{a} \xrightarrow{\tau}^* s' \)
- \( s \xrightarrow{d} s' \) iff \( s \xrightarrow{\tau}^* \xrightarrow{d_1} \xrightarrow{\tau}^* \xrightarrow{d_2} \xrightarrow{\tau}^* \cdots \xrightarrow{\tau}^* \xrightarrow{d_n} \xrightarrow{\tau}^* s' \)
  where \( d = \sum_{k=1}^{n} d_k \)
- Usually generalized to sequences

Observable Timed Traces \( TTr(s) \)

\[ TTr(s) = \{ \sigma \in (A \cup \mathbb{R}_{\geq 0})^* | s \xrightarrow{\sigma} \} \]

Example: \( \sigma = ?coin \cdot 1 \cdot ?req \cdot 2 \cdot !wCoffee \cdot 9 \cdot ?coin \)

After

\( s \text{ After } \sigma = \{ s' | s \xrightarrow{\sigma} s' \}, \quad S' \text{ After } \sigma = \bigcup_{s \in S'} s \text{ After } \sigma \)

Out

\[ Out(s) = \{ a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \xrightarrow{a} \} \quad Out(S') = \bigcup_{s \in S'} Out(s) \]
Timed Automata (with Inputs and Outputs) : definition

Given $X$ set of clock variables, $G(X)$ set of guards, $U(X)$ set of updates.

**Timed Automaton**

$TA = (L, l_0, I, E)$ where
- $L$ set of locations, $l_0$ initial location
- $I : L \rightarrow G(X)$ assigns invariants to locations
- $E \subseteq L \times G(X) \times A_{\tau} \times U(X) \times L$ set of edges (written $l \xrightarrow{g, \alpha, u} l'$)

Observable trace example : $\sigma = ?coin \cdot 6 \cdot ?req \cdot 3 \ldots$

$Out(?coin \cdot 6 \cdot ?req \cdot 3) = \{sCoffee\} \cup [0, 2]$
Semantics of Timed Automata

Semantics as a TIOTS defined by:

- **States** of the form \( s = (l, \bar{v}) \), s.t.
  - \( l \) is a location
  - \( \bar{v} \in \mathbb{R}^X_{\geq 0} \) clock valuation satisfying invariant of \( l \)

- **Delay transitions**

\[
\forall d' \leq d.I_l(d') \\
(l, \bar{v}) \xrightarrow{d} (l, \bar{v} + d)
\]

- **Discrete transitions**

\[
l ^{g,\alpha,u} \xrightarrow{} l' \land g(\bar{v}) \land I_v(\bar{v}'), \bar{v}' = u(\bar{v}) \\
(l, \bar{v}) \xrightarrow{\alpha} (l', \bar{v}')
\]

most reasoning done on the semantics
Relativized timed conformance

- \( S = (S^S, s_0^S, A_{in}, A_{out}, \rightarrow S) \) a weakly input enabled (i.e. \( \forall s \in S^S, \forall i \in A_{in}, s \overset{i}{\Rightarrow} \)) TIOTS

- \( IUT = (S^{IUT}, s_{0}^{IUT}, A_{in}, A_{out}, \rightarrow_{IUT}) \) a weakly input enabled TIOTS

- \( E = (E^E, e_{0}^E, A_{out}, A_{in}, \rightarrow E) \) (input / output inversion) weakly input enabled TIOTS.

\texttt{rtioco}_{e}

Let \( s \in S^S, e \in E^E \) and \( iut \in S^{IUT} \):

\[ iut \text{ rtioco}_e s \]

iff

\[ \forall \sigma \in TTr(e), \text{Out}((iut, e) \text{ After } \sigma) \subseteq \text{Out}((s, e) \text{ After } \sigma) \]

iff

\[ TTr(iut) \cap TTr(e) \subseteq TTr(s) \cap TTr(e) \]
rtioco ensures Implementation has only the behavior allowed by Specification:

- Implementation not allowed to produce an output at a time when not allowed by Specification

- Implementation not allowed to omit producing an output when required by the Specification
rtiloco examples

