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

Outline

Introduction on testing

Why testing?

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

Limitations

- "Testing can only be used to show the presence of bugs, but never to show their absence" (Dijkstra)
  $\rightarrow$ need to make some assumptions
  $\rightarrow$ 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
  $\rightarrow$ 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
  $\rightarrow$ 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...

$\rightarrow$ 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).

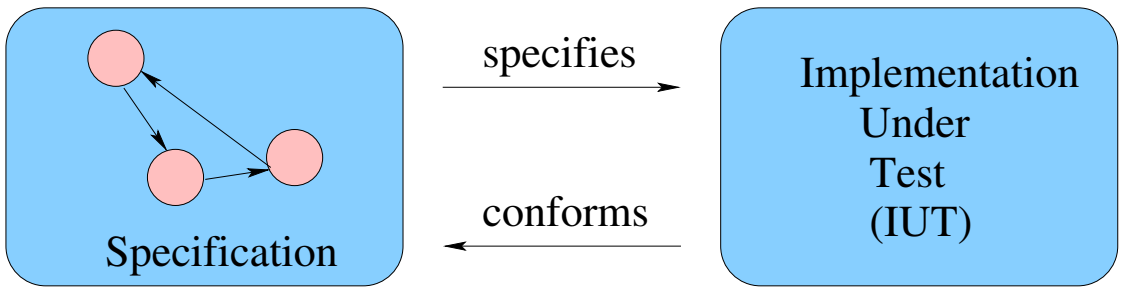
→ definition of a test architecture

Model Based Testing

Industrial practice: manual design of test suites from informal specifications

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

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

Model Based Testing (2)

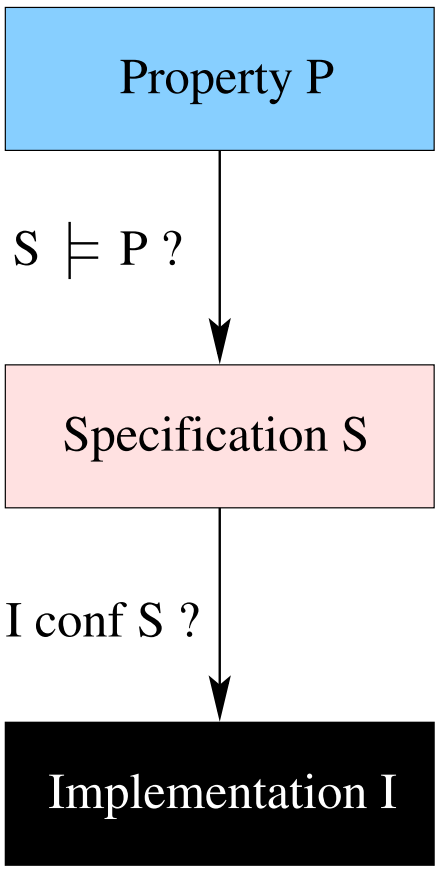
- Test cases are generated from the Model
- Problems :
  - need to find a "good" model of the specification
  - what does specify mean?
  - what does conform mean?
- Implementation is supposed to be equivalent to a formal model (but Implementation is unknown)
- Need to define a conformance relation between the Specification and the Implementation

At the beginning...

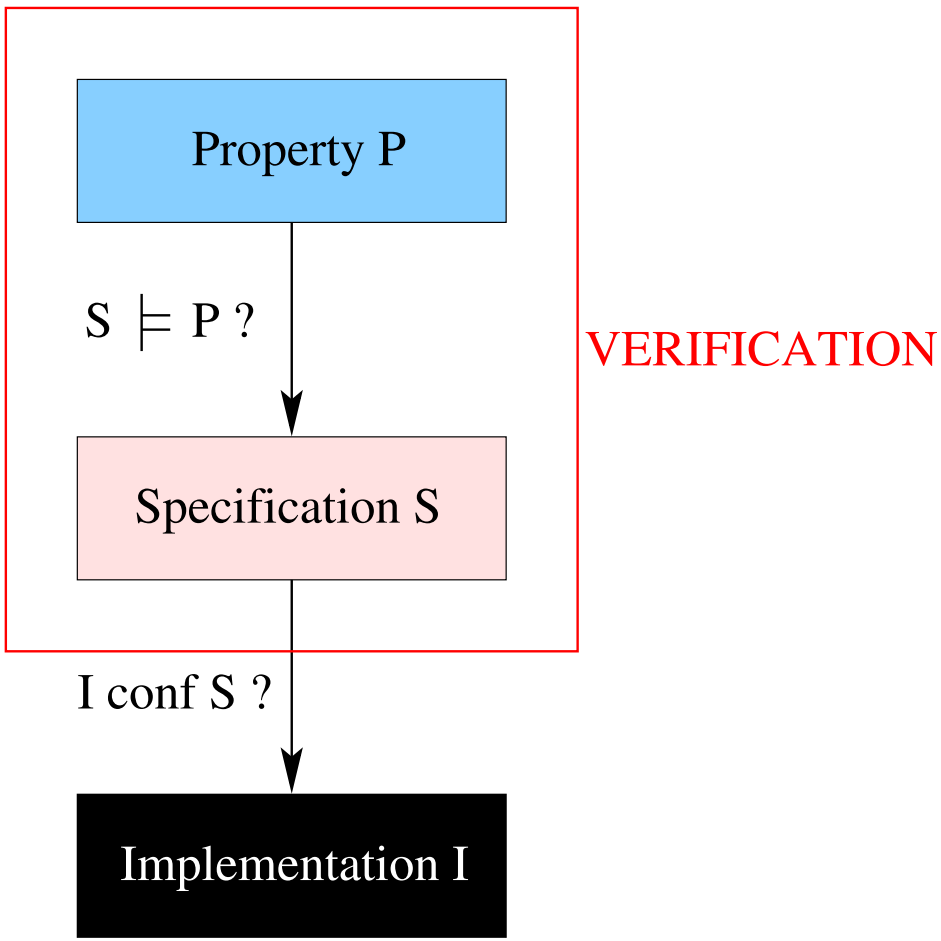
Two main approaches of MBT :

- Finite State Machines
- Labeled Transition Systems

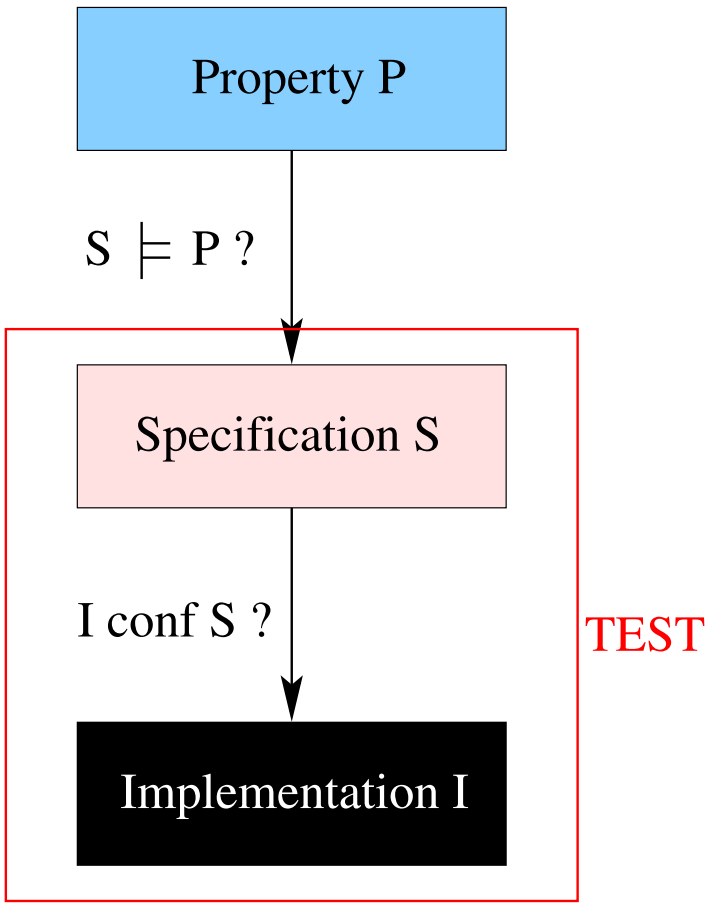
General schema



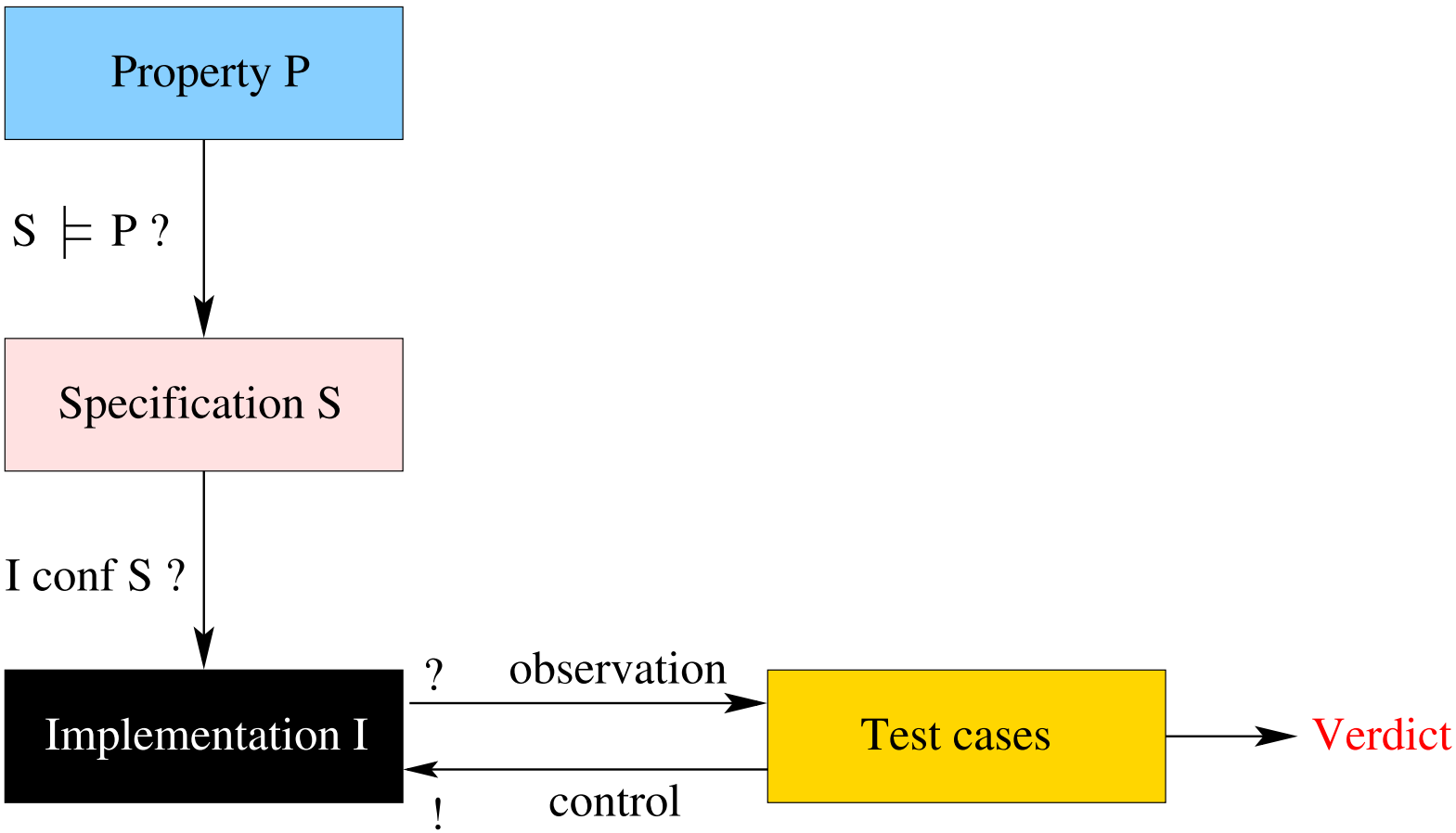
General schema



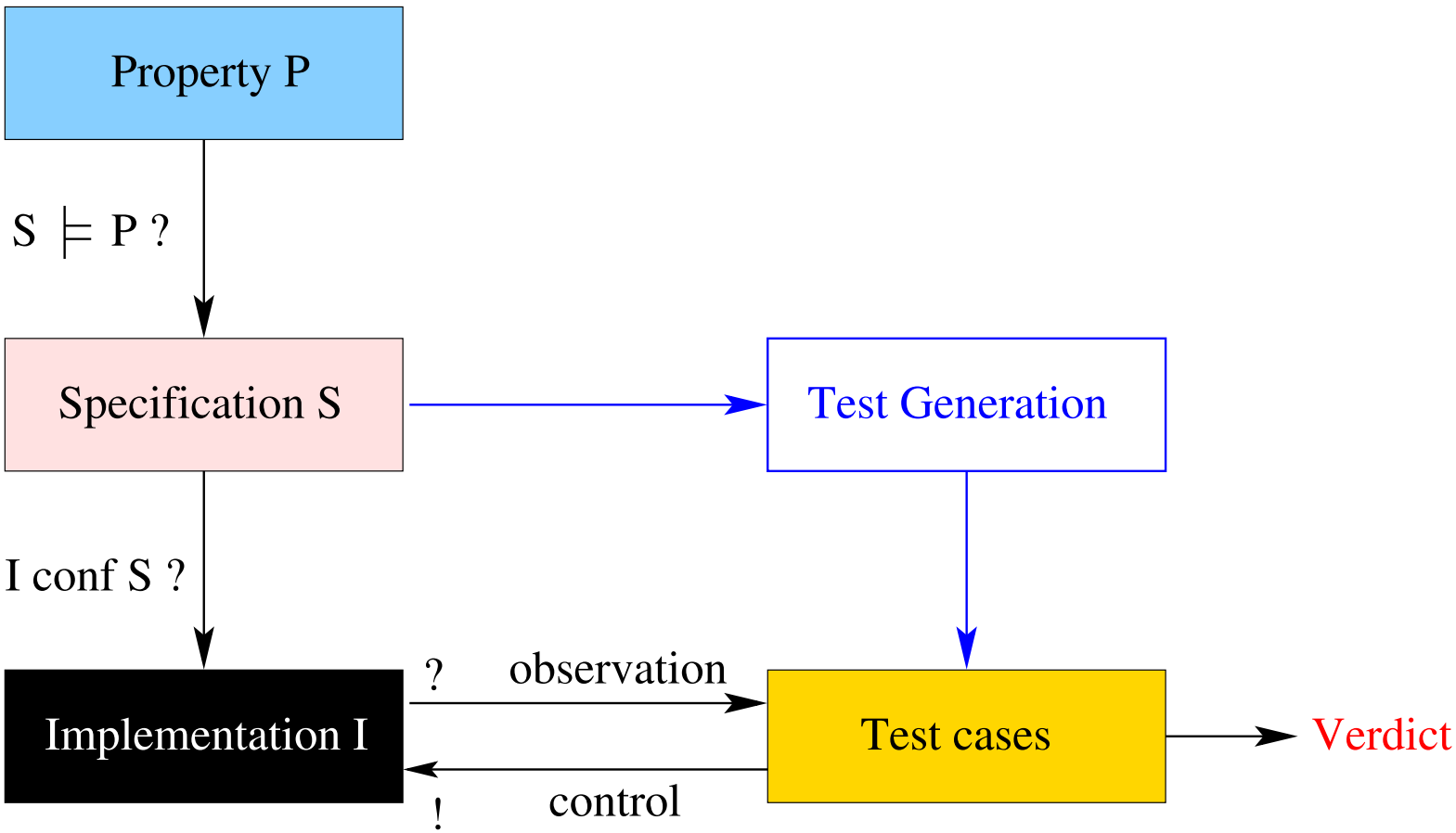
General schema



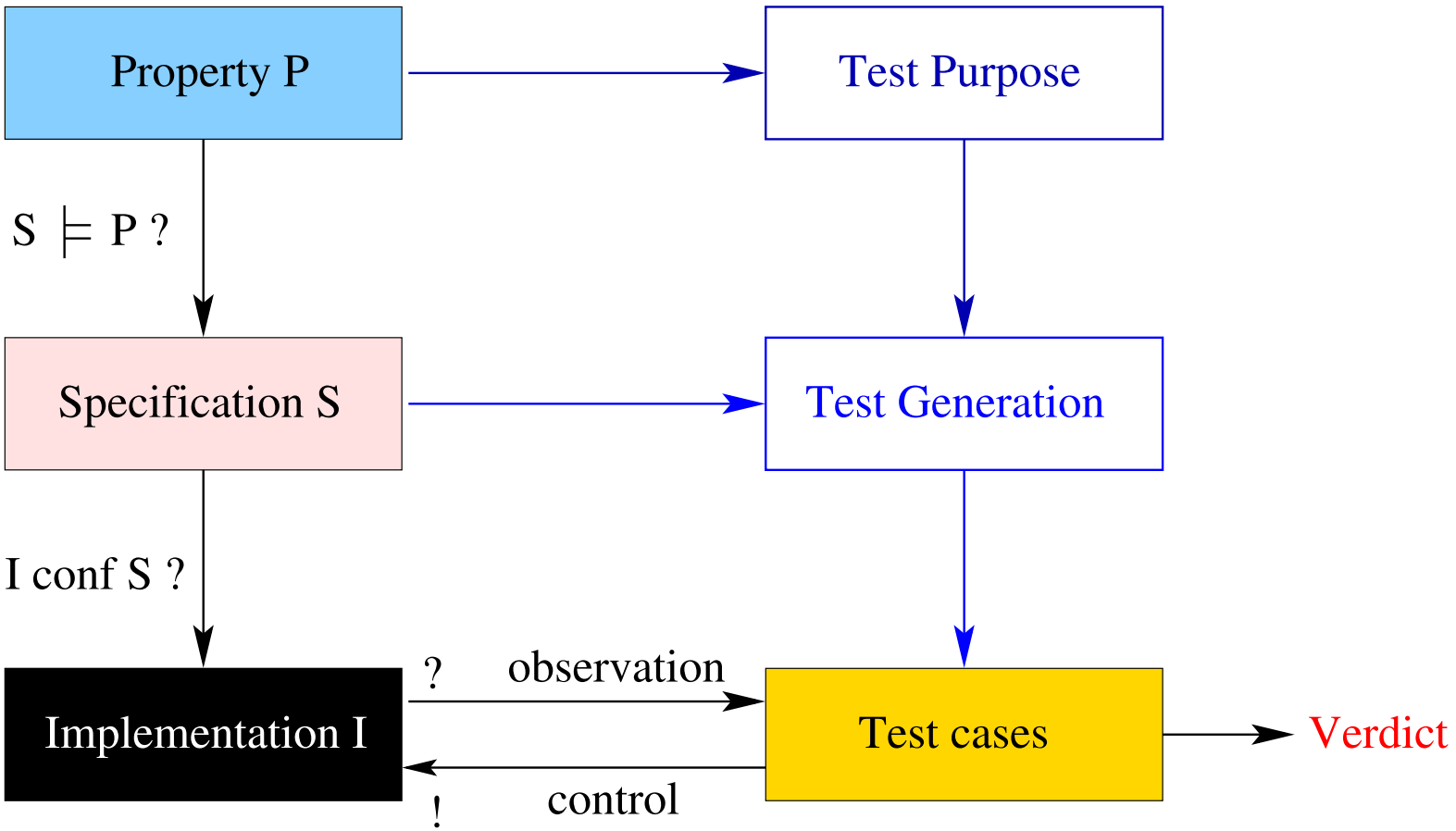
General schema



General schema



General schema



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 : tests = testgen(Spec (+ TestPurpose))

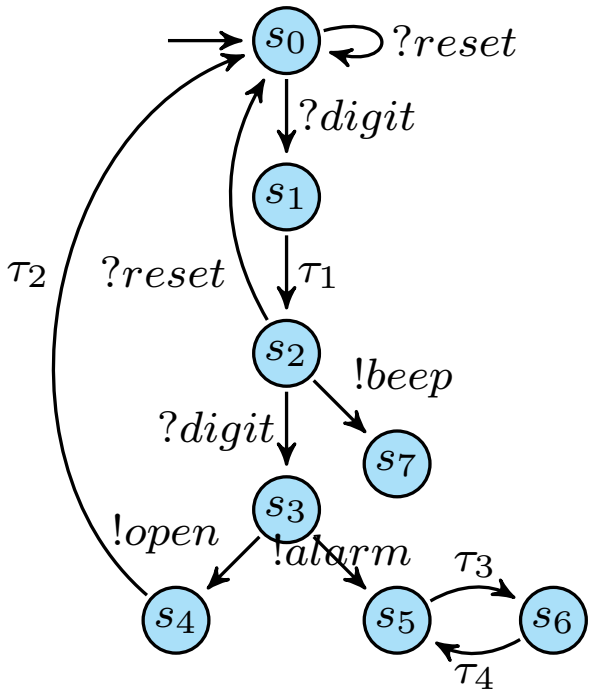
Outline

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.