Environment

Specification $s$

Implementation $i_1$

<table>
<thead>
<tr>
<th>Trace $\sigma$</th>
<th>$Out(s \text{ After } \sigma)$</th>
<th>$Out(i_1 \text{ After } \sigma)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \cdot 2$</td>
<td>$\mathbb{R}_{\geq 0}$</td>
<td>$\mathbb{R}_{\geq 0}$</td>
</tr>
<tr>
<td>$c \cdot 4 \cdot r \cdot 1$</td>
<td>${wCoffee, sCoffee} \cup [0, 4]$</td>
<td>$[0, 1]$</td>
</tr>
<tr>
<td>$c \cdot 4 \cdot r \cdot 2$</td>
<td>${wCoffee, sCoffee} \cup [0, 3]$</td>
<td>${wCoffee, 0}$</td>
</tr>
<tr>
<td>$c \cdot 5 \cdot r \cdot 3$</td>
<td>${sCoffee} \cup [0, 2]$</td>
<td>${sCoffee, 0}$</td>
</tr>
<tr>
<td>$c \cdot 5 \cdot r \cdot 5$</td>
<td>${sCoffee, 0}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
**rtioco examples (2)**

Environment

```
<table>
<thead>
<tr>
<th>Trace σ</th>
<th>Out(s After σ)</th>
<th>Out(i2 After σ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>c · 2</td>
<td>R ≥ 0</td>
<td>[0, 2]</td>
</tr>
<tr>
<td>c · 4 · r · 1</td>
<td>{wCoffee, sCoffee} ∪ [0, 4]</td>
<td>{wCoffee} ∪ [0, 1]</td>
</tr>
<tr>
<td>c · 4 · r · 2</td>
<td>{wCoffee, sCoffee} ∪ [0, 3]</td>
<td></td>
</tr>
<tr>
<td>c · 5 · r · 3</td>
<td>{sCoffee} ∪ [0, 2]</td>
<td>[0, 4]</td>
</tr>
<tr>
<td>c · 5 · r · 5</td>
<td>{sCoffee, 0}</td>
<td>[0, 2]</td>
</tr>
</tbody>
</table>
```

Specification s

Implementation i2
Online testing (à la TorX)

- On-the-fly testing: combines test generation and execution
- Non-deterministic generation
- Symbolic states
- Weakly input-enabled and non-blocking TIOTS

Advantages:
- reduces state space explosion
- handles non-determinism

Drawbacks:
- specification must be analyzed online, in real-time
- test runs may be long...
- coverage criteria can not be guaranteed
Non-determinism

Often used:
- as means of abstraction
- to model optional behavior, permitted but not required

**Determinism definition**

An TIOTS \( (S') \) is **deterministic** if
\[
\forall \alpha \in (A_T \cup \mathbb{R}_{\geq 0}), \forall s \in S, (s \overset{\alpha}{\rightarrow} s' \land s \overset{\alpha}{\rightarrow} s'') \Rightarrow s' = s''.
\]

\[(l_0, x = 3) \text{ After } a = \{(l_2, x = 3), (l_4, x = 3), (l_3, x = 0)\}\]
\[(l_5, x = 0) \text{ After } 4 = \{(l_5, x = 4), (l_6, 0 \leq x \leq 4)\}\]
Uppaal TRON algorithm $TestGenExe(S, E, IUT, T)$

```
while $\mathcal{Z} \neq \emptyset \land \#\text{iterations} \leq T$ do
    switch randomly choose btw action, delay and restart do
        case action /* offer an input */
            if $EnvOutput(\mathcal{Z}) \neq \emptyset$ then
                randomly choose $i \in EnvOutput(\mathcal{Z})$; send $i$ to $IUT$;
                $\mathcal{Z} := \mathcal{Z} \text{ After } i$;
            
        case delay /* wait for an output */
            randomly choose $d \in Delays(\mathcal{Z})$;
            sleep for $d$ time units or wake up on output $o$ at $d^{' \leq d}$;
            if $o$ occurs then
                $\mathcal{Z} := \mathcal{Z} \text{ After } d^{'}$;
                if $o \notin ImpOutput(\mathcal{Z})$ then return $FAIL$ else
                    $\mathcal{Z} := \mathcal{Z} \text{ After } o$
            else
                $\mathcal{Z} := \mathcal{Z} \text{ After } d$
            
        case restart $\mathcal{Z} := \{(s_0, e_0)\}$, reset $IUT$ /* reset and restart */
    
if $\mathcal{Z} = \emptyset$ then return $FAIL$ else return $PASS$
```
Example of test execution

Symbolic state set:
\[ \{(k_0l_0, x = 0)\} \]

EnvOutput: coin

ImpOutput: \(\emptyset\)
Example of test execution

Symbolic state set:
\{(k_0 l_0, x = 0)\}

EnvOutput: coin

ImpOutput: \emptyset

Wait for output (delay) or offer input?
Example of test execution

Symbolic state set: 
\{(k_0l_0, x = 0)\}
EnvOutput: coin
ImpOutput: ∅
Example of test execution

Symbolic state set:
\{ (k_1 l_1, x = 0) \}
EnvOutput: req
ImpOutput: \emptyset

Update the state set and other variables
Example of test execution

Symbolic state set: \( \{(k_1l_1, x = 0)\} \)

EnvOutput: \( req \)

ImpOutput: \( \emptyset \)
Example of test execution

Symbolic state set: \{(k_1l_1, x = 5)\}
EnvOutput: req
ImpOutput: ∅
Example of test execution

Symbolic state set:
\[ \{(k_1l_1, x = 5)\} \]
EnvOutput: \( req \)
ImpOutput: \( \emptyset \)
### Example of test execution

**Tester**

```
Symbolic state set:
{(k₂l₂, x = 0), (k₂l₃, x = 0)}

EnvOutput: ∅
ImpOutput: ∅
```

**Implementation**

Update the state set and other variables.
Example of test execution

Symbolic state set:
\{ (k_2l_2, x = 0), (k_2l_3, x = 0) \}
EnvOutput: ∅
ImpOutput: ∅

Wait or offer input? Let’s wait for 4 units
Example of test execution

Symbolic state set: \((k_2l_3, x = 4)\)
EnvOutput: 0
ImpOutput: \(\{sCoffee\}\)

... no output so far: update the state set
Example of test execution

Symbolic state set:
\[ \{ (k_2 l_3, x = 4) \} \]
EnvOutput: \( \emptyset \)
ImpOutput: \( \{ sCoffee \} \)

Wait or offer input? Let’s wait for 2 units
Example of test execution

**Tester**

```
Symbolic state set: 
{(k2l3, x = 4)}
EnvOutput:  ∅
ImpOutput: {sCoffee}
```

**Implementation**

```
Got output after 0 delay: update the state set
```

A. Rollet - ETR2011 - Brest (France) - August 2011
Example of test execution

Symbolic state set:
\{ (k_2l_3, x = 4) \}

EnvOutput: ∅

ImpOutput: \{ sCoffee \}

What if there is a bug? Let’s wait back for 2 units
Example of test execution

**Tester**

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_0$</td>
<td>$k_1$</td>
<td><img src="example-diagram.png" alt="Diagram" /></td>
</tr>
<tr>
<td>$k_1$</td>
<td>$l_0$</td>
<td>$x \geq 1$</td>
</tr>
<tr>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$x \geq 3$</td>
</tr>
<tr>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$x \leq 5$</td>
</tr>
<tr>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$x \geq 3$</td>
</tr>
<tr>
<td>$l_3$</td>
<td>$x \geq 1$</td>
<td>$x \geq 3$</td>
</tr>
</tbody>
</table>

**Symbolic state set:**

\[ \{(k_2l_3, x = 4)\} \]

**EnvOutput:** $\emptyset$

**ImpOutput:** $\{sCoffee\}$

---

**Implementation**

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_0$</td>
<td>$l_1$</td>
<td><img src="example-diagram.png" alt="Diagram" /></td>
</tr>
<tr>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$x \leq 4$</td>
</tr>
<tr>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$x &gt; 4$</td>
</tr>
<tr>
<td>$l_3$</td>
<td>$x \geq 1$</td>
<td>$x \geq 3$</td>
</tr>
</tbody>
</table>

**output after 0 delay:** $wCoffee \notin \{sCoffee\}$
Properties of test cases

Let a $S$, $E$, and $IUT$ three weakly input enabled TIOTS, with $IUT$ deterministic.

### Soundness, Exhaustiveness

- **Soundness**:  
  \[ \text{TestGenExe}(S, E, IUT, T) = \text{Fail} \Rightarrow \neg (IUT \ rtioco_{E} S) \]

- **Exhaustiveness**:  
  \[ \neg (IUT \ rtioco_{E} S) \Rightarrow \text{Prob}(\text{TestGenExe}(S, E, IUT, T) = \text{Fail}) \xrightarrow{T \to \infty} 1 \]

If $IUT$ is not deterministic, exhaustiveness is not guaranteed.
Offline test generation: main ideas

- **Advantages**:
  - Test cases are easier and faster to execute
  - Possibility to guarantee a coverage or a test objective

- **Drawbacks**:
  - Specification has to be analyzed entirely \(\Rightarrow\) state explosion
  - Only deterministic (and impossible to determinize in general case)

---

Test Generation with Test Purpose

- Synchronous Product btw Spec. and T.P. \(\rightarrow\) need a finite symbolic representation of TA (Region Graph, Zones, ...)
- Test Case Generation with Uppaal
- Test Case Generation using Observers