Input Output Labelled Transition System (IOLTS)

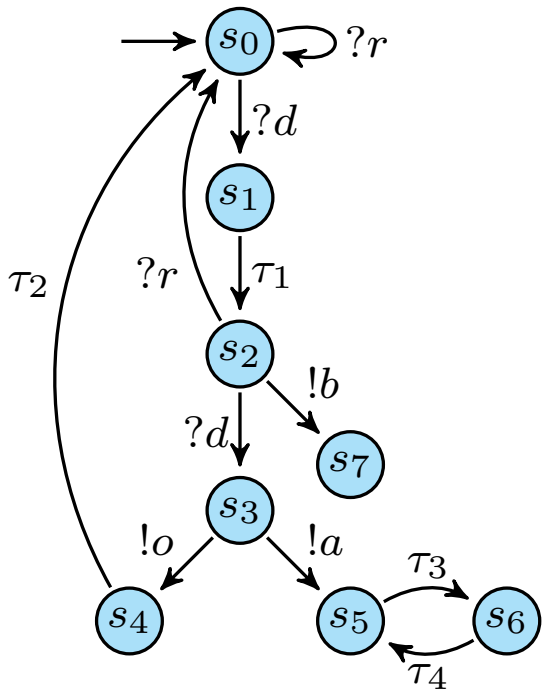


$M = (Q^M, A^M, \longrightarrow_M, q_0^M)$ with :

- 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)

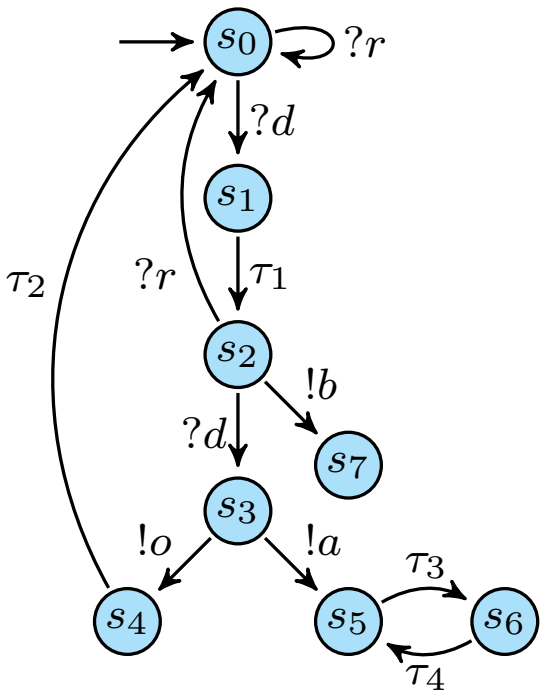


$M = (Q, A, \longrightarrow, q_0)$ with :

- 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:

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

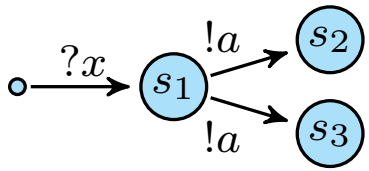
P after σ: set of states reachable from P after observation σ:

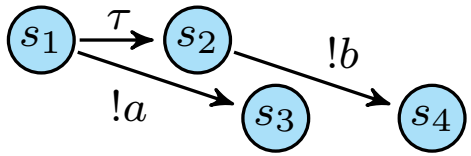
$\{s_2\}$ after $?d.!o = \{s_0, s_4\}$

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

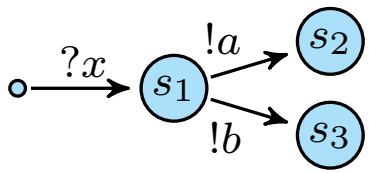
M after $\sigma \triangleq \{q_0\}$ after σ

Non-determinism





Not to be confused with uncontrolled choice

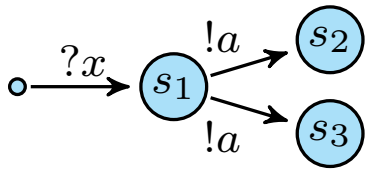


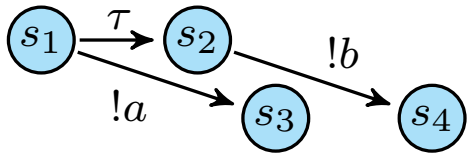
M is deterministic if it has no internal action,
and $\forall q, q', q'' \in Q, \forall a \in A_{VIS}, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \Rightarrow q' = q''$

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.

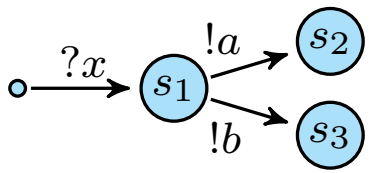
$$Traces(M) = Traces(det(M))$$

Non-determinism





Not to be confused with uncontrolled choice



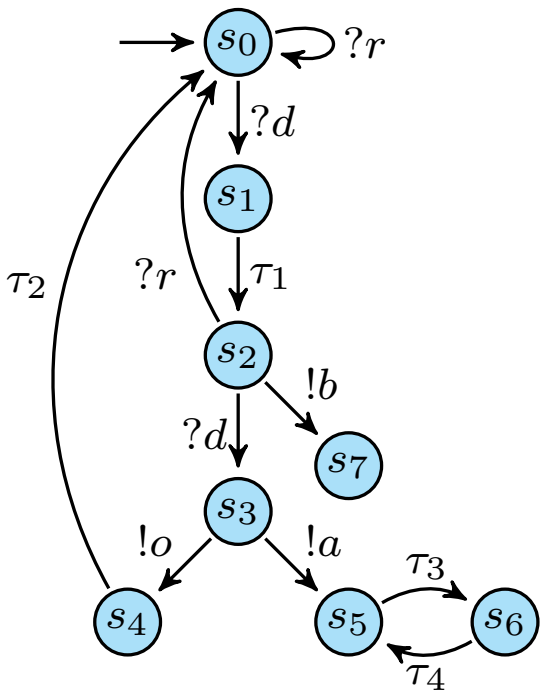
M is deterministic if it has no internal action,
and $\forall q, q', q'' \in Q, \forall a \in A_{VIS}, (q \xrightarrow{a} q' \wedge q \xrightarrow{a} q'') \Rightarrow q' = q''$

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.

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

Observation of quiescence

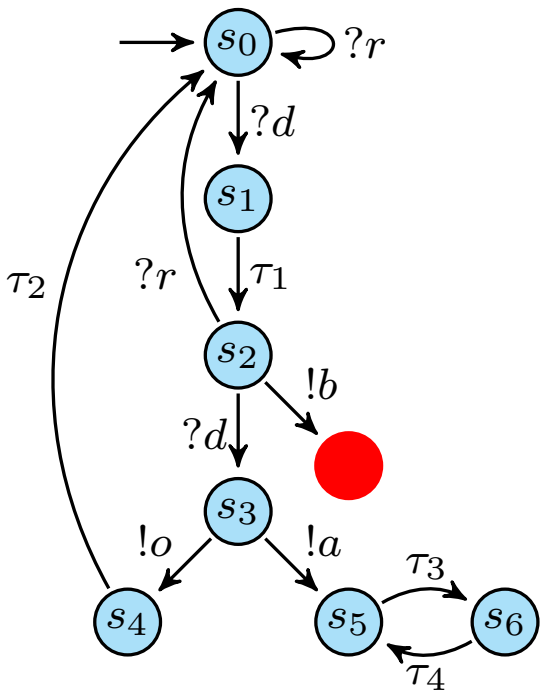
In testing practice, one can observe traces of the IUT, but also its quiescences with timers.
Only quiescences of IUT unspecified in S should be rejected.



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.

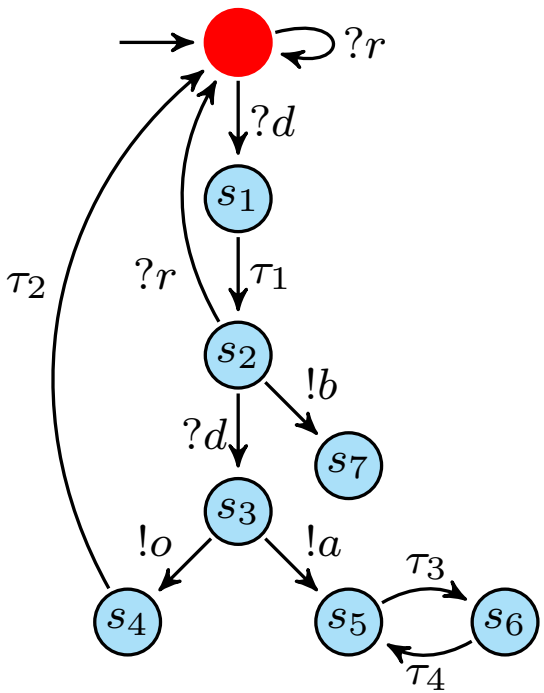


Notation : $\Gamma(q) \triangleq \{a \in A \mid q \xrightarrow{a}\}$
deadlock : no possible evolution :
$\Gamma(q) = \emptyset$.

.

Observation of quiescence

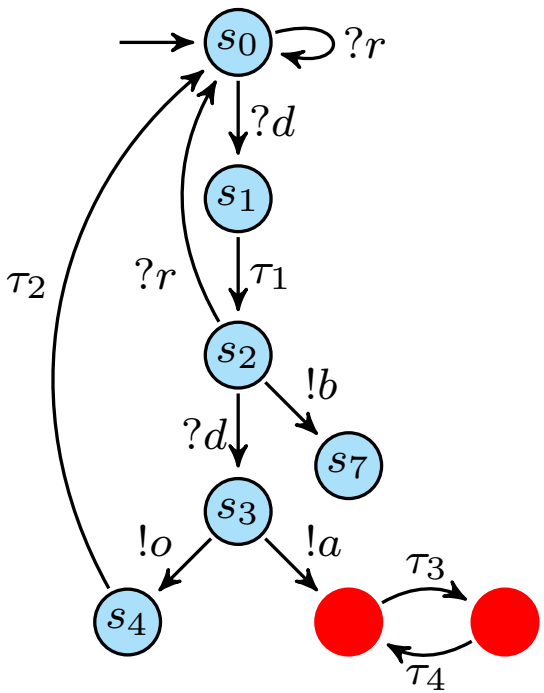
In testing practice, one can observe traces of the IUT, but also its quiescences with timers.
Only quiescences of IUT unspecified in S should be rejected.



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

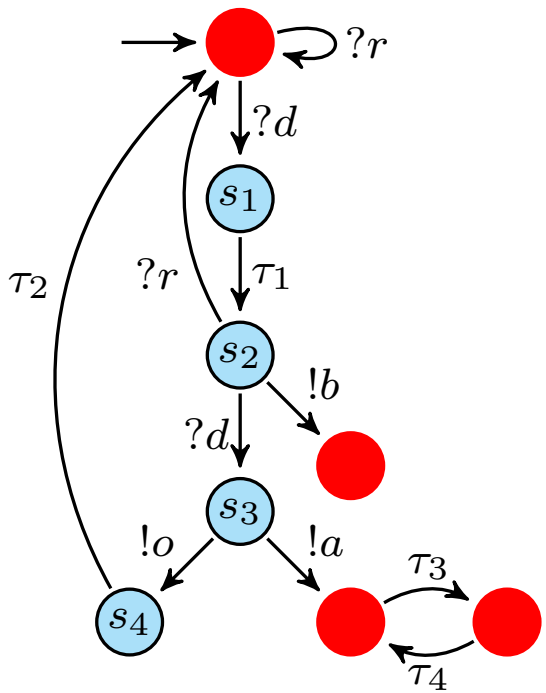
In testing practice, one can observe traces of the IUT, but also its quiescences with timers.
Only quiescences of IUT unspecified in S should be rejected.



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

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.



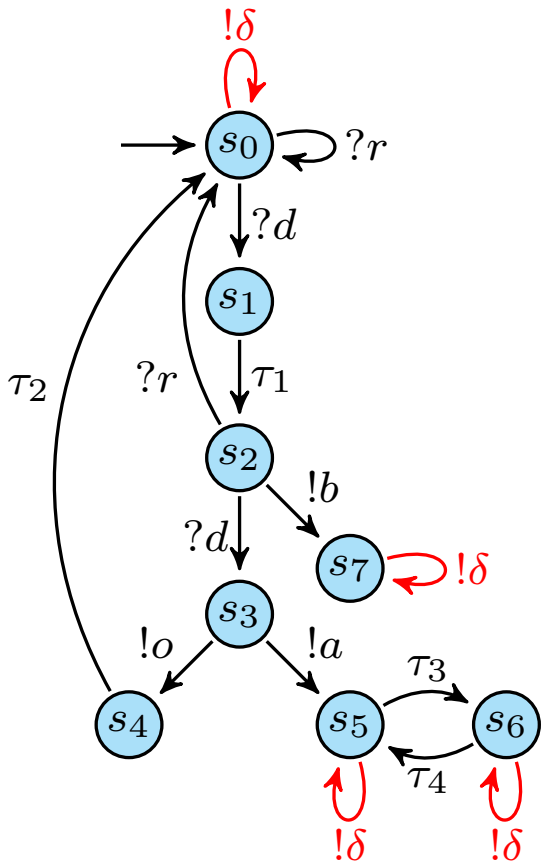
Notation : $\Gamma(q) \triangleq \{a \in A \mid q \xrightarrow{a}\}$
deadlock : no possible evolution :
$\Gamma(q) = \emptyset$.
outputlock : system waiting for an action :
$\Gamma(q) \subseteq A_I$.
livelock : internal actions loop :
$\exists \tau_1, ...\tau_n : q \xrightarrow{\tau_1 ... \tau_n} q$.

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

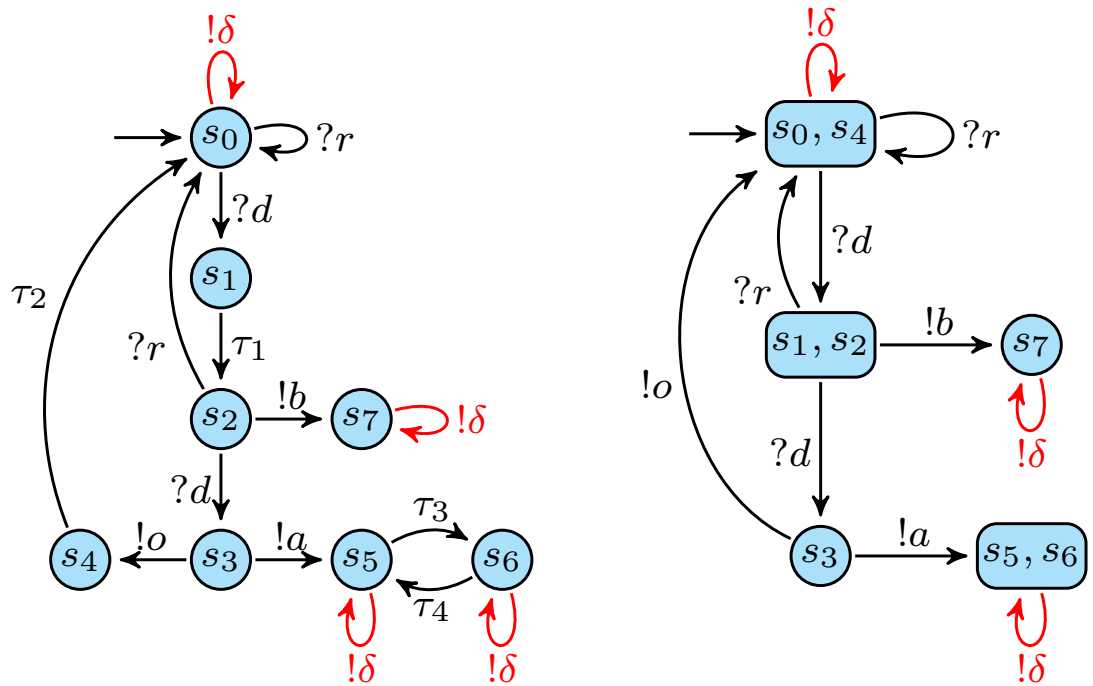
Suspension automaton

Quiescence : special output δ

The suspension ioLTS of $M = (Q, A, \longrightarrow, q_0)$ is an ioLTS $\Delta(M) = (Q, A \cup \{\delta\}, \longrightarrow_{\Delta(M)}, q_0)$ where $\longrightarrow_{\Delta(M)} = \longrightarrow \cup \{q \longrightarrow_\delta q | q \in quiescent(M)\}$.



Suspension traces



$\Delta(S)$ $det(\Delta(S))$

Suspension traces

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

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

Testing framework

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

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

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

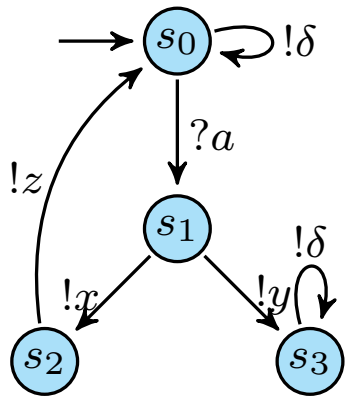
$$IUT \textbf{ ioco } S \triangleq \forall \sigma \in STraces(S), Out(\Delta(IUT) \text{ after } \sigma) \subseteq Out(\Delta(S) \text{ after } \sigma)$$

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

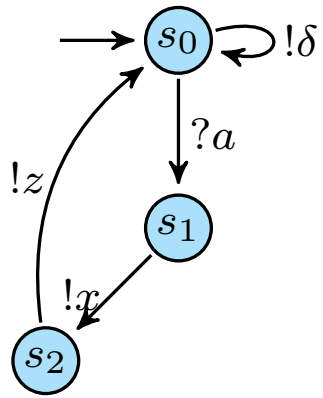
[a] A_O^δ is equivalent notation for A_O since δ is an output of $\Delta(S)$ and $\Delta(IUT)$

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

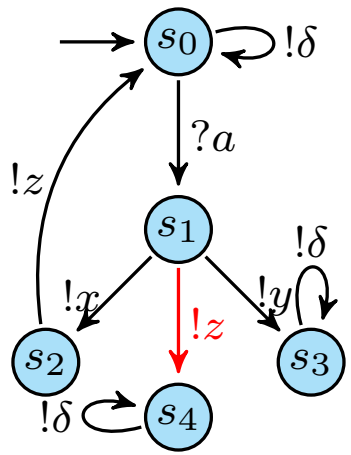
ioco: example



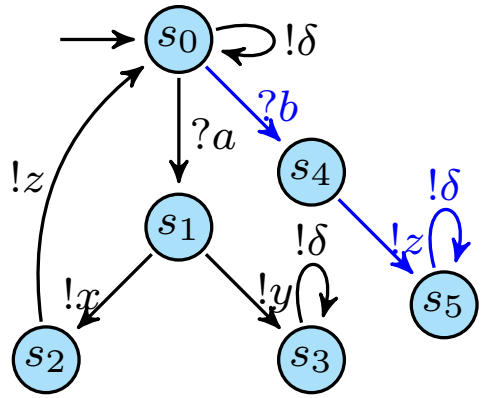
specification $\Delta(S)$



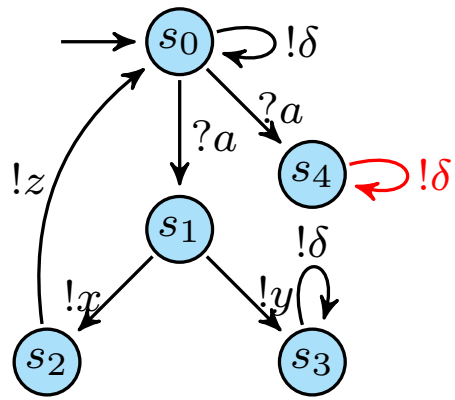
I_1: Implem. choice



I_3: Unspec. output



I_2: Implem. of a partial spec.



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 permitting to detect non-conformance of implementation IUT.

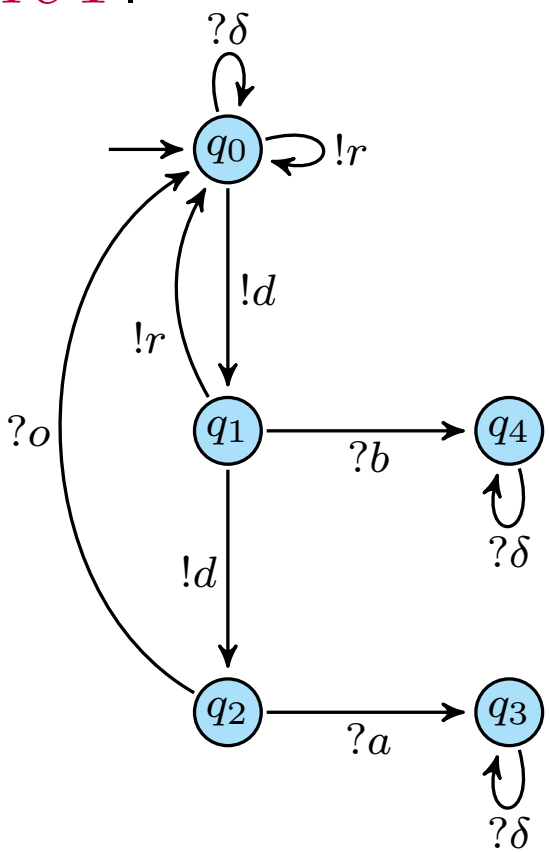
Canonical Tester

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 permitting to detect non-conformance of implementation IUT.

- $Q^c = Q^d \cup \{\textbf{Fail}\}$ and $q_0^c = q_0^d$
- $A^c = A_I^c \cup A_O^c$ where $A_I^c = A_O^d$ and $A_O^c = A_I^d$ input / output inversion
- $\longrightarrow_c = \longrightarrow_d \cup \{q \xrightarrow{a}_c \textbf{Fail} \mid q \in Q^d, a \in A_I^c \wedge \neg(q \xrightarrow{a}_d)\}$, all non-specified outputs lead to **Fail**.