Still immature...
Offline test generation: main ideas

- **Advantages:**
  - test cases are easier and faster to execute
  - possibility to guarantee a coverage or a test objective

- **Drawbacks:**
  - specification has to be analyzed entirely \(\Rightarrow\) state explosion
  - only deterministic (and impossible to determinize in general case)

**Test Generation with Test Purpose**

- Synchronous Product btw Spec. and T.P. \(\rightarrow\) need a finite symbolic representation of TA (Region Graph, Zones, ...)
- Test Case Generation with Uppaal
- Test Case Generation using Observers

Still immature...
Offline test generation: main ideas

- **Advantages:**
  - Test cases are easier and faster to execute
  - Possibility to guarantee a coverage or a test objective

- **Drawbacks:**
  - Specification has to be analyzed entirely ⇒ state explosion
  - Only deterministic (and impossible to determinize in general case)

### Test Generation with Test Purpose

- Synchronous product between Spec. and T.P. → need a finite symbolic representation of TA (Region Graph, Zones, ...)
  - Test Case Generation with Uppaal
  - Test Case Generation using Observers

Still immature...
Offline test generation: main ideas

- **Advantages**:  
  - Test cases are easier and faster to execute  
  - Possibility to guarantee a coverage or a test objective  

- **Drawbacks**:  
  - Specification has to be analyzed entirely \(\Rightarrow\) state explosion  
  - Only deterministic (and impossible to determinize in general case)

Test Generation with Test Purpose

- Synchronous Product btw Spec. and T.P. \(\rightarrow\) need a finite symbolic representation of TA (Region Graph, Zones, ...)
- Test Case Generation with Uppaal
- Test Case Generation using Observers

Still immature...
Offline test generation: main ideas

- **Advantages:**
  - Test cases are easier and faster to execute
  - Possibility to guarantee a coverage or a test objective

- **Drawbacks:**
  - Specification has to be analyzed entirely ⇒ state explosion
  - Only deterministic (and impossible to determinize in general case)

**Test Generation with Test Purpose**

- Synchronous Product btw Spec. and T.P. → need a finite symbolic representation of TA (Region Graph, Zones, ...)
- Test Case Generation with Uppaal
- Test Case Generation using Observers

Still immature...
Offline test generation: main ideas

- **Advantages**:  
  - test cases are easier and faster to execute  
  - possibility to guarantee a coverage or a test objective

- **Drawbacks**:  
  - specification has to be analyzed entirely $\Rightarrow$ state explosion  
  - only deterministic (and impossible to determinize in general case)

---

**Test Generation with Test Purpose**

- Test Case Generation with Uppaal

**Still immature...**
Test Case generation with Test Purpose using Uppaal

**Uppaal Tool:**
- Model checker for temporal properties
- Symbolic efficient analysis (using DBM)
- Generates diagnostic traces (shortest or fastest)

**Assumptions:** TIOTS are deterministic, weakly input enabled and output urgent

**Idea**
- Formulate the problem as safety property (usually solved by a reachability analysis) → obtain a trace of the form $(s_0, e_0) \xrightarrow{\gamma_0} (s_1, e_1) \ldots \xrightarrow{\gamma_{n-1}} (s_n, e_n)$
- Obtain a test sequence by projecting the trace to the $E$ - component (and summing delays)
- Add Verdicts to the test sequence to obtain a test case

*Test sequences are guaranteed to be included in the specification.*
Test Case generation with Test Purpose using Uppaal

Uppaal Tool:
- Model checker for temporal properties
- Symbolic efficient analysis (using DBM)
- Generates diagnostic traces (shortest or fastest)

Assumptions: TIOTS are deterministic, weakly input enabled and output urgent

Idea

- Formulate the problem as safety property (usually solved by a reachability analysis) → obtain a trace of the form $(s_0, e_0) \xrightarrow{\gamma_0} (s_1, e_1) \ldots \xrightarrow{\gamma_{n-1}} (s_n, e_n)$
- Obtain a test sequence by projecting the trace to the $E$ component (and summing delays)
- Add Verdicts to the test sequence to obtain a test case

Test sequences are guaranteed to be included in the specification
Example of test case

Sequence:

\( \! in_0 \cdot delay \cdot ?out_0 \)
Examples of Test Purposes (light controller)

**TP1**: Check that the light can become bright:
Simple reachability property: *eventually* the system specification can enter location **BRIGHT**