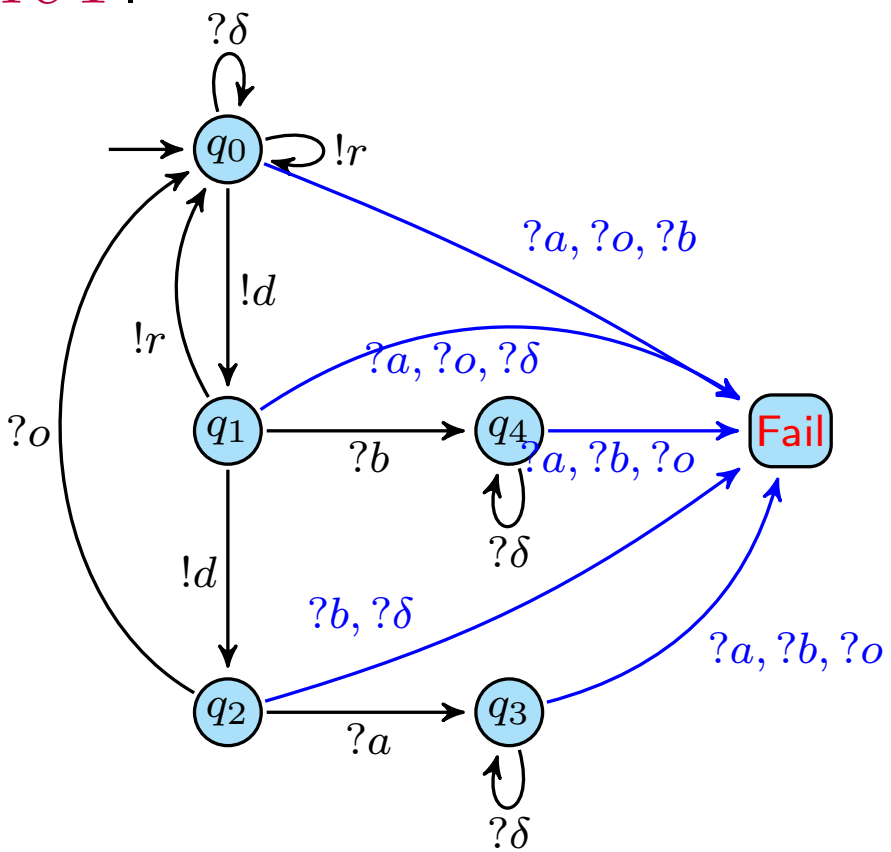
Canonical Tester

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 permitting to detect non-conformance of implementation IUT.



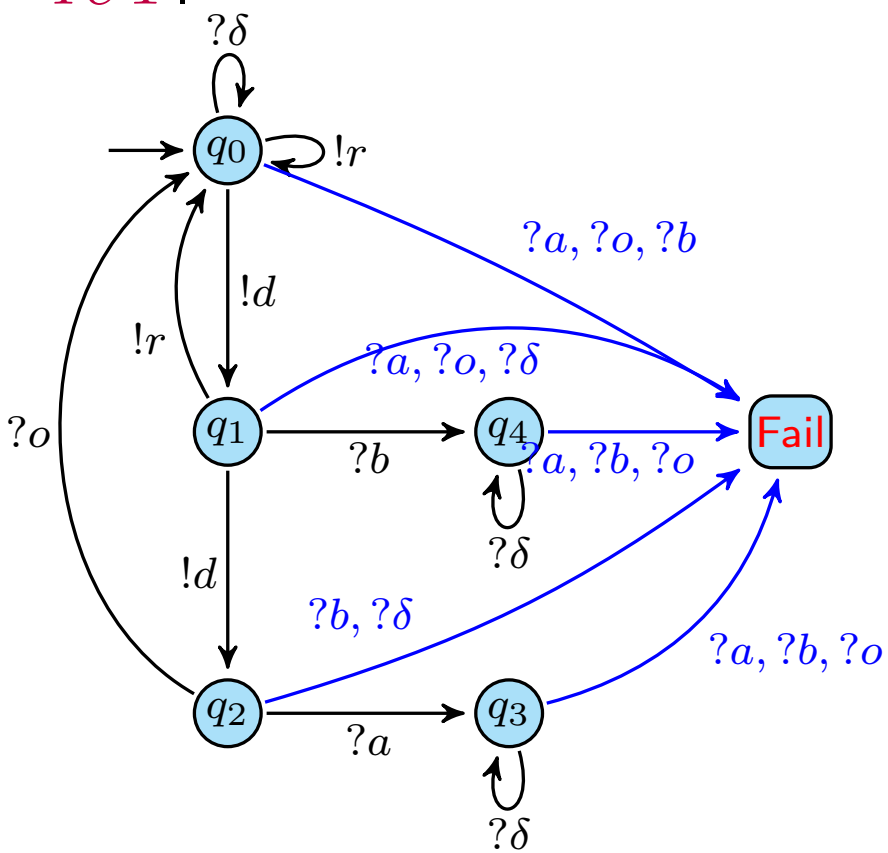
Canonical Tester

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 permitting to detect 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 permitting to detect non-conformance of implementation IUT.

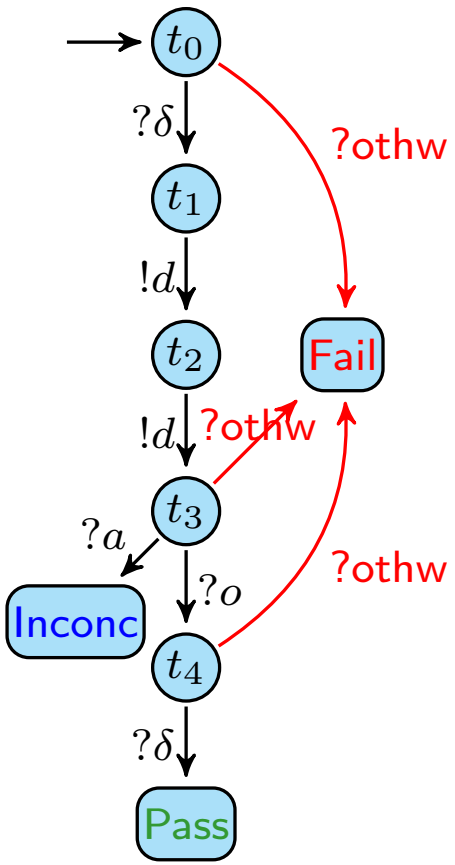


IUT **ioco** $S \iff$
$STraces(IUT) \cap$
$Traces_{\mathsf{Fail}}(Can(S)) = \emptyset$

Test cases

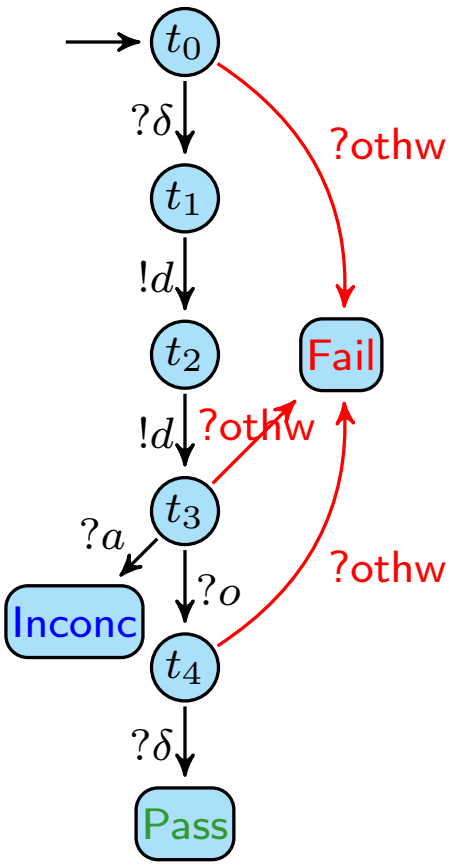
A test case is a deterministic ioLTS $(Q^{\mathsf{TC}}, A^{\mathsf{TC}}, \longrightarrow_{\mathsf{TC}}, t_0^{\mathsf{TC}})$, equipped with **verdict** states: Pass, Fail and Inconc s.t.

- $A_O^{\mathsf{TC}} = A_I^{\mathsf{S}}$ and $A_I^{\mathsf{TC}} = A_O^{\mathsf{S}} \cup \{\delta\}$ (input / output inversion)
- TC is controllable, i.e. never have to choose btw. several outputs or btw. inputs and outputs : $\forall q \in Q^{\mathsf{TC}}, (\exists a \in A_O^{\mathsf{TC}}, q \xrightarrow{a}_{\mathsf{TC}} \Rightarrow \forall b \in A^{\mathsf{TC}}, (b \neq a \Rightarrow q \not\xrightarrow{b}_{\mathsf{TC}}))$
- All states permitting an input, are input-complete, except verdict states.

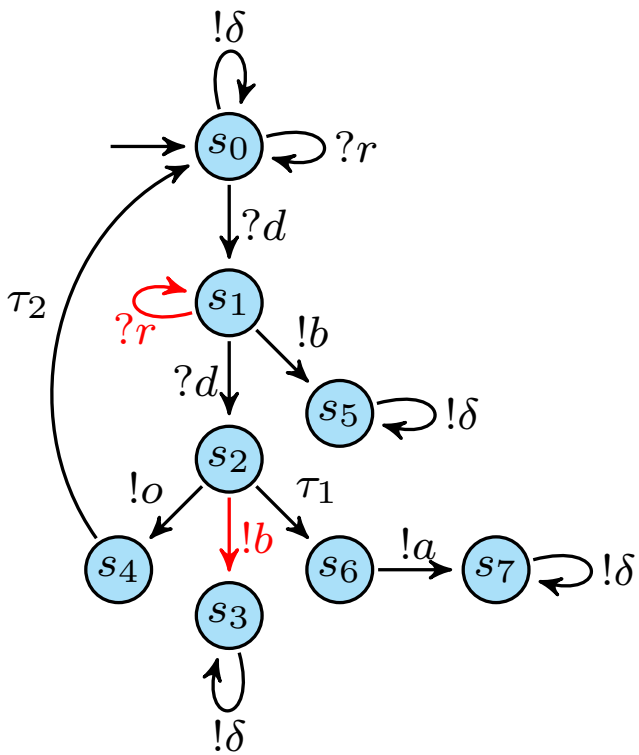


Test execution

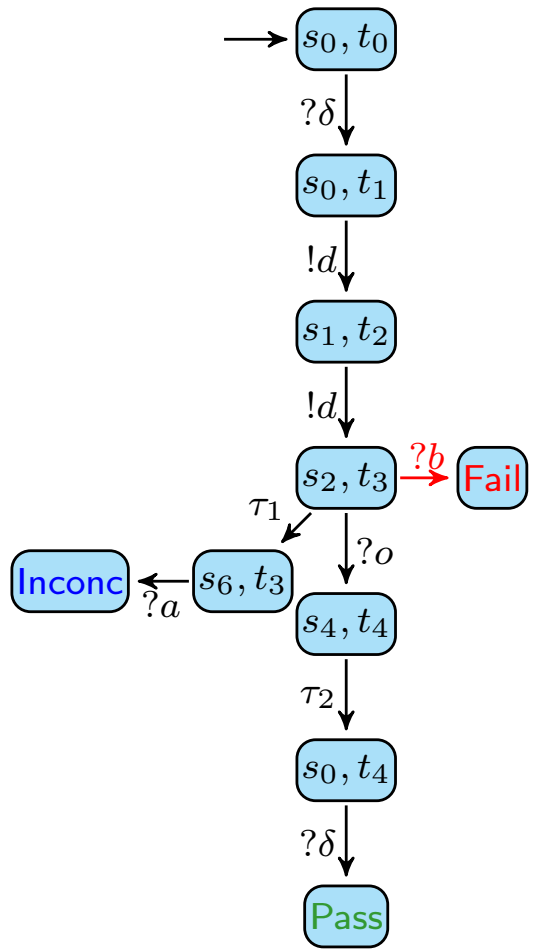
Modelled by the parallel composition $TC \| \Delta(IUT)$ synchronizing on common visible actions



$TC1$



$\Delta(IUT)$



$TC1 \| \Delta(IUT)$

Properties of test suites

TC fails IUT iff an execution of $TC \| \Delta(IUT)$ 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* $\triangleq$
  $\forall IUT : (IUT \textbf{ ioco } S \implies \forall TC \in TS : \neg(TC \ fails \ IUT))$,
  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.

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* $\triangleq$
  $\forall IUT : (IUT \textbf{ ioco } S \implies \forall TC \in TS : \neg(TC \ fails \ IUT))$,
  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

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 $Can(S)$

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 $Can(S)$

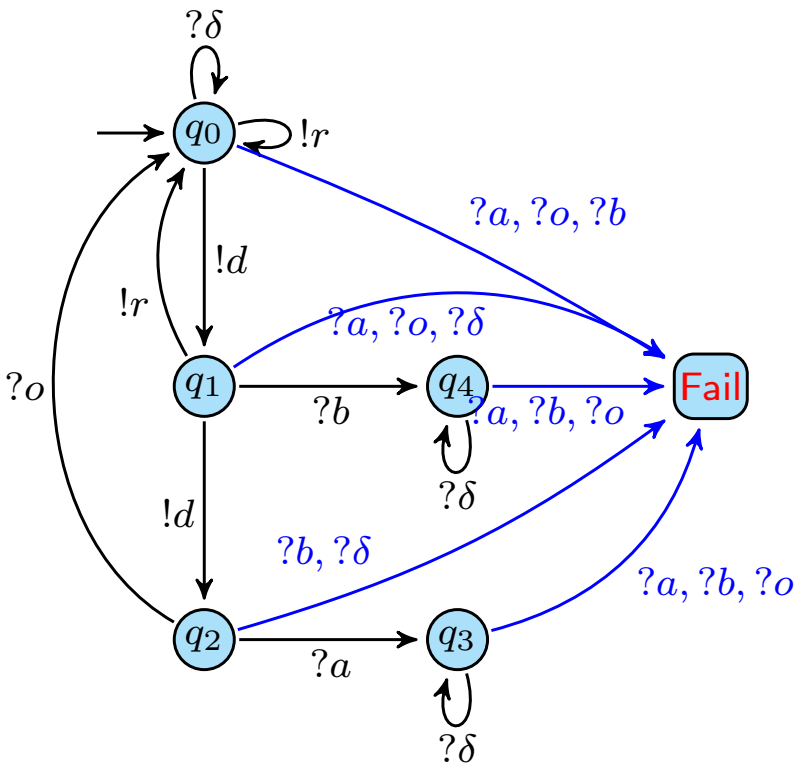
Start in q_0^c. After any trace σ in $Can(S)$

- 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.

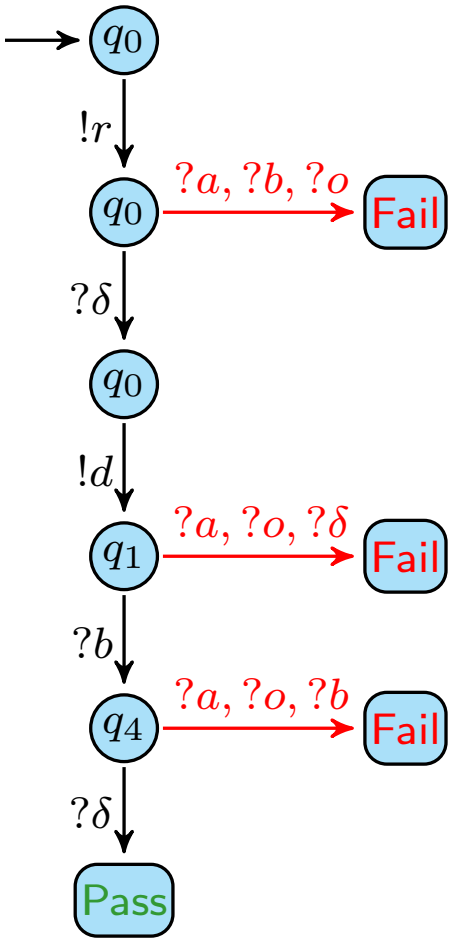
Properties

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

Examples

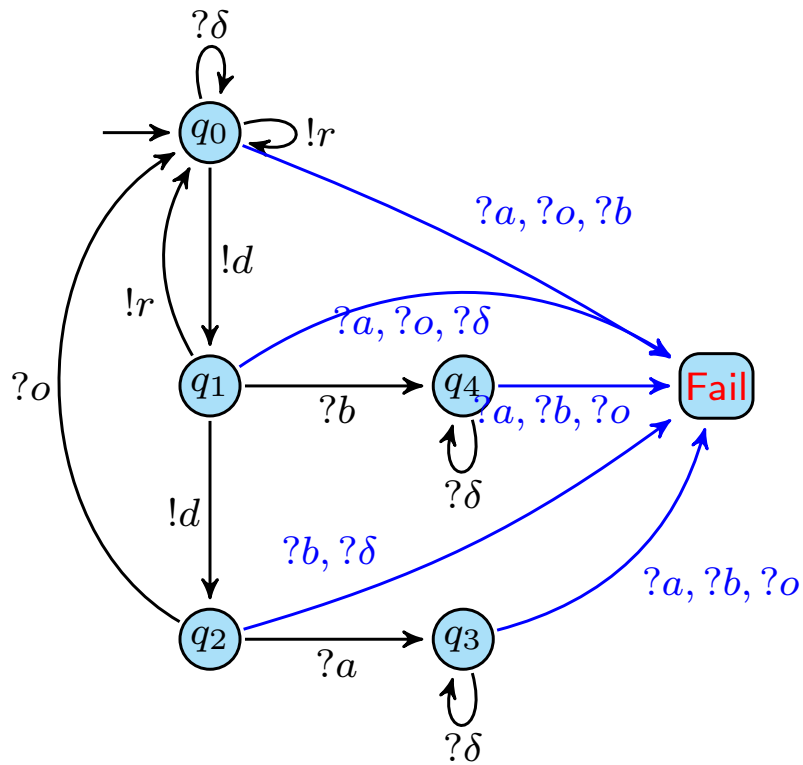


$Can(S)$

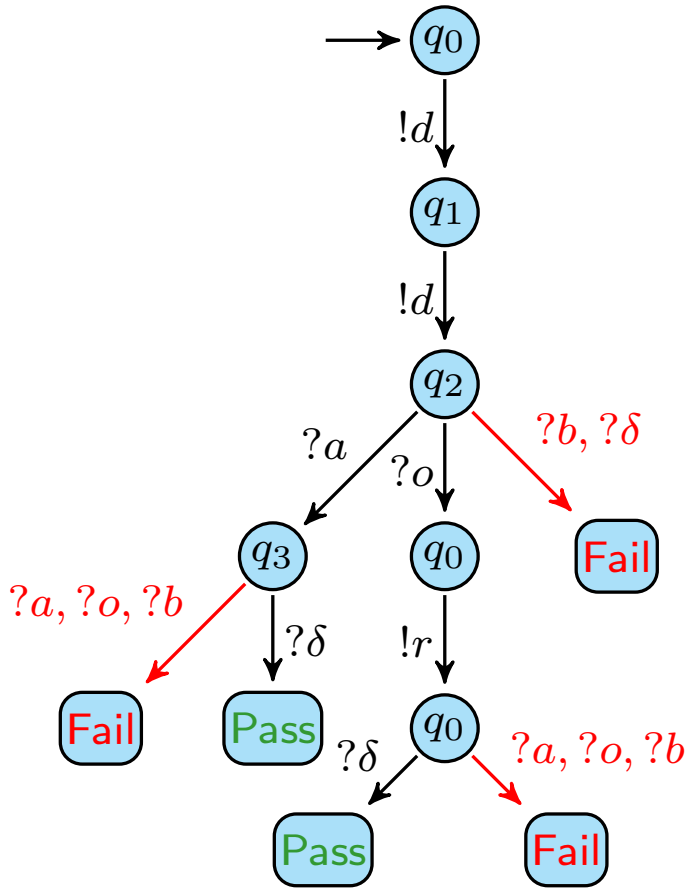


$TC1$

Examples



$Can(S)$



$TC2$

Test Purpose generation

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

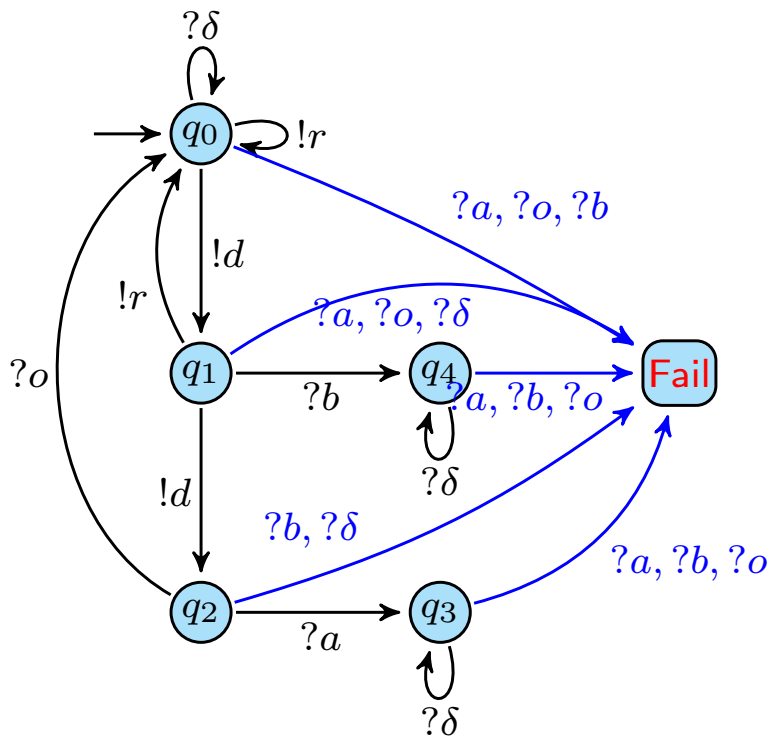
Main characteristics of Test Purpose Generation:

- test selection by test purposes describing a set of behaviors to be tested, targeted by a test case,
- off-line selection, *a posteriori* execution.

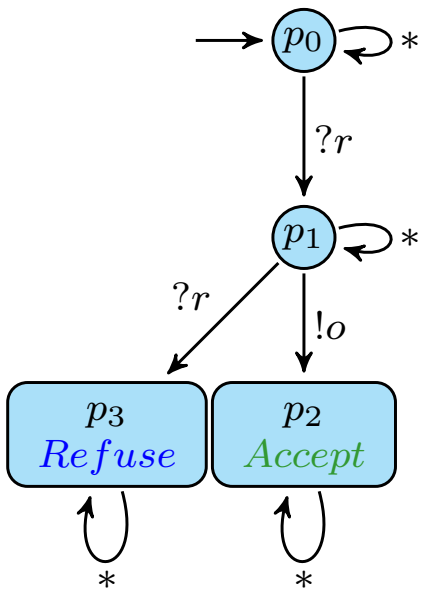
Test Purpose definition

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, s.t. $A^{TP} = A^S_{VIS} \cup \{\delta\}$

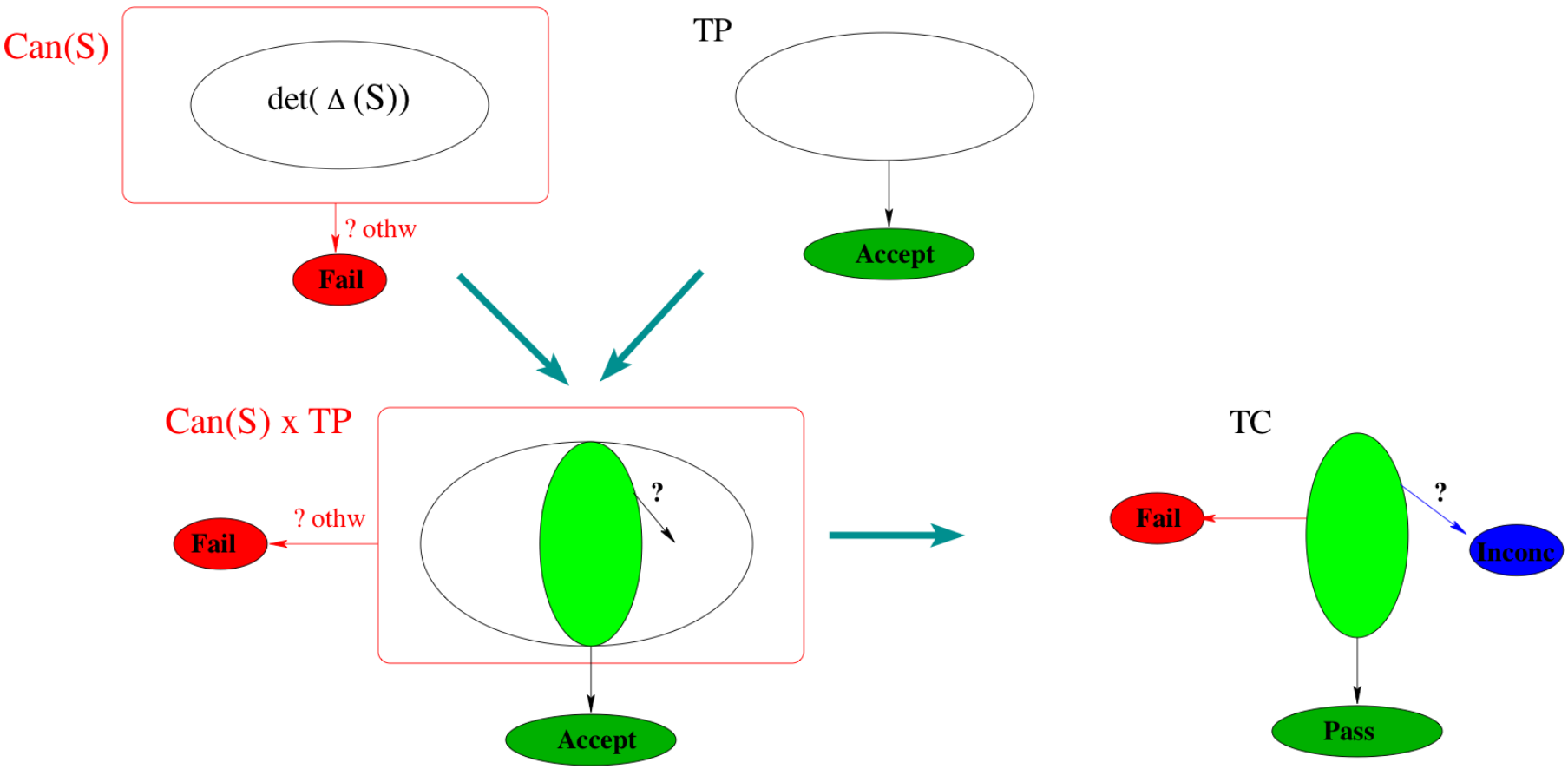


$Can(S)$



TP

Selection principle



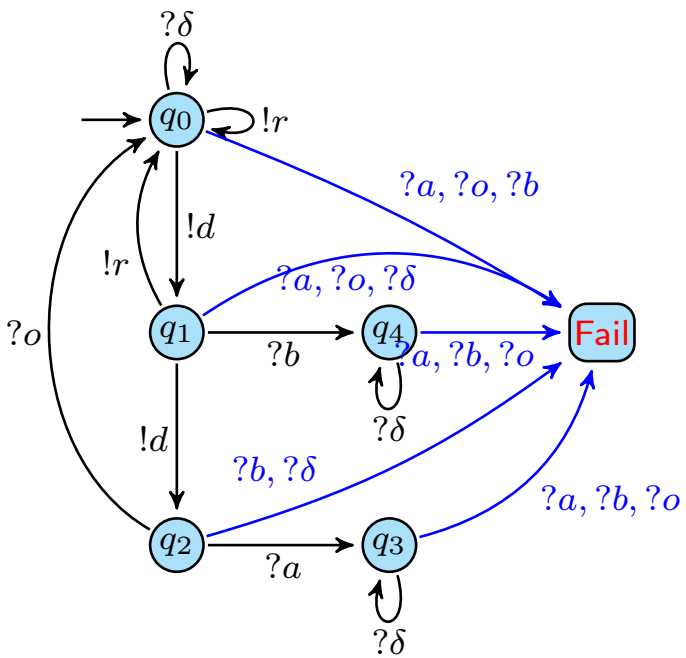
Synchronous Product : definition

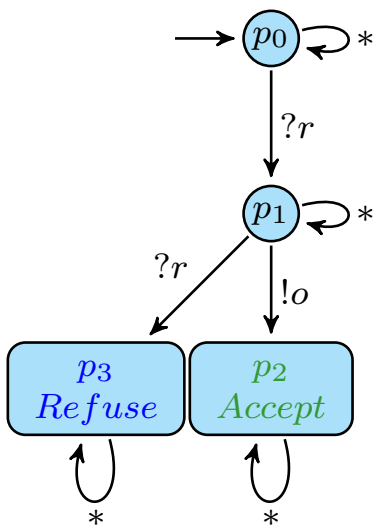
Definition of Synchronous Product

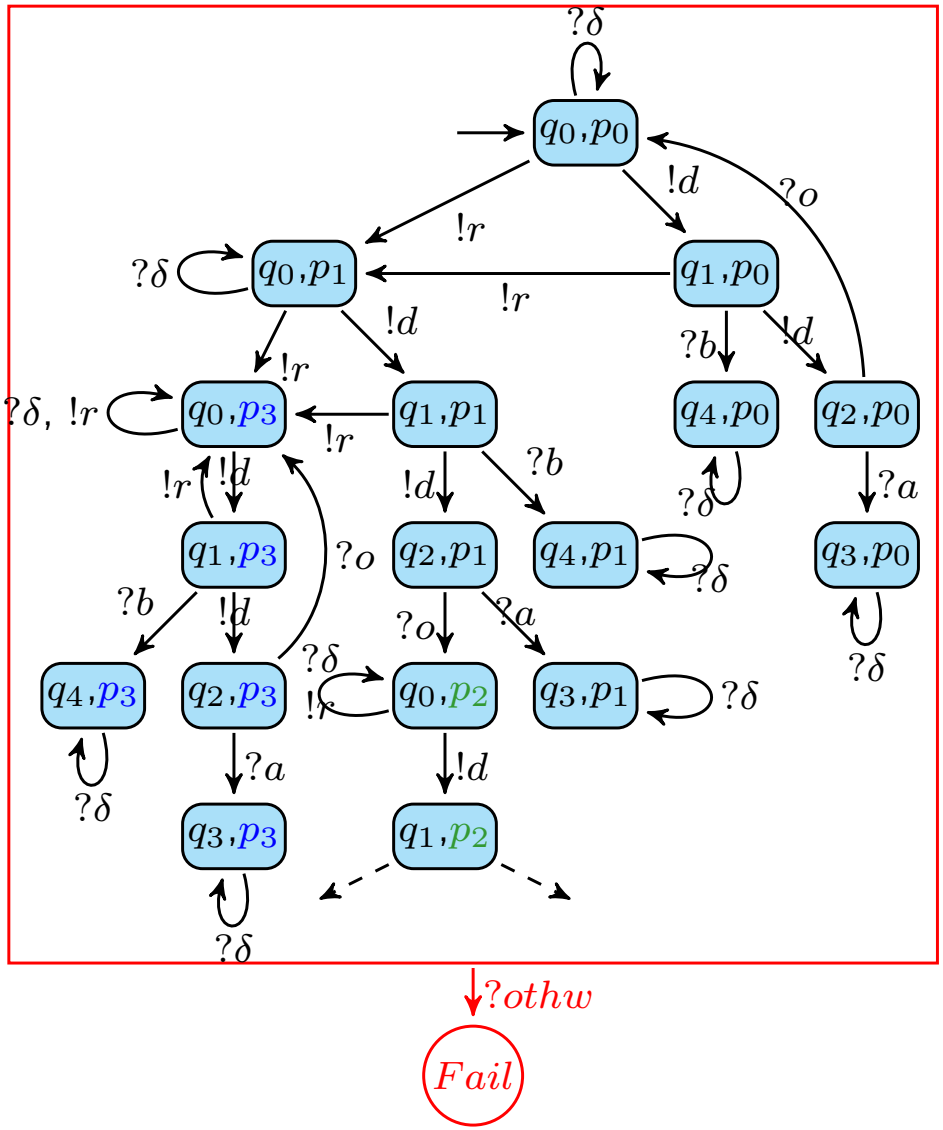
The Synchronous Product of two ioLTS $M_1 = (Q^{M1}, A, \longrightarrow_{M1}, q_0^{M1})$, and $M_2 = (Q^{M2}, A, \longrightarrow_{M2}, q_0^{M2})$ is the ioLTS $M_1 \times M_2 = (Q^{M1} \times Q^{M2}, A, \longrightarrow, q_0^{M1} \times q_0^{M2})$ where $\longrightarrow$ is defined by :

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

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

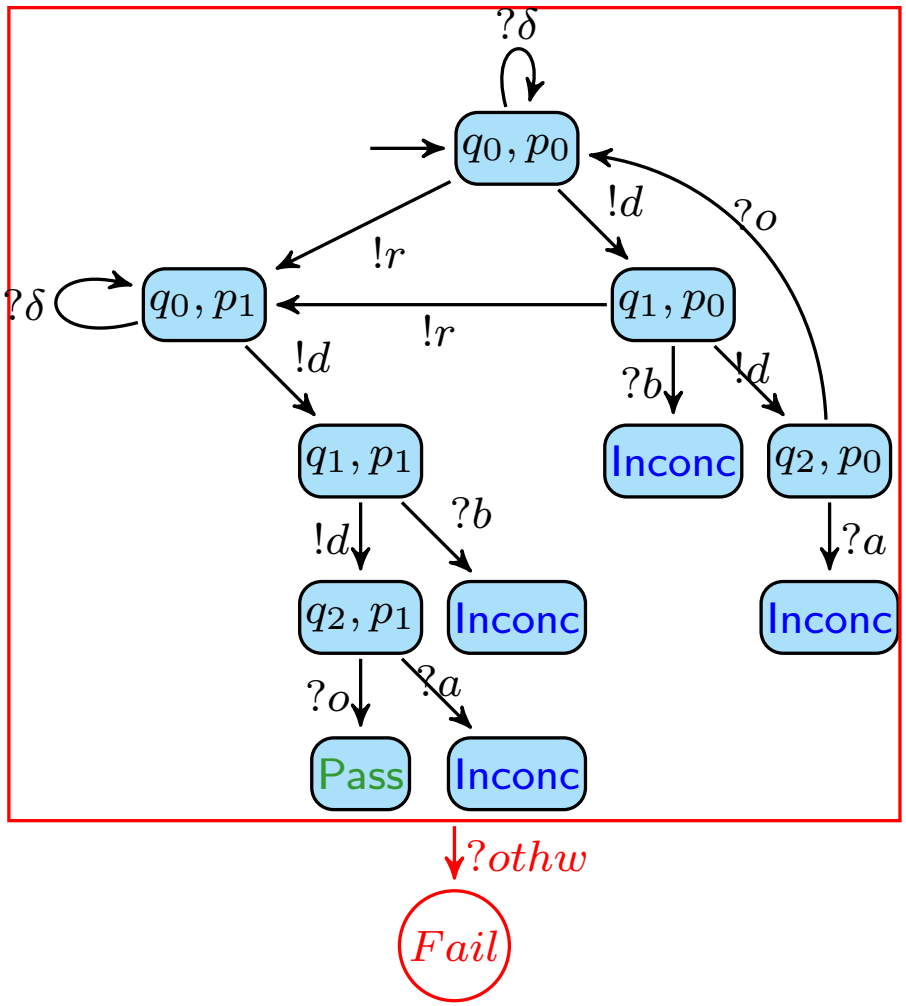




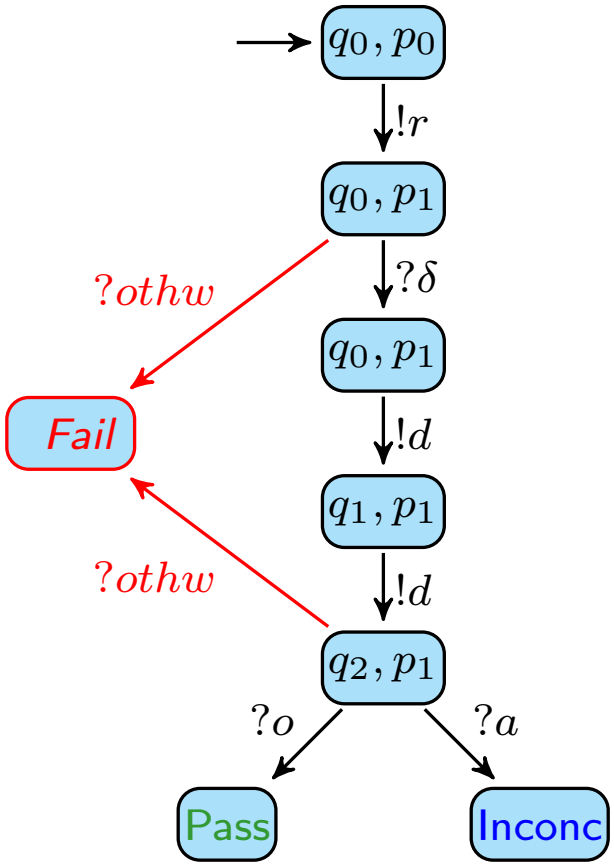


Complete Test Graph (CTG)

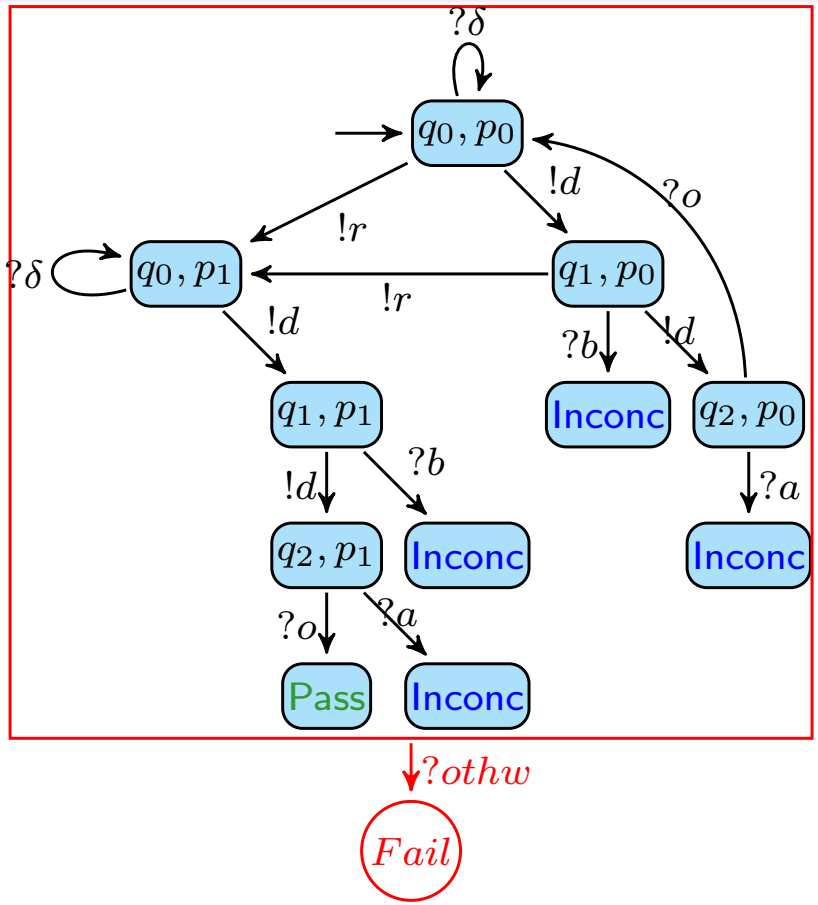
- Keep the first $Accept$ state in a path
- If $q \in coreach(Pass)$ keep q
- If $q \in \{Fail\}$ keep q
- If $q \notin coreach(Pass)$ input (tester point of view) successor of a state $q' \in coreach(Pass)$ then $Inconc$



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.

Conclusion

- Testing theory for ioLTS
- Test generation for **finite** ioLTS
  - Non-deterministic selection: unfolding of $Can(S)$
  - Selection by test purpose: for finite ioLTS based on co-reachability analysis.
  - Soundness and exhaustiveness.

Outline

References

Part essentially based on :

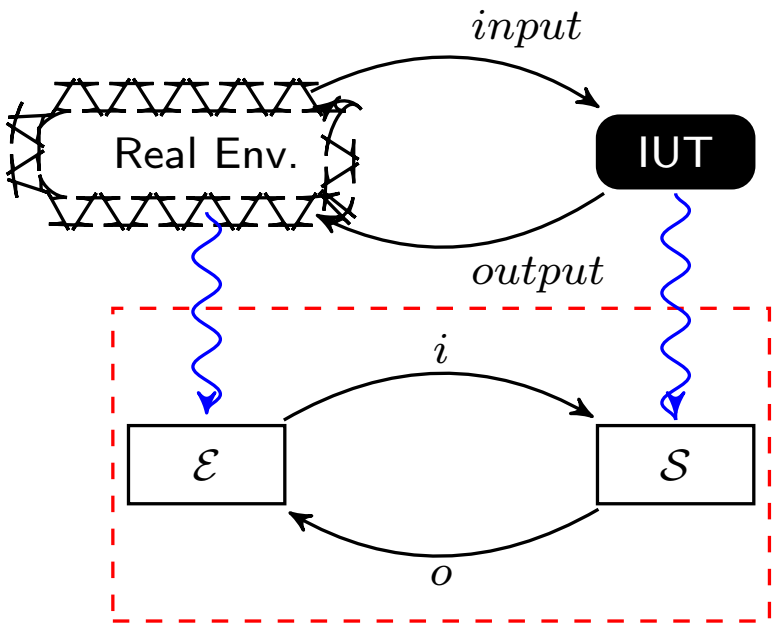
- [HLMNPS08] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, LNCS, vol. 4949. Springer Berlin / Heidelberg, 2008, pp. 77–117.
- [MLN04] M. Mikucionis, K. G. Larsen, and B. Nielsen, "T-uppaal: Online model-based testing of real-time systems," in *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*. IEEE Computer Society, 2004, pp. 396–397.
- [KT04] M. Krichen and S. Tripakis, "Black-box conformance testing for real-time systems," in *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004*, LNCS vol. 2989. Springer, 2004, pp. 109–126.