**TP2**: Check the light switch off after 3 successive touches
reachability property + specific environment:

- ![Diagram](https://via.placeholder.com/150)
Examples of Test Purposes (light controller)

**TP1**: Check that the light can become bright:

*Simple reachability property*: eventually the system specification can enter location **BRIGHT**

**TP2**: Check the light switch off after 3 successive touches

*Reachability property + specific environment*:

```
!touch
z := 0
```

```
z ≥ Treact
!touch
z := 0
```

```
z ≥ Treact
!touch
z := 0
```

```
?off
```

```
?off
?dim
?bright
```

```
?off
?dim
?bright
```

```
?dim
?bright
```

```
?off
```

```
?off
```

```
?off
```

```
goal
```

Examples of Test Purposes (light controller)

**TP1**: Check that the light can become bright:

*Simple reachability property*: eventually the system specification can enter location **BRIGHT**

**TP2**: Check the light switch off after 3 successive touches

*reachability property + specific environment*:
Examples of Test Purposes (light controller)

**TP1** : Check that the light can become bright:
Simple reachability property: **eventually** the system specification can enter location **BRIGHT**

**TP2** : Check the light switch off after 3 successive touches
reachability property + specific environment:

![Diagram](image-url)
Examples of coverage criteria

**Edge Coverage**

Reachability property :
- add a boolean variable $e_i$ for each edge to be covered, initially $false$
- add assignment $e_i := true$ for each edge to be covered
- property to reach : $\wedge e_i == true$

**Location ($l_i$) Coverage**

- add a boolean variable $b_i$ for each node, initially $false$ (except initial)
- for every edge $l' \xrightarrow{g,a,u} l_i$ add assignment $b_i := true$
- property to reach : $\wedge b_i == true$

Etc... but not always possible
Examples of coverage criteria

Edge Coverage

Reachability property :

- add a boolean variable $e_i$ for each edge to be covered, initially $false$
- add assignment $e_i := true$ for each edge to be covered
- property to reach : $\land e_i == true$

Location ($l_i$) Coverage

- add a boolean variable $b_i$ for each node, initially $false$ (except initial)
- for every edge $l' \xrightarrow{g,a,u} l_i$ add assignment $b_i := true$
- property to reach : $\land b_i == true$

Etc... but not always possible
Examples of coverage criteria

**Edge Coverage**

**Reachability property:**
- add a boolean variable $e_i$ for each edge to be covered, initially $false$
- add assignment $e_i := true$ for each edge to be covered
- property to reach: $\bigwedge e_i == true$

**Location ($l_i$) Coverage**
- add a boolean variable $b_i$ for each node, initially $false$ (except initial)
- for every edge $l' \xrightarrow{g,a,u} l_i$ add assignment $b_i := true$
- property to reach: $\bigwedge b_i == true$

Etc... but not always possible
Examples of coverage criteria

Edge Coverage
Reachability property:
- add a boolean variable $e_i$ for each edge to be covered, initially $false$
- add assignment $e_i := true$ for each edge to be covered
- property to reach: $\bigwedge e_i == true$

Location ($l_i$) Coverage
- add a boolean variable $b_i$ for each node, initially $false$ (except initial)
- for every edge $l' \xrightarrow{g,a,u} l_i$ add assignment $b_i := true$
- property to reach: $\bigwedge b_i == true$

Etc... but not always possible
Using observers

Weakness of this offline approach:

- time-consuming to find the proper model annotation
- model-checking tools not adapted for test cases generation: may lead to performance problems

→ Possibility to use a language of observers to describe coverage criteria
→ Adaptation of model-checking algorithms for test generation based on observers
Outline

1. Model Based Testing
2. Conformance Testing with IOLTS
3. Testing Timed Systems
4. Conclusion and further work
Conclusion

- Testing theory and generation algorithms for finite ioLTS
- Extensions for Timed Automata with Inputs and Outputs
- Off-line and on-line algorithms

Perspectives

- Mature tools (scaling)
- “Real-time” coverage criteria
- Testing seen as “Game theory”
- Add variables with “complex” assignments
- Run-time verification / enforcement dans le cadre temporisé
Conclusion

- Testing theory and generation algorithms for finite ioLTS
- Extensions for Timed Automata with Inputs and Outputs
- Off-line and on-line algorithms

Perspectives

- Mature tools (scaling)
- “Real-time” coverage criteria
- Testing seen as “Game theory”
- Add variables with “complex” assignments
- Run-time verification / enforcement dans le cadre temporisé
Thank you for your attention

rollet@labri.fr