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

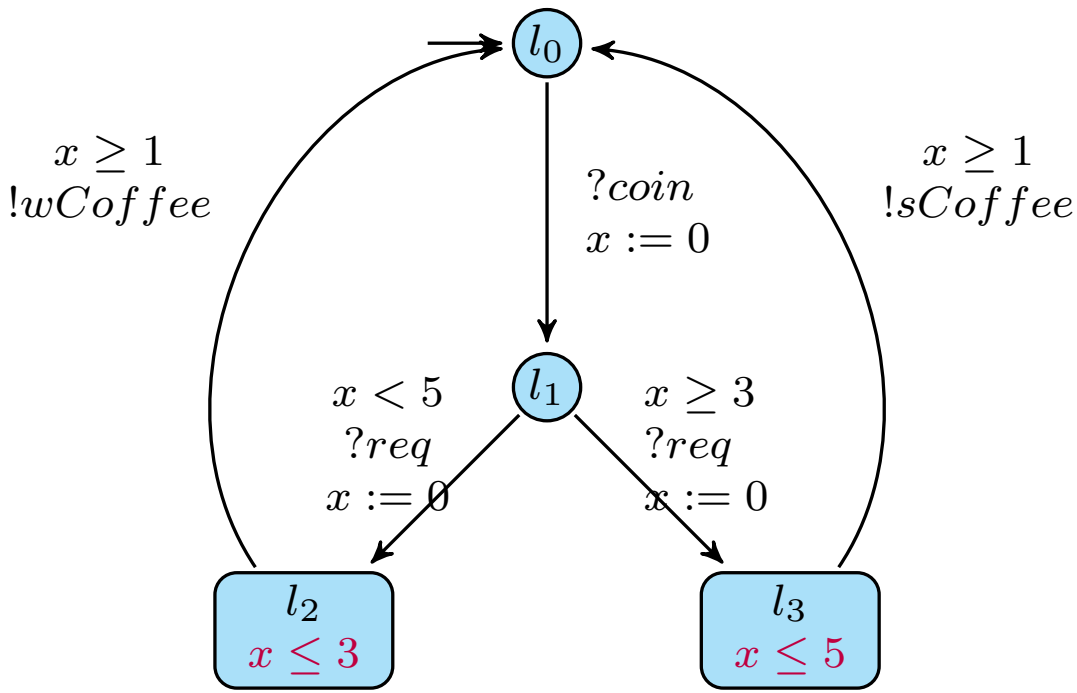
"Uppaal-like" approach

Explicit and separate model of the environment



+ 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 $\tau \notin A$. $(A_\tau \triangleq A \cup \{\tau\})$

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

TIOTS definition

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

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

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

Observable Timed Traces $TTr(s)$

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

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

After

s After $\sigma = \{s' | s \stackrel{\sigma}{\Rightarrow} s'\}$, $\quad S'After\sigma = \bigcup_{s \in S'} s$ After σ

Out

$Out(s) = \{a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \stackrel{a}{\Rightarrow}\}$ $\quad Out(S') = \bigcup_{s \in S'} Out(s)$

Notations / Definitions

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

Observable Timed Traces $TTr(s)$

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

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

After

$s \text{ After } \sigma = \{s' | s \stackrel{\sigma}{\Rightarrow} s'\}, \quad S' After \sigma = \bigcup_{s \in S'} s \text{ After } \sigma$

Out

$Out(s) = \{a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \stackrel{a}{\Rightarrow}\} \quad Out(S') = \bigcup_{s \in S'} Out(s)$

Notations / Definitions

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

Observable Timed Traces $TTr(s)$

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

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

After

s After $\sigma = \{s' | s \stackrel{\sigma}{\Rightarrow} s'\}$, $\quad S'After\sigma = \bigcup_{s \in S'} s$ After σ

Out

$Out(s) = \{a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \stackrel{a}{\Rightarrow}\} \qquad Out(S') = \bigcup_{s \in S'} Out(s)$

Notations / Definitions

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

Observable Timed Traces $TTr(s)$

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

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

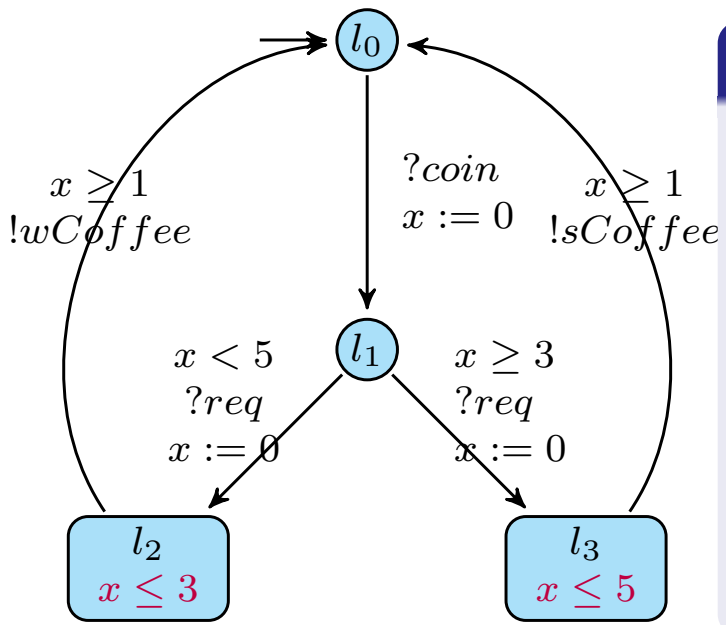
After

s After $\sigma = \{s' | s \stackrel{\sigma}{\Rightarrow} s'\}$, $\quad S'After\sigma = \bigcup_{s \in S'} s$ After σ

Out

$Out(s) = \{a \in A_{out} \cup \mathbb{R}_{\geq 0} | s \stackrel{a}{\Rightarrow}\}$ $\quad Out(S') = \bigcup_{s \in S'} Out(s)$

Timed Automata (with Inputs and Outputs) : definition



Given X set of clock variables, $\mathcal{G}(X)$ set of guards, $\mathcal{U}(X)$ set of updates.

Timed Automaton

$TA = (L, l_0, I, E)$ where

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

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

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

Semantics of Timed Automata

Semantics as a TIOTS defined by :

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

$$\frac{\forall d' \leq d . I_l(d')}{(l, \overline{v}) \xrightarrow{d} (l, \overline{v} + d)}$$

- Discrete transitions

$$\frac{l \xrightarrow{g,\alpha,u} l' \wedge g(\overline{v}) \wedge I_{l'}(\overline{v'}), \overline{v'} = u(\overline{v})}{(l, \overline{v}) \xrightarrow{\alpha} (l', \overline{v'})}$$

most reasoning done on the semantics

Relativized timed conformance

- $\mathcal{S} = (S^{\mathcal{S}}, s_0^{\mathcal{S}}, A_{in}, A_{out}, \longrightarrow_{\mathcal{S}})$ a weakly input enabled (i.e. $\forall s \in S^{\mathcal{S}}, \forall i \in A_{in}, s \stackrel{i}{\Rightarrow}$) TIOTS
- $\mathcal{IUT} = (S^{\mathcal{IUT}}, s_0^{\mathcal{IUT}}, A_{in}, A_{out}, \longrightarrow_{\mathcal{IUT}})$ a weakly input enabled TIOTS
- $\mathcal{E} = (E^{\mathcal{E}}, e_0^{\mathcal{E}}, A_{out}, A_{in}, \longrightarrow_{\mathcal{E}})$ (input / output inversion) weakly input enabled TIOTS.

rtioco$_e$

Let $s \in S^{\mathcal{S}}$, $e \in E^{\mathcal{E}}$ and $iut \in S^{\mathcal{IUT}}$:

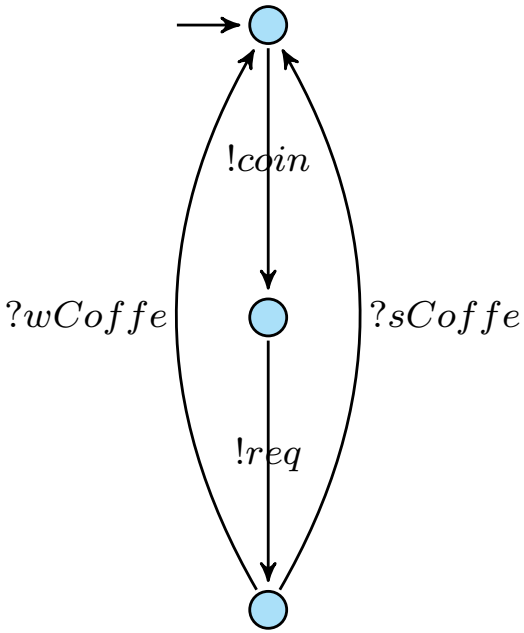
$$iut \textbf{ rtioco}_e \ s$$
iff
$$\forall \sigma \in TTr(e), Out((iut, e) \text{ After } \sigma) \subseteq Out((s, e) \text{ After } \sigma)$$
iff
$$TTr(iut) \cap TTr(e) \subseteq TTr(s) \cap TTr(e)$$

Relativized timed conformance (2)

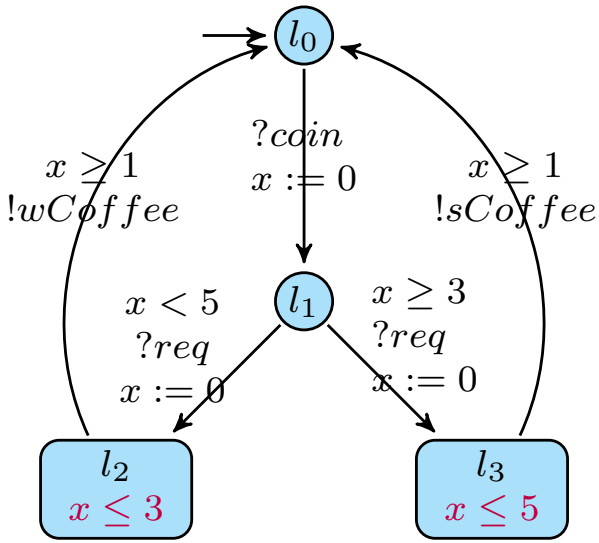
rtioco ensures Implementation has only the behavior allowed by Specification :

- 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

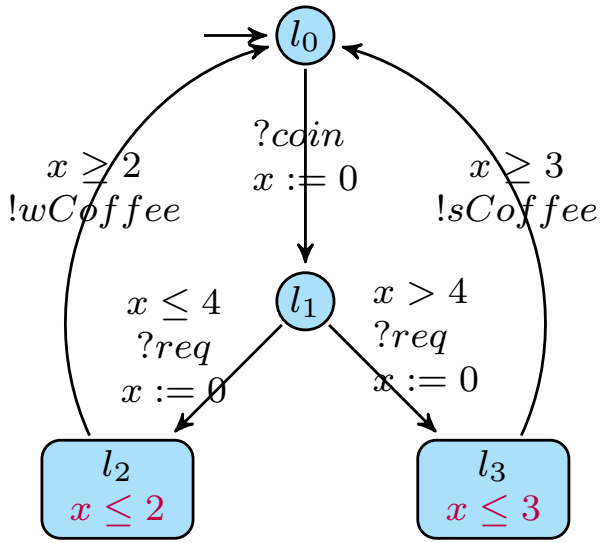
rtioco examples



Environment



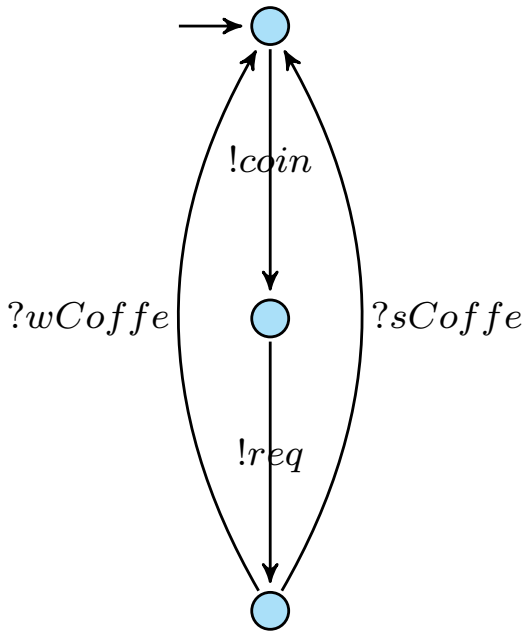
Specification s



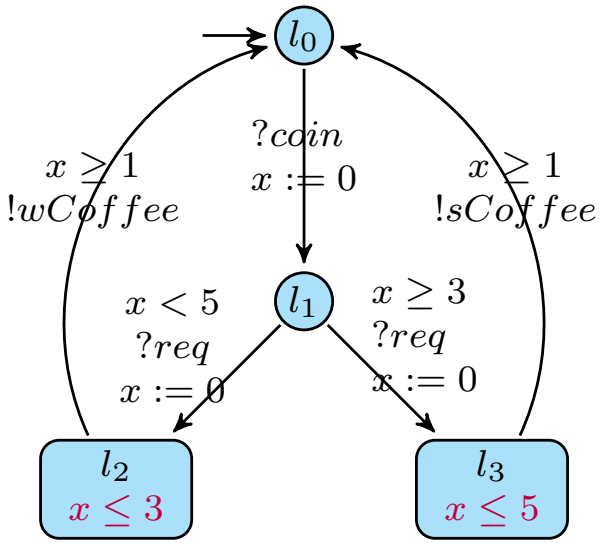
Implementation i_1

Trace σ	$Out(s \text{ After } \sigma)$	$Out(i_1 \text{ After } \sigma)$
$c \cdot 2$	$\mathbb{R}_{\geq 0}$	$\mathbb{R}_{\geq 0}$
$c \cdot 4 \cdot r \cdot 1$	$\{wCoffee, sCoffee\} \cup [0, 4]$	$[0, 1]$
$c \cdot 4 \cdot r \cdot 2$	$\{wCoffee, sCoffee\} \cup [0, 3]$	$\{wCoffee, 0\}$
$c \cdot 5 \cdot r \cdot 3$	$\{sCoffee\} \cup [0, 2]$	$\{sCoffee, 0\}$
$c \cdot 5 \cdot r \cdot 5$	$\{sCoffee, 0\}$	$\emptyset$

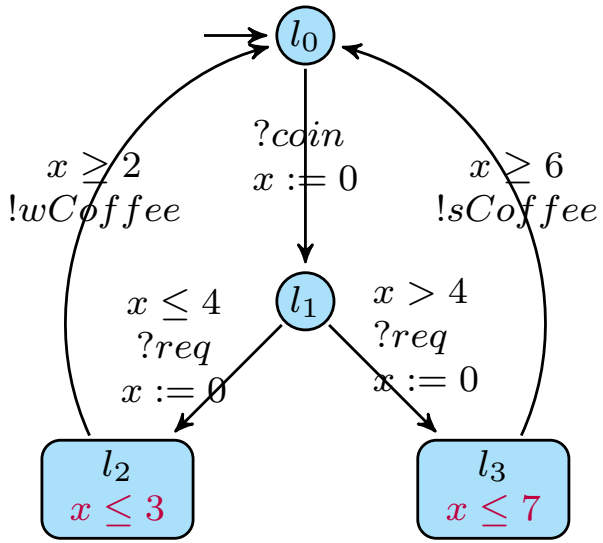
rtioco examples (2)



Environment



Specification s



Implementation i_2

Trace σ	$Out(s \text{ After } \sigma)$	$Out(i_2 \text{ After } \sigma)$
$c \cdot 2$	$\mathbb{R}_{\geq 0}$	$\mathbb{R}_{\geq 0}$
$c \cdot 4 \cdot r \cdot 1$	$\{wCoffee, sCoffee\} \cup [0, 4]$	$[0, 2]$
$c \cdot 4 \cdot r \cdot 2$	$\{wCoffee, sCoffee\} \cup [0, 3]$	$\{wCoffee\} \cup [0, 1]$
$c \cdot 5 \cdot r \cdot 3$	$\{sCoffee\} \cup [0, 2]$	$[0, 4]$
$c \cdot 5 \cdot r \cdot 5$	$\{sCoffee, 0\}$	$[0, 2]$

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

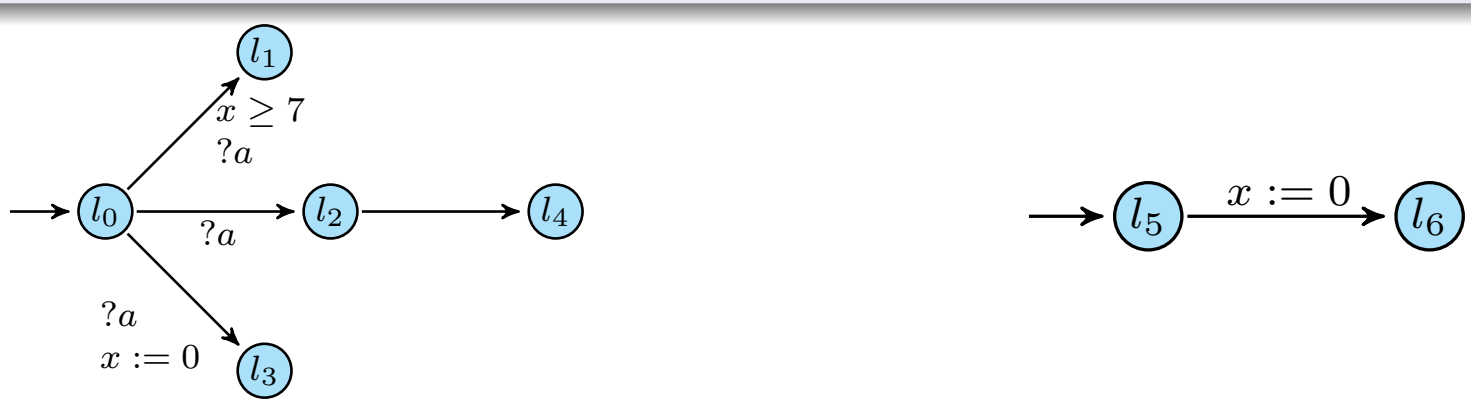
Non-determinism

Often used :

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

determinism definition

An TIOTS (S) is deterministic if
$\forall \alpha \in (A_\tau \cup \mathbb{R}_{\geq 0}), \forall s \in S, (s \xrightarrow{\alpha} s' \wedge s \xrightarrow{\alpha} s'') \Rightarrow s' = s''$.

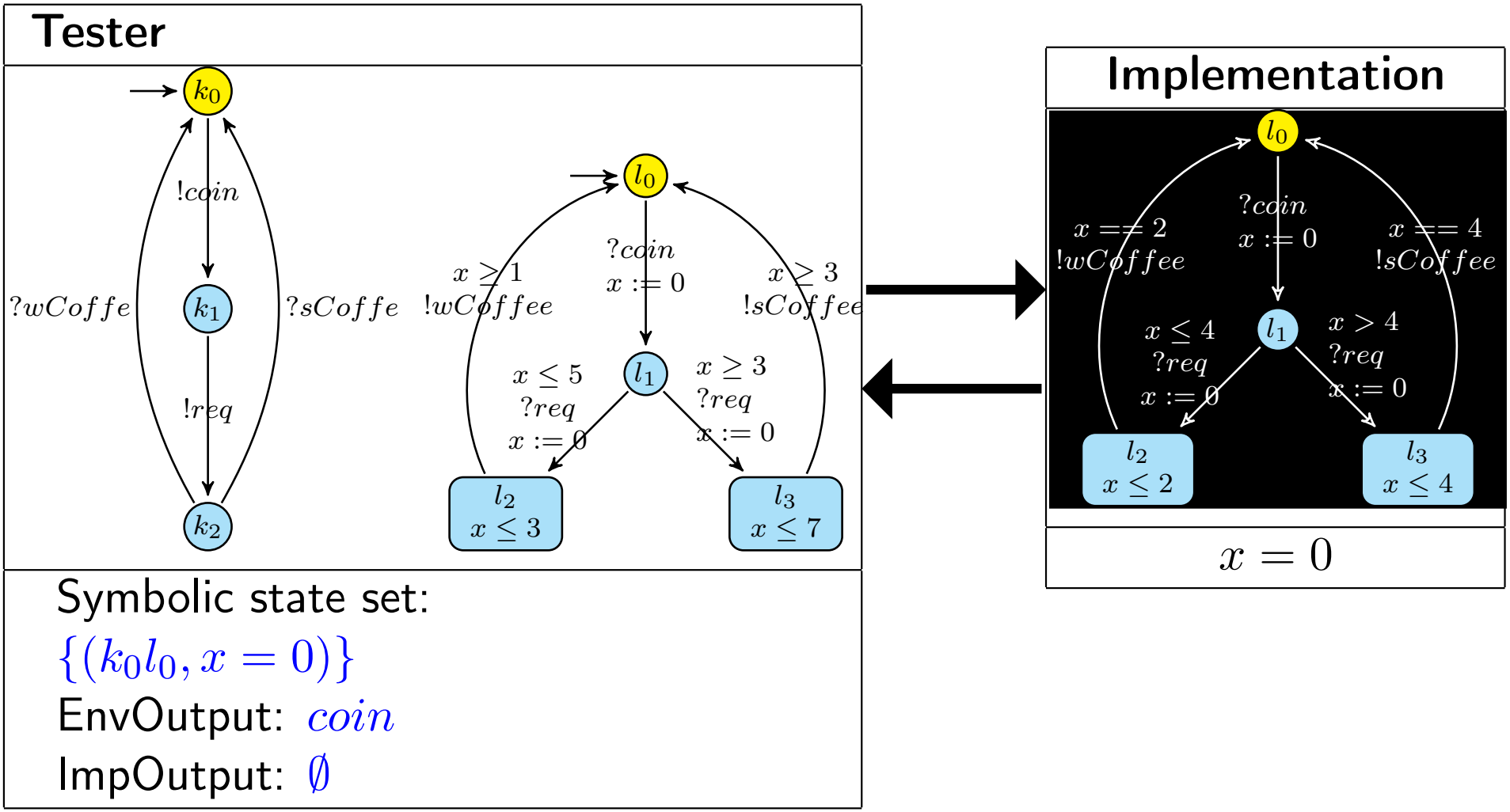


$(l_0, x = 3)$ After $a = \{(l_2, x = 3), (l_4, x = 3), (l3, x = 0)\}$
$(l_5, x = 0)$ After $4 = \{(l_5, x = 4), (l_6, 0 \leq x \leq 4)\}$

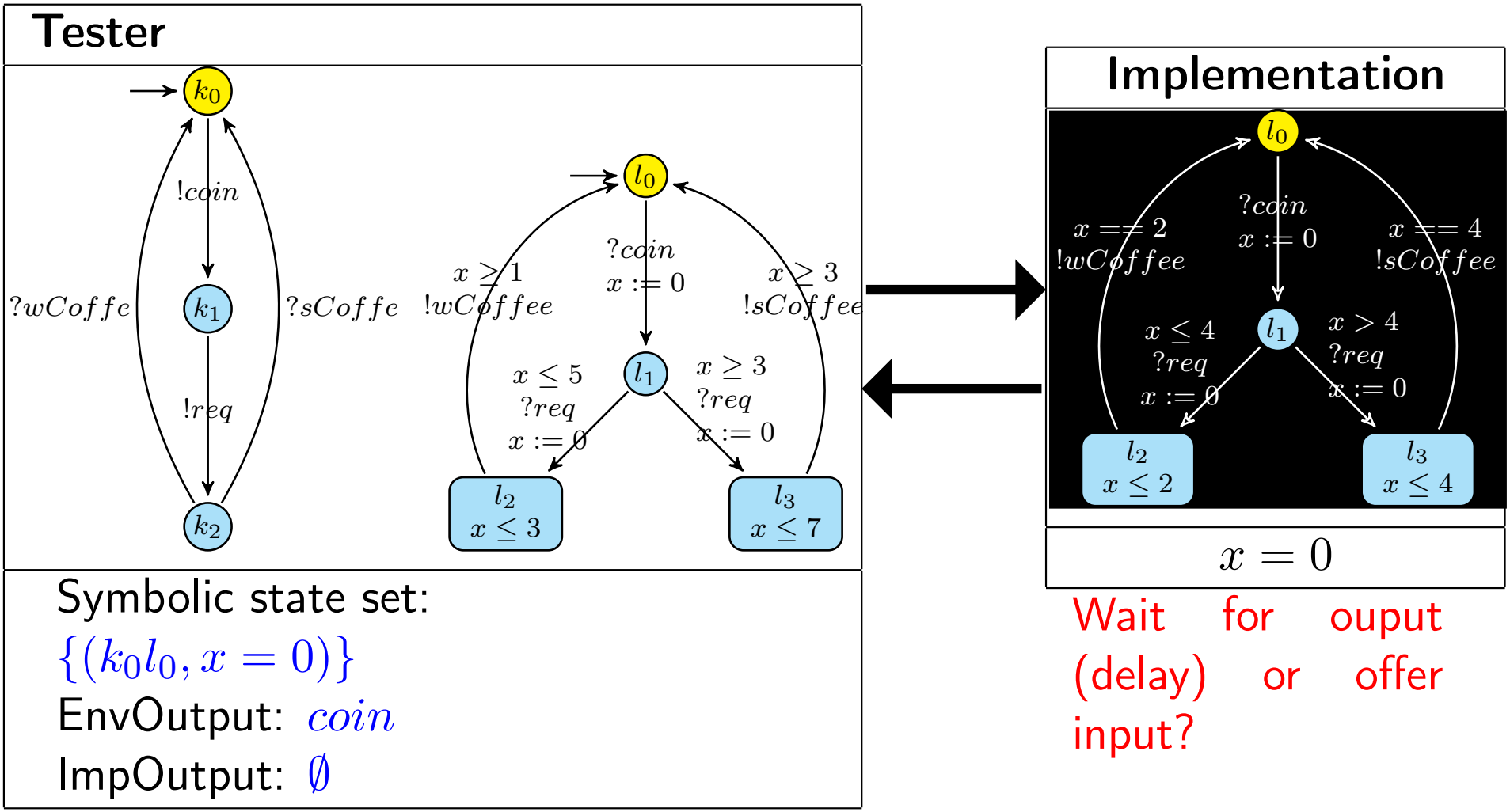
Uppaal TRON algorithm $TestGenExe(\mathcal{S}, \mathcal{E}, IUT, T)$

while $\mathcal{Z} \neq \emptyset \wedge \sharp iterations \leq T$ **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 ;
        $\mathcal{Z} := \mathcal{Z}$ 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**
        $\mathcal{Z} := \mathcal{Z}$ After d' ;
        if $o \notin ImpOutput(\mathcal{Z})$ **then** **return** *FAIL* **else**
        $\mathcal{Z} := \mathcal{Z}$ After o
      else
        $\mathcal{Z} := \mathcal{Z}$ 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*

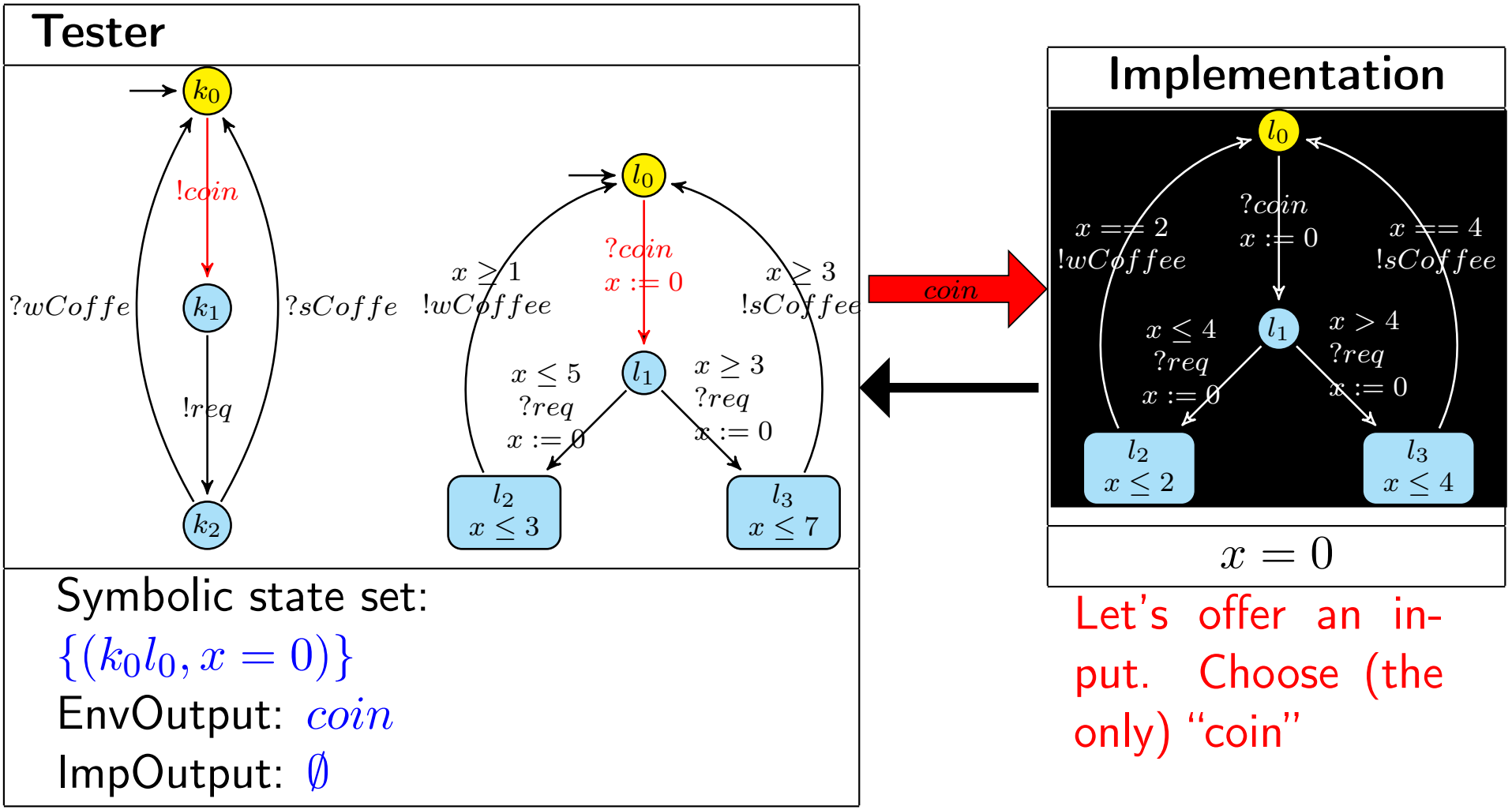
Example of test execution



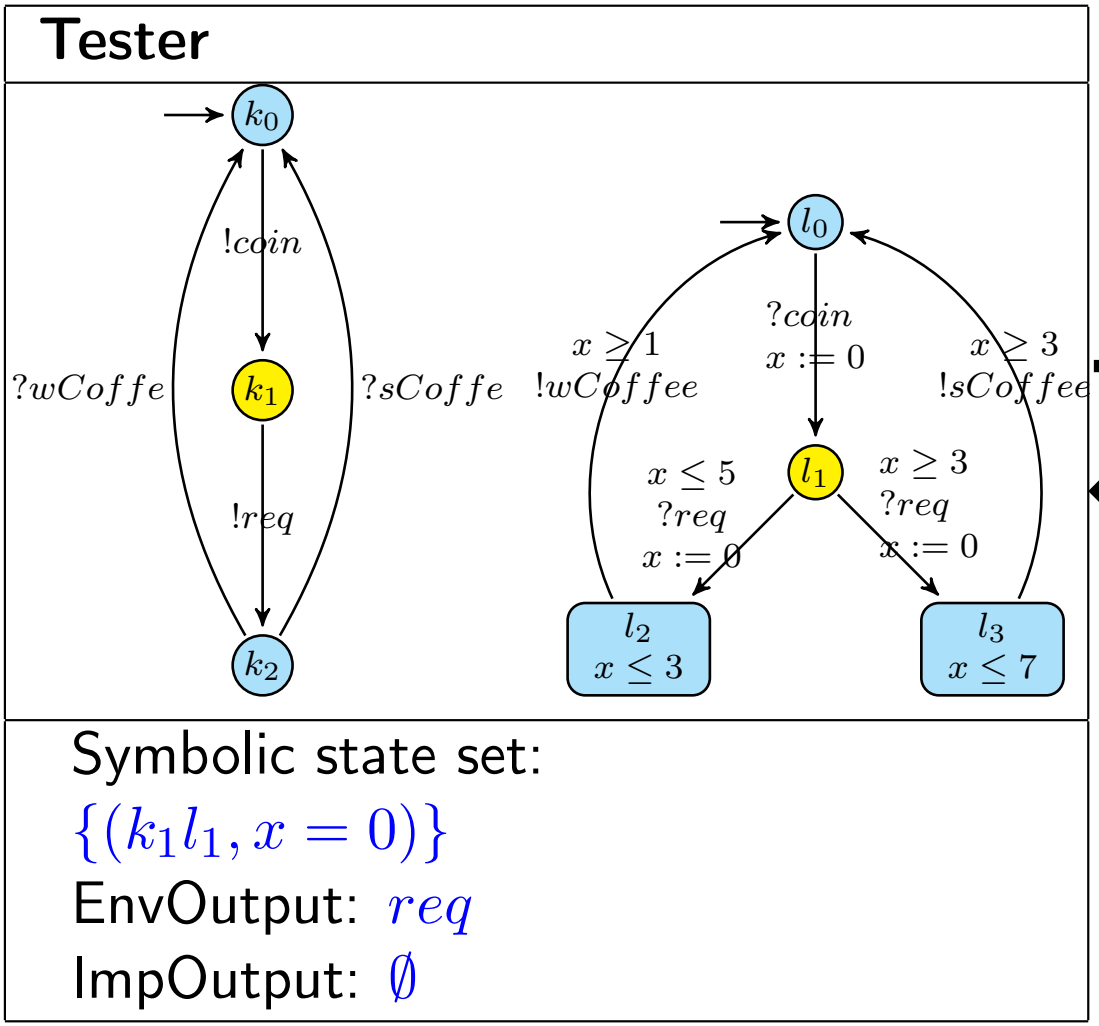
Example of test execution

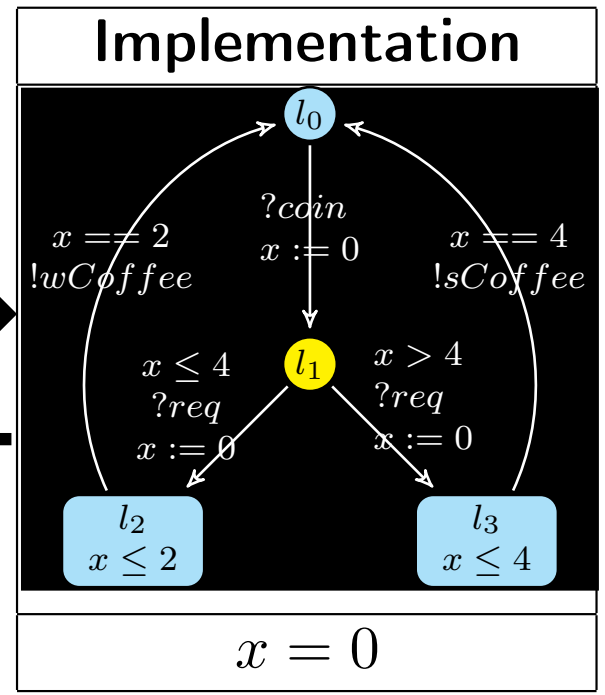


Example of test execution



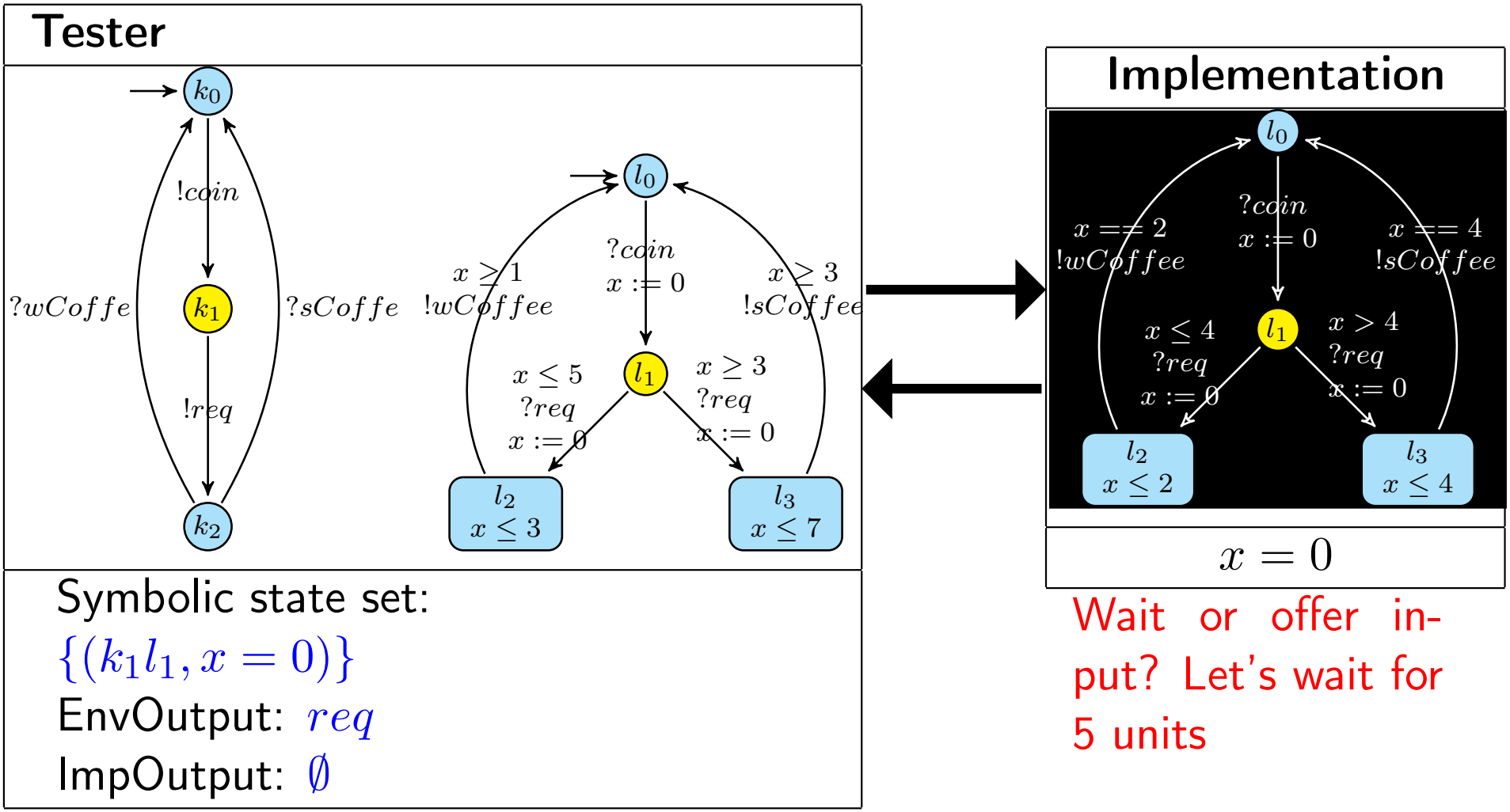
Example of test execution





Update the state set and other variables

Example of test execution



Symbolic state set:

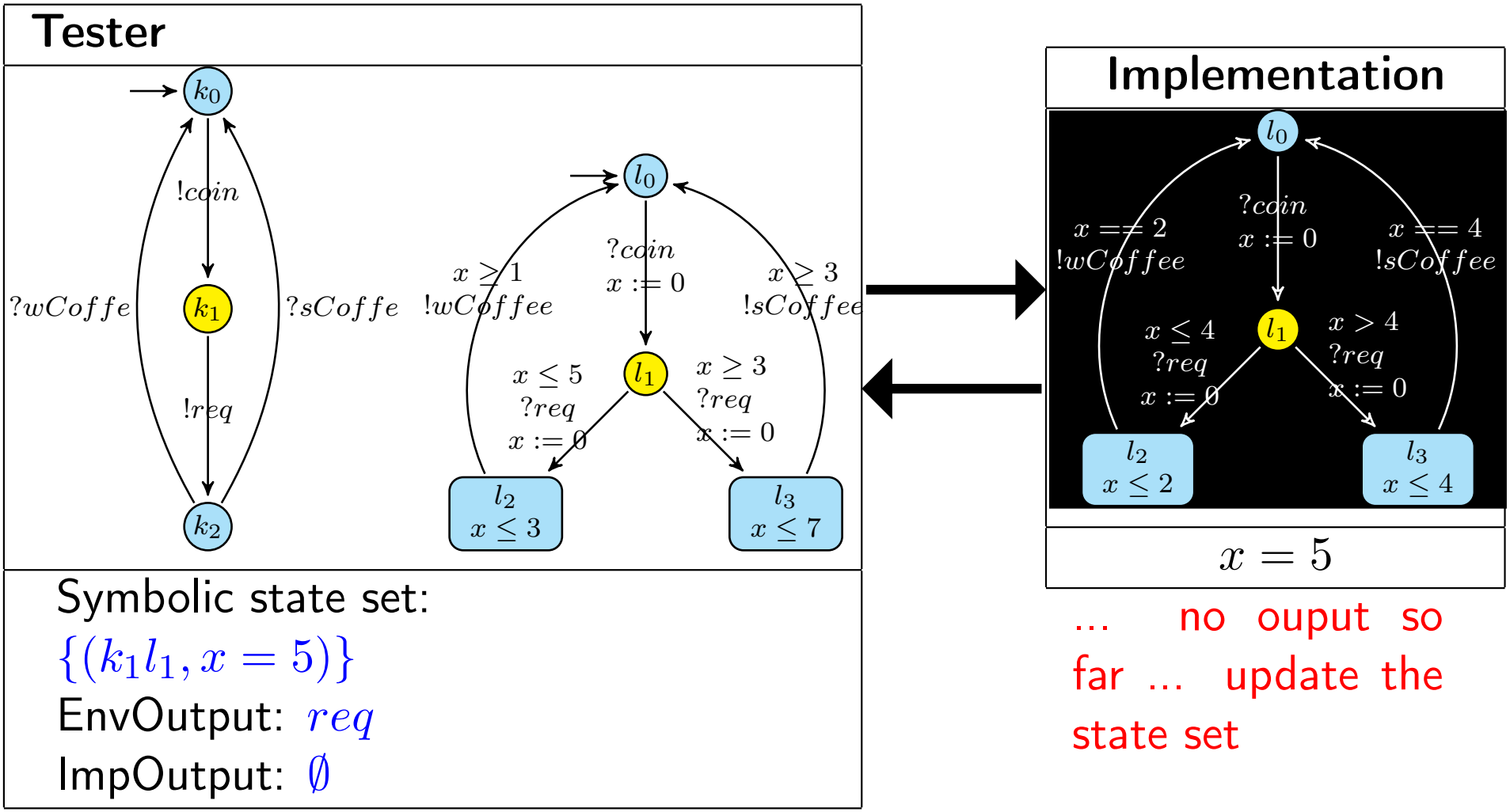
$\{(k_1 l_1, x = 0)\}$

EnvOutput: req

ImpOutput: $\emptyset$

Wait or offer input? Let's wait for 5 units

Example of test execution



Symbolic state set:

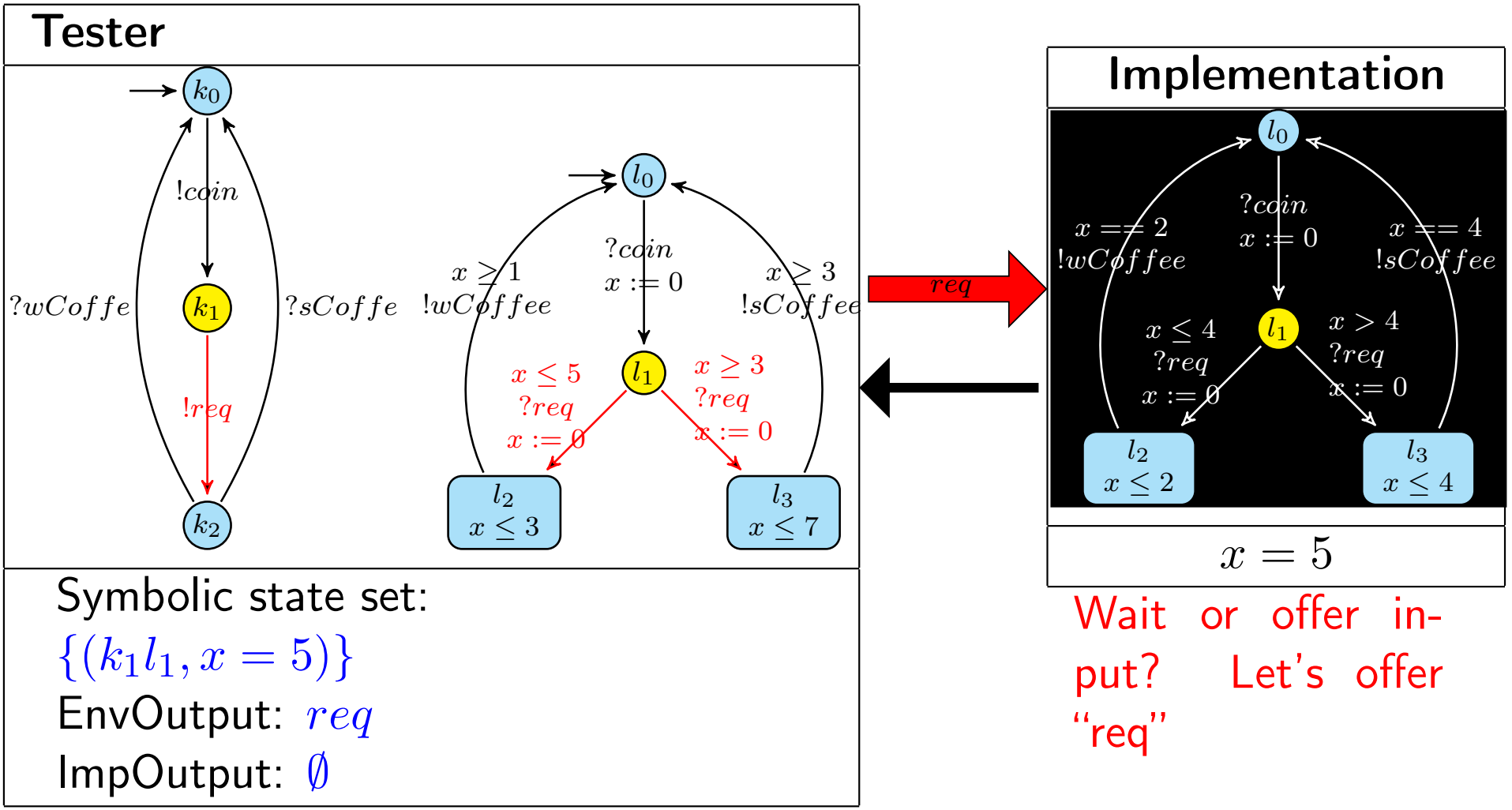
$\{(k_1 l_1, x = 5)\}$

EnvOutput: req

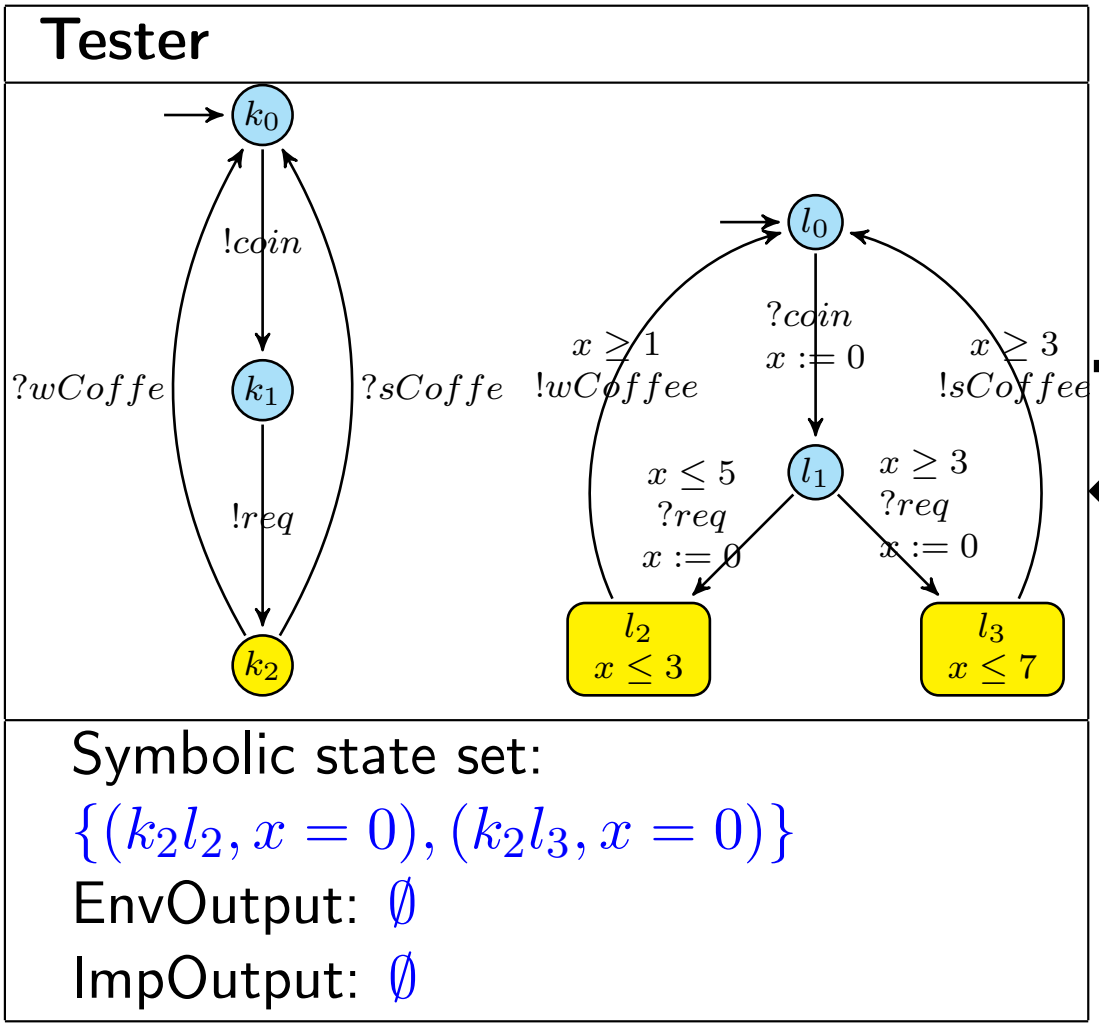
ImpOutput: $\emptyset$

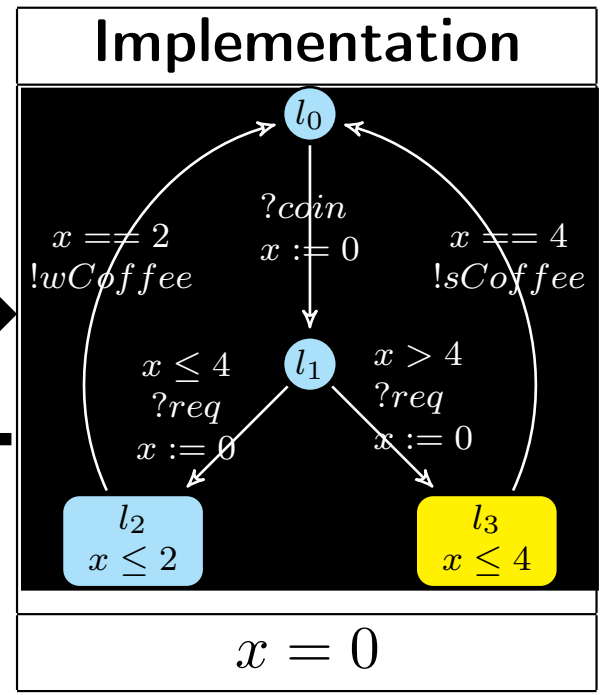
... no ouput so far ... update the state set

Example of test execution



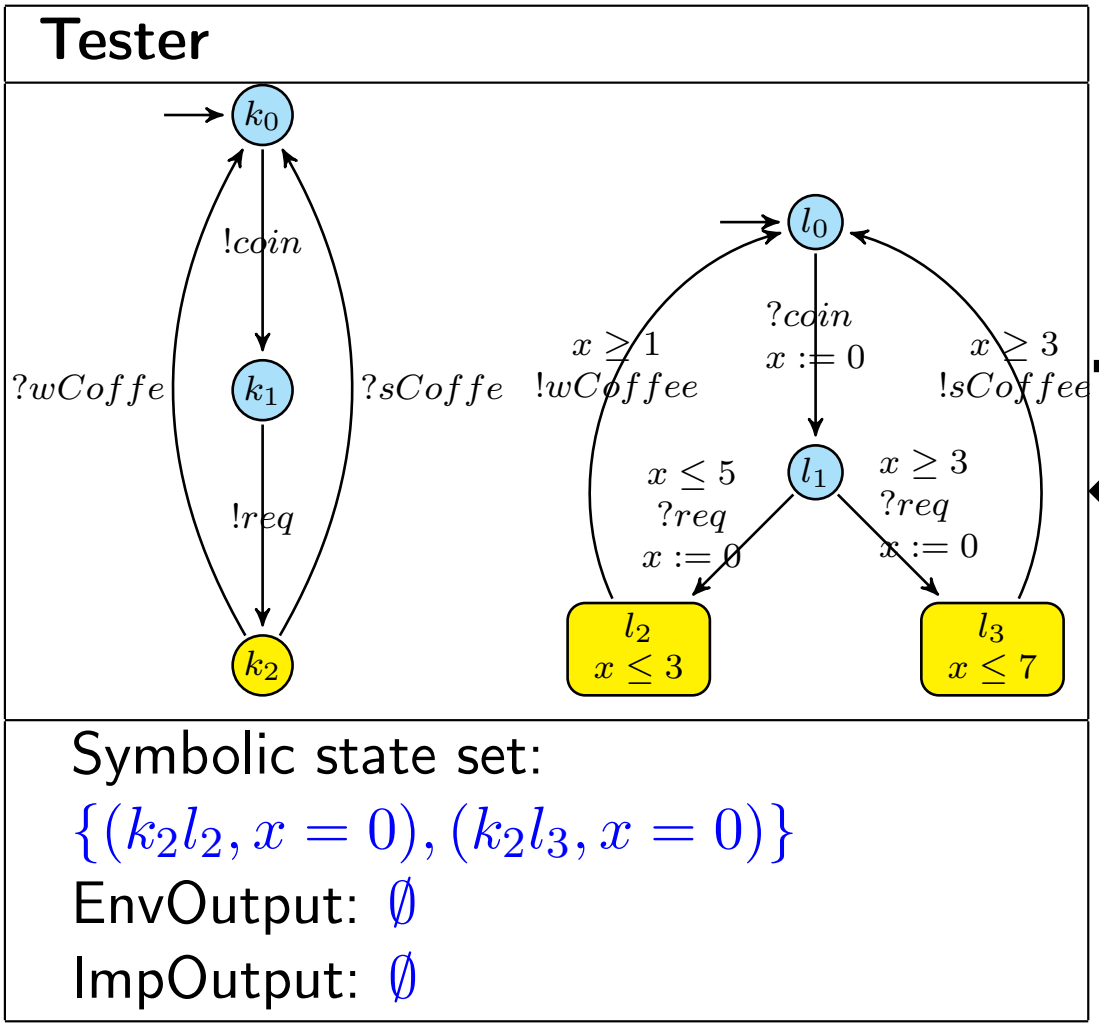
Example of test execution

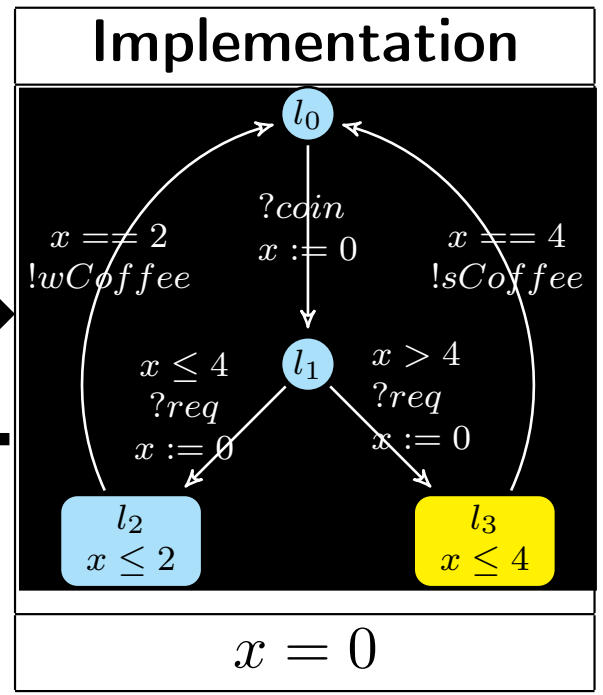




Update the state set and other variables

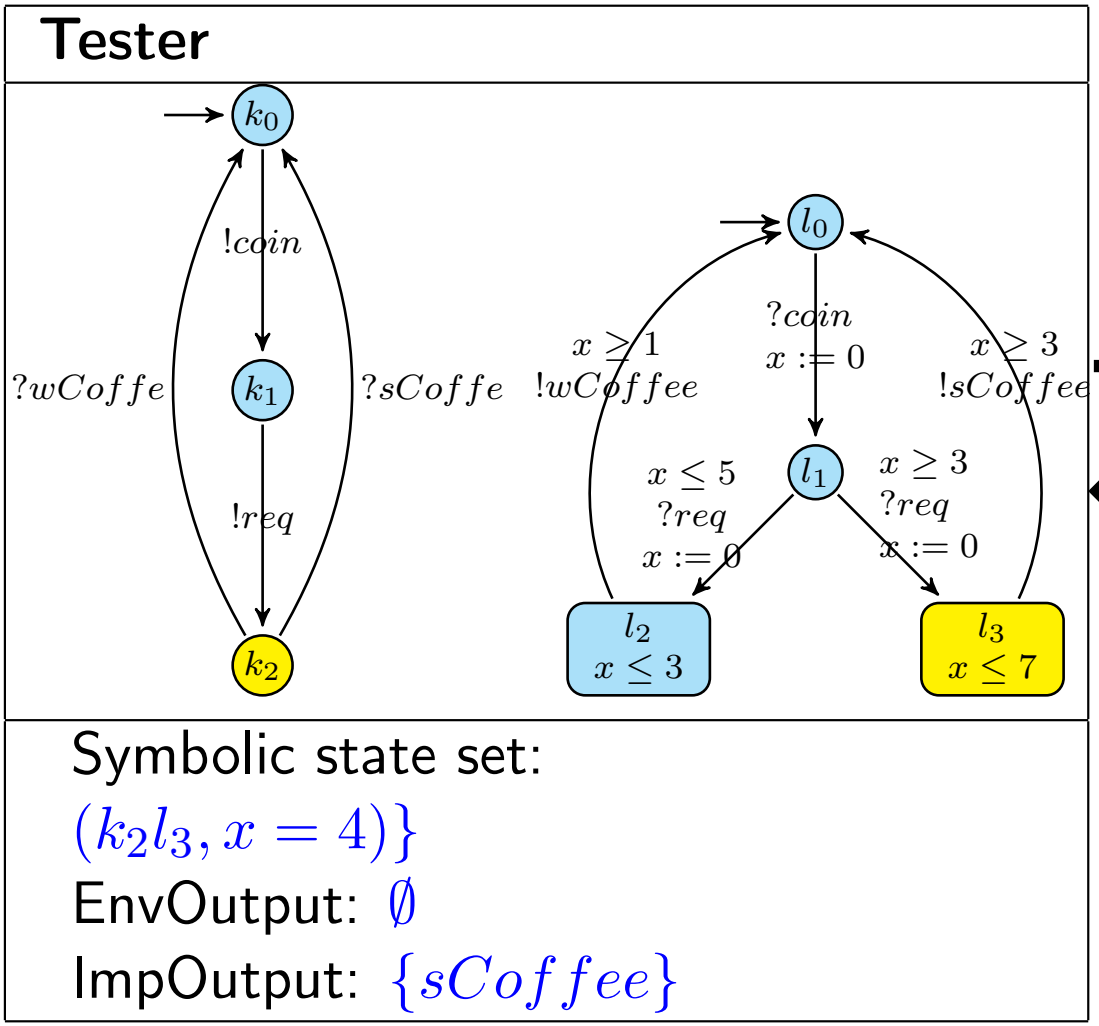
Example of test execution

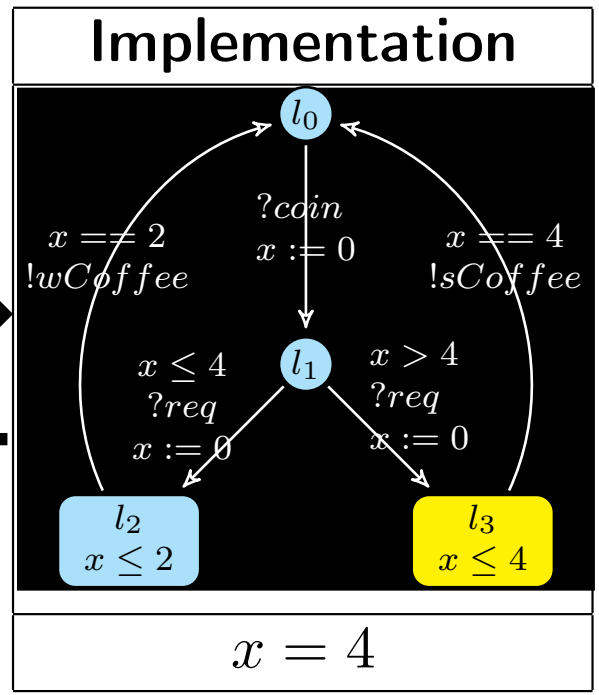




Wait or offer input? Let's wait for 4 units

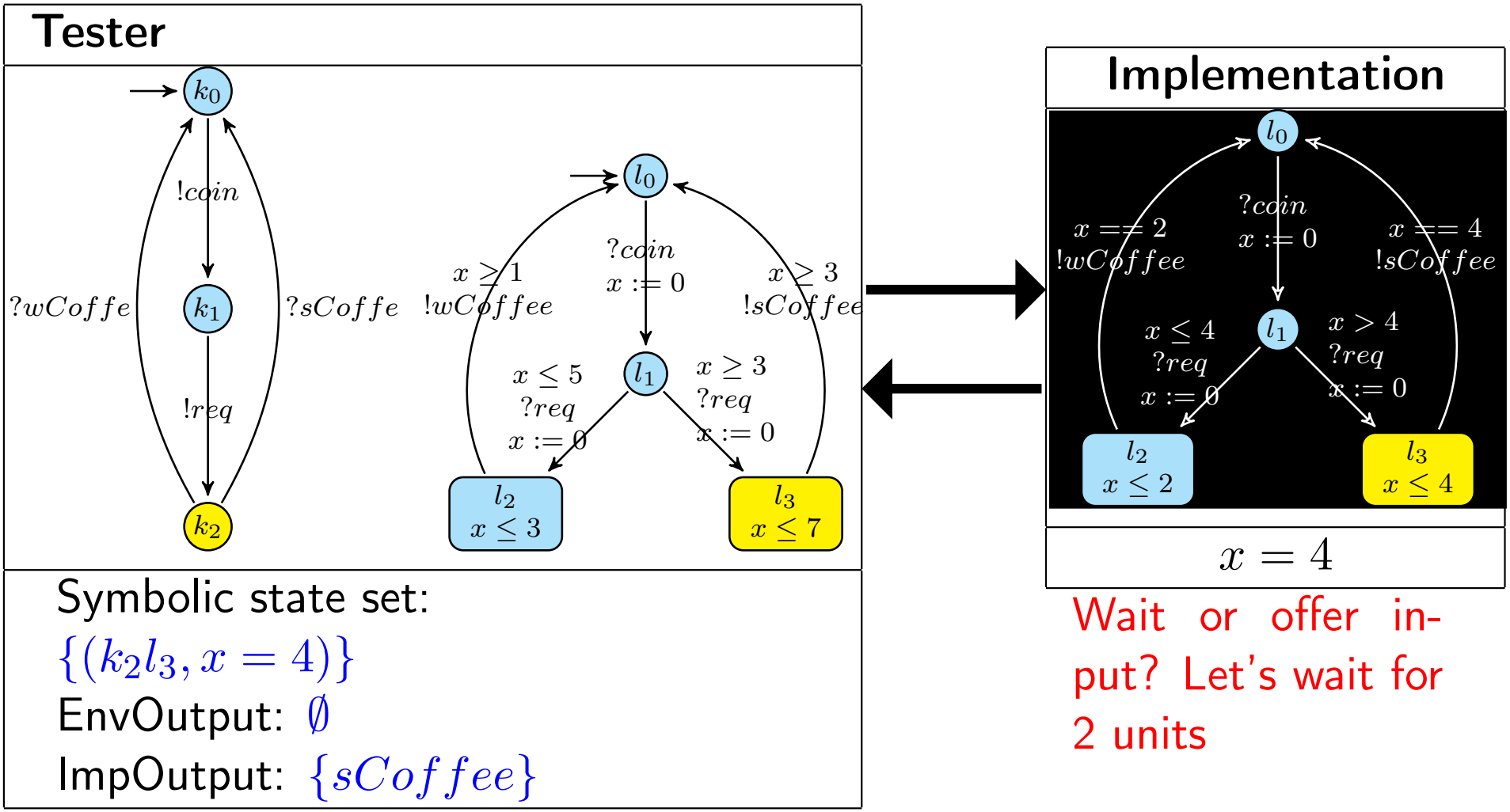
Example of test execution





... no output so far: update the state set

Example of test execution



Symbolic state set:

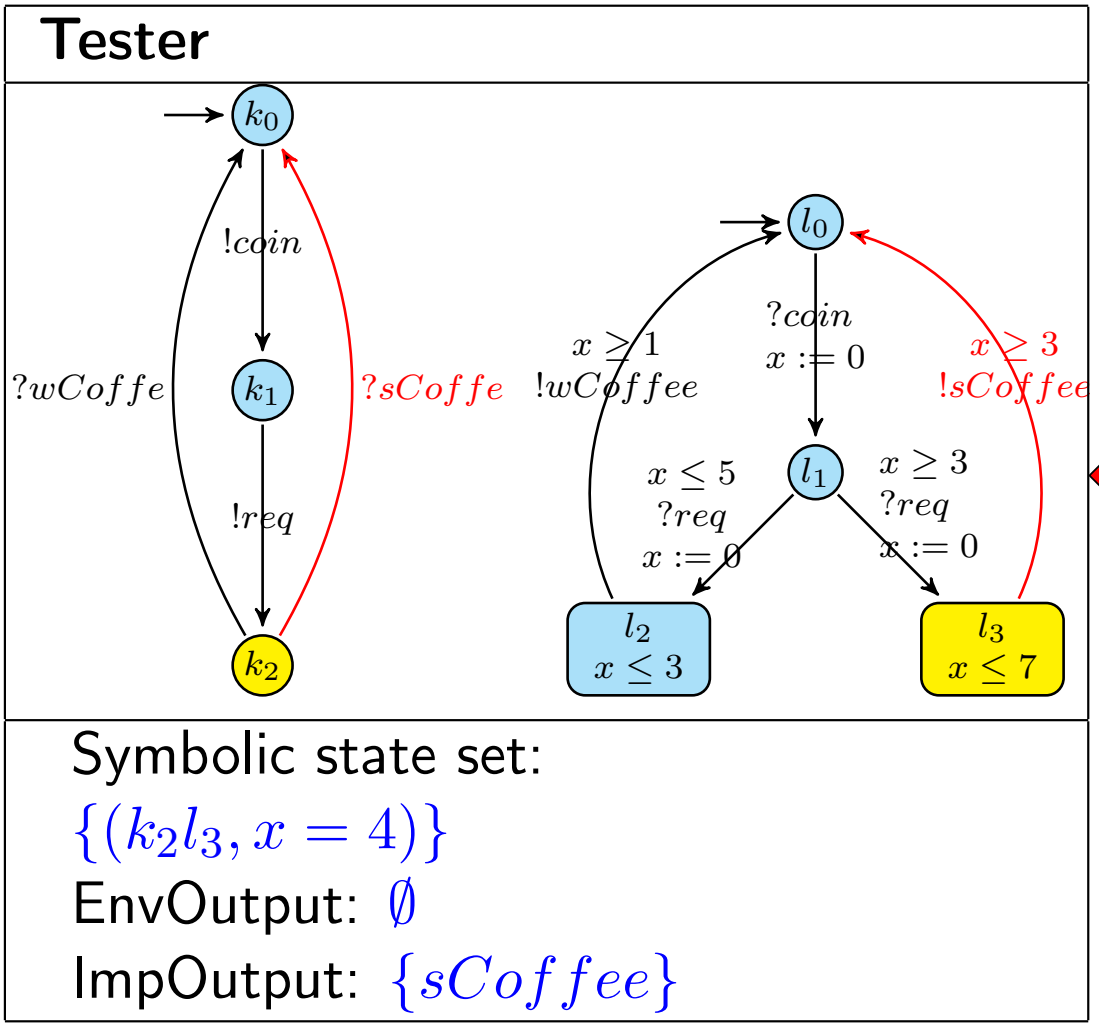
$\{(k_2 l_3, x = 4)\}$

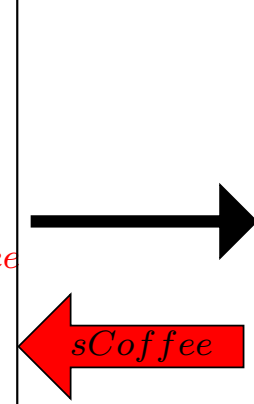
EnvOutput: $\emptyset$

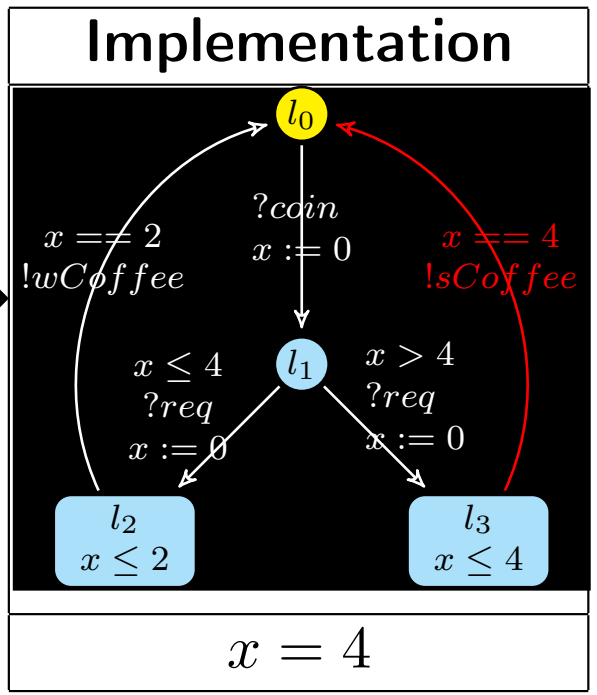
ImpOutput: $\{sCoffee\}$

Wait or offer input? Let's wait for 2 units

Example of test execution

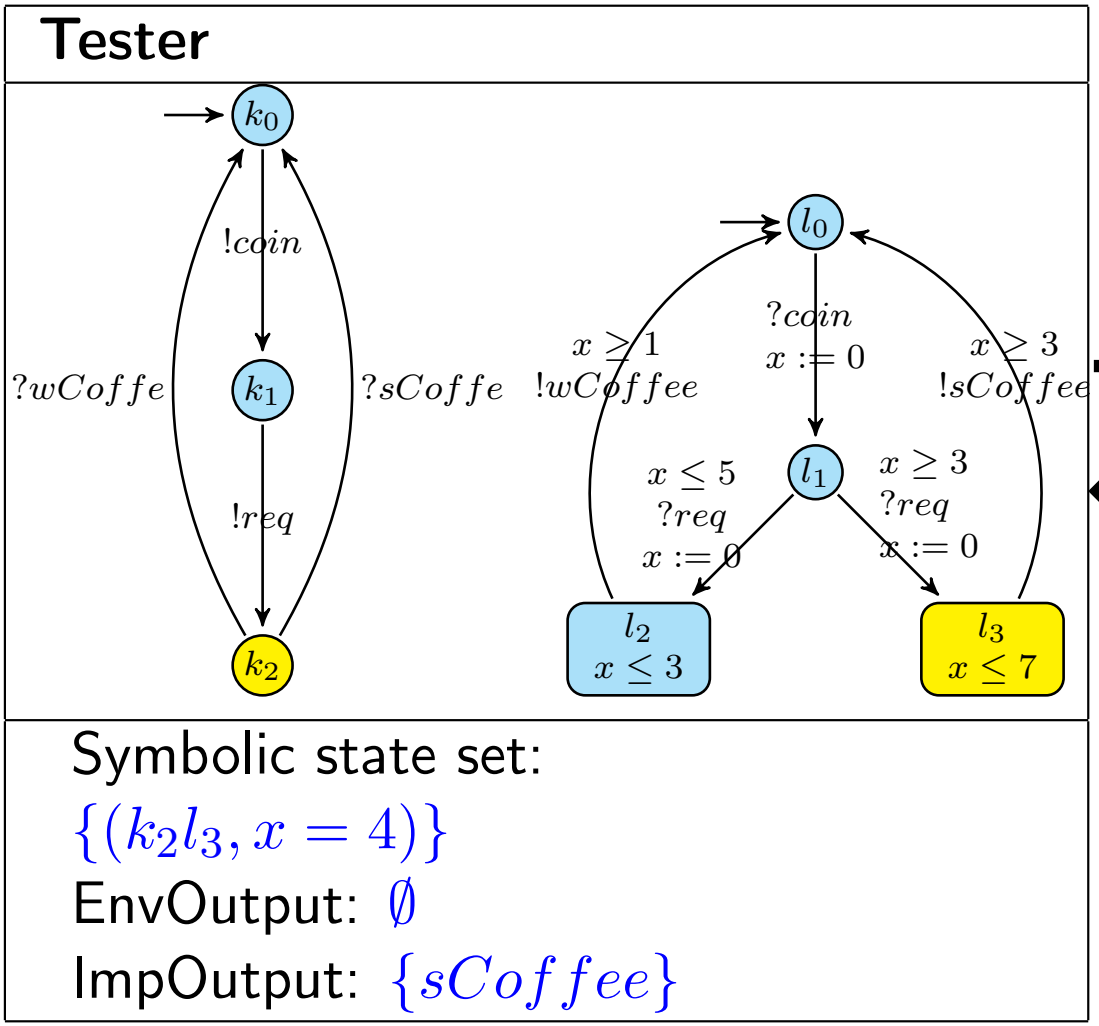


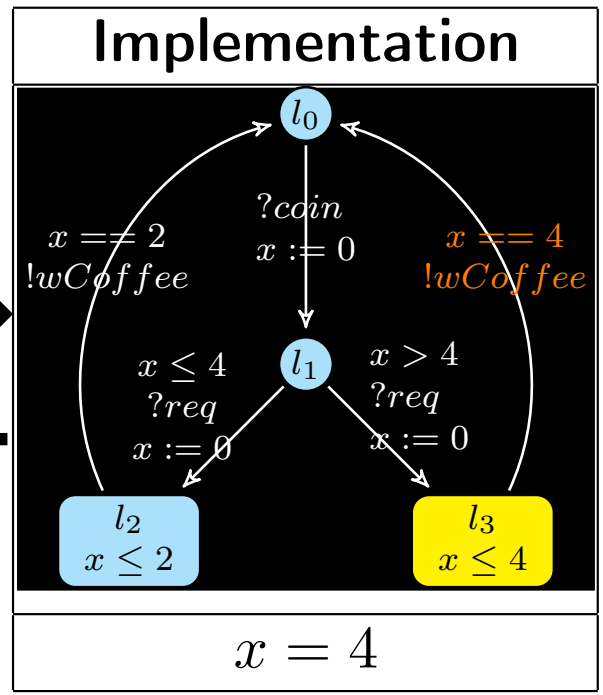




Got output after 0 delay: update the state set

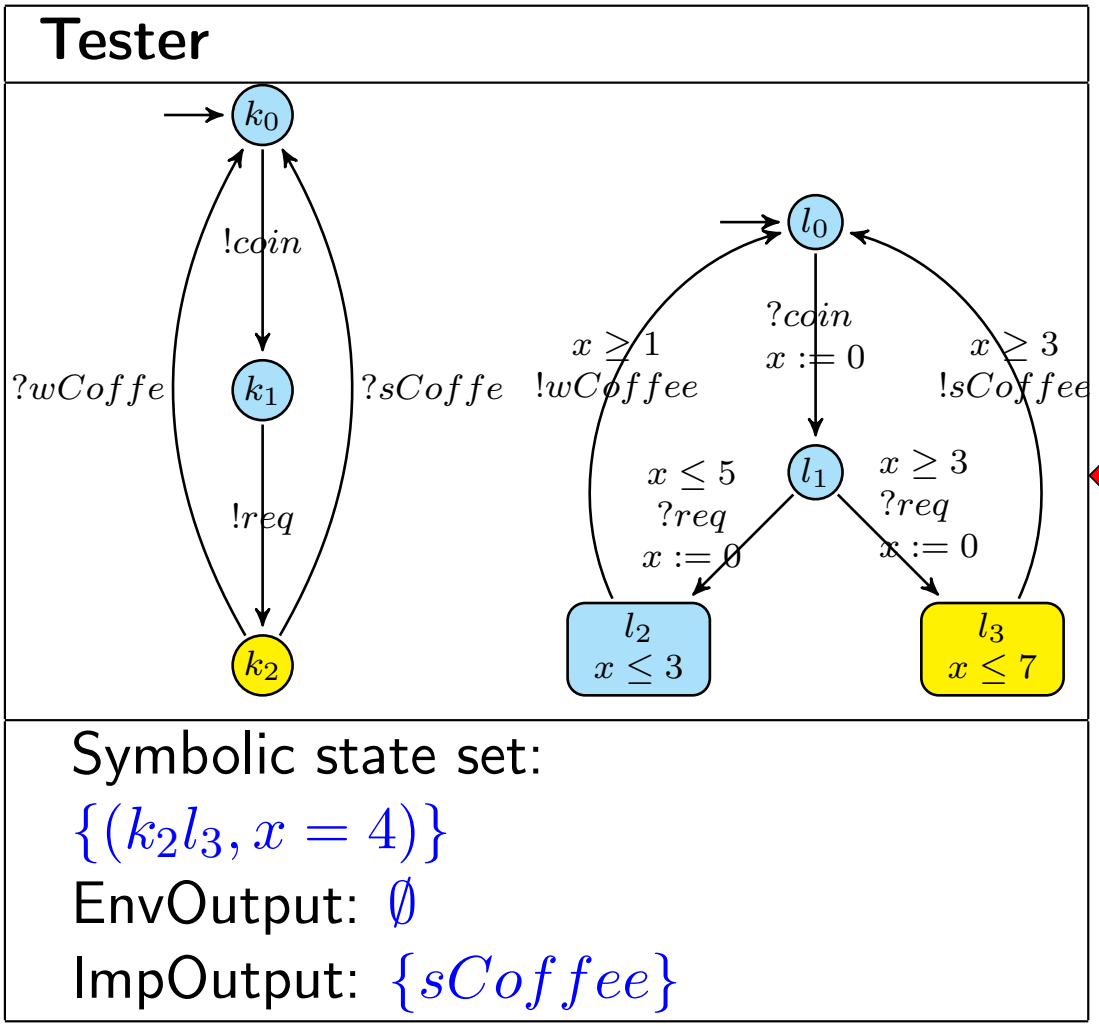
Example of test execution

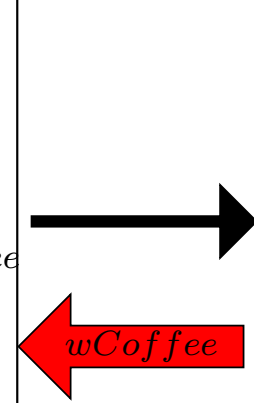


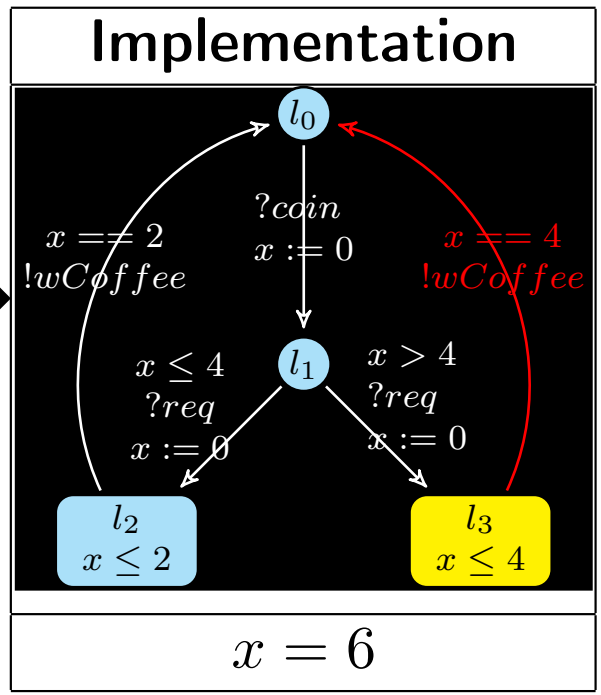


What if there is a bug? Let's wait back for 2 units

Example of test execution







output after 0 delay: $wCoffee \notin \{sCoffee\}$

Properties of test cases

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

Soundness, Exhaustiveness

- Soundness :
  $TestGenExe(\mathcal{S}, \mathcal{E}, \mathcal{IUT}, T) = \mathit{Fail} \Rightarrow \neg(\mathcal{IUT}\ \mathbf{rtioco}_{\mathcal{E}}\ \mathcal{S})$
- Exhaustiveness :
  $\neg(\mathcal{IUT}\ \mathbf{rtioco}_{\mathcal{E}}\ \mathcal{S}) \Rightarrow \mathrm{Prob}(TestGenExe(\mathcal{S}, \mathcal{E}, \mathcal{IUT}, T) = \mathit{Fail}) \overset{T \to \infty}{\longrightarrow} 1$

If $\mathcal{IUT}$ 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 $\Rightarrow$ 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 $\Rightarrow$ 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

-
- 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) $\rightarrow$ 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 $\mathcal{E} - component$ (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

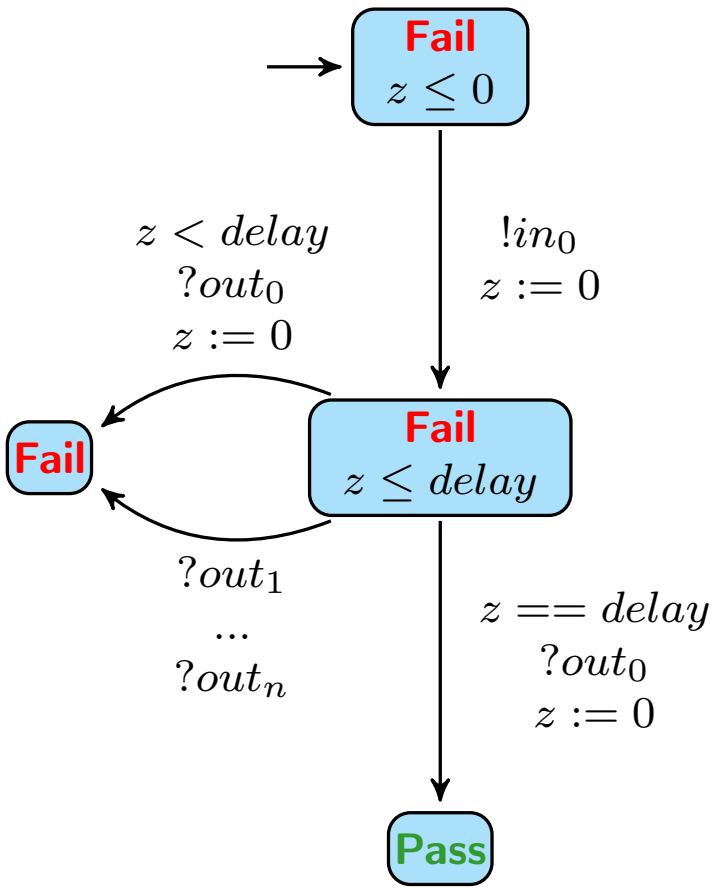
Idea

- Formulate the problem as safety property (usually solved by a reachability analysis) $\rightarrow$ 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 $\mathcal{E} - component$ (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 :

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)

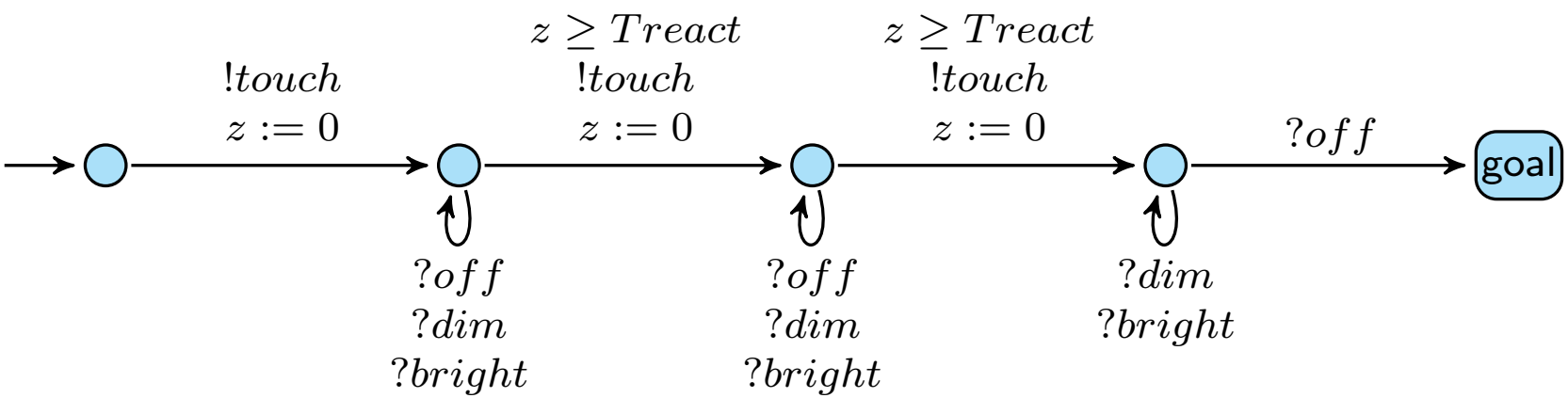
TP1 : Check that the light can become bright :
Simple reachability property : eventually the system specification can enter location **BRIGHT**

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

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 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

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

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

Outline

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