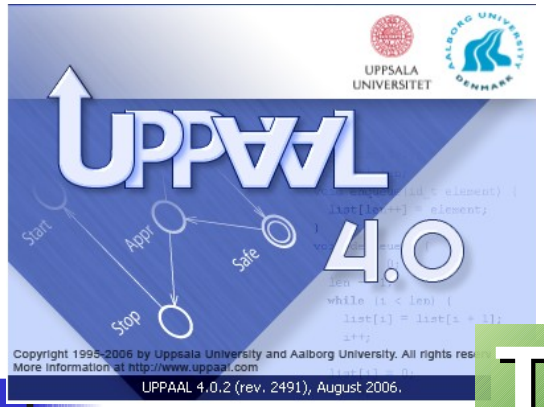


TRON

Real Time Testing

using UPPAAL

Marius Mikučionis with

Kim G. Larsen, Brian Nielsen, Shuhao Li,
Arne Skou, Anders Hessel, Paul Pettersson

AALBORG UNIVERSITY DENMARK

BRICS
Basic Research
in Computer Science

Center for Indlejrede Software Systemer

Overview

- Basic Concepts about Testing
- Conformance for Real-Time Systems

- Off-line Test Generation
  - Controllable Timed Automata
  - Observable Timed Automata

CLASSIC CORA TIGA

- On-line Test Generation

TRON

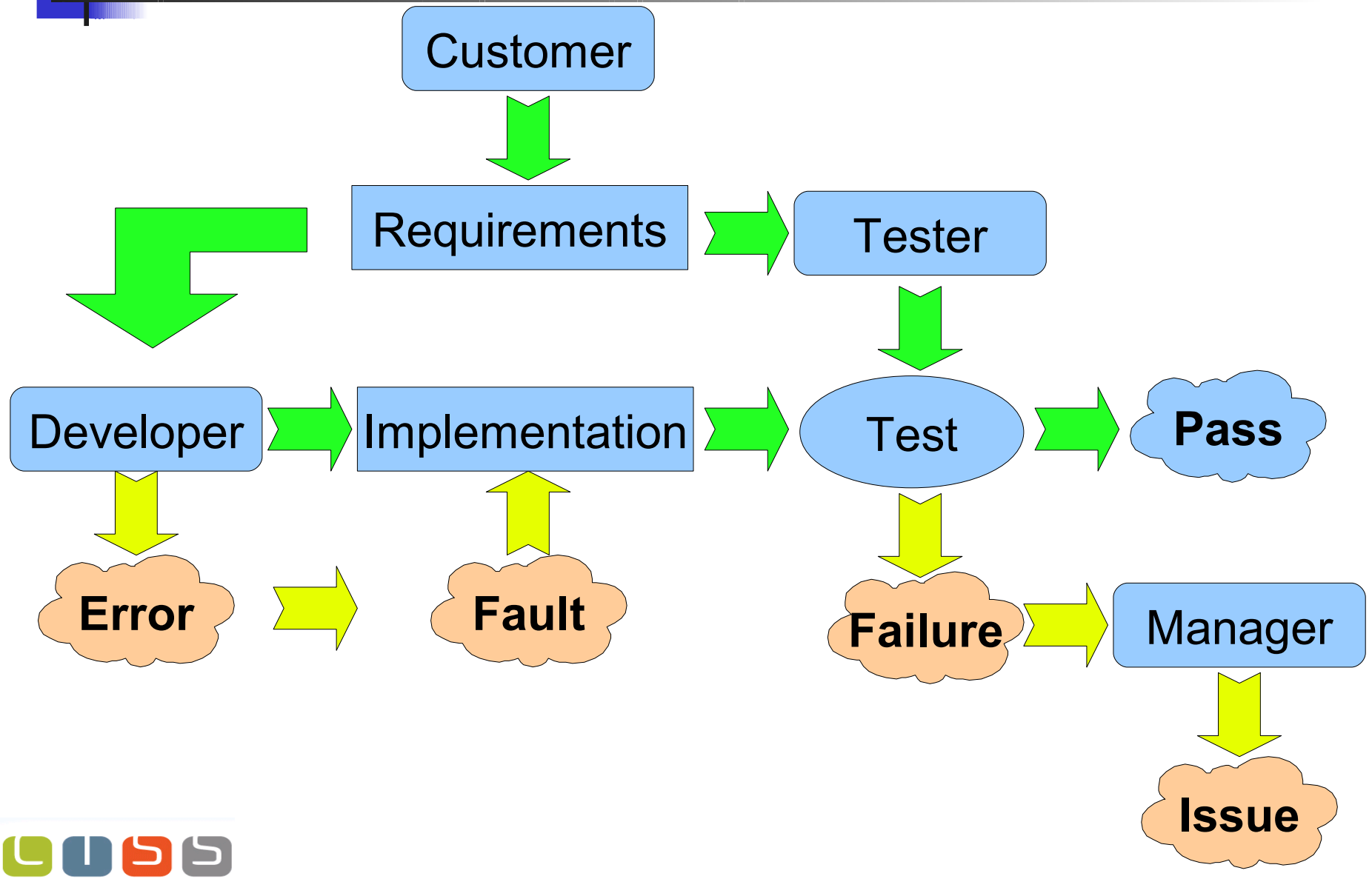
- Uppaal TRON tutorial

Testing is...

- an experiment (empyrical activity)
- with a real object (implementation under test or IUT)
- with a goal of determining it's qualities

- ... (almost) never complete:
  - Tests can only show presence of faults but never their absence (E. Dijkstra)
- ... complementary to formal verification:
  - Formal proofs deal with abstractions and leave out details.
- Types of testing:
  - White box, black box, functional, non-functional...
- Levels of testing:
  - Function, component, module, integration, system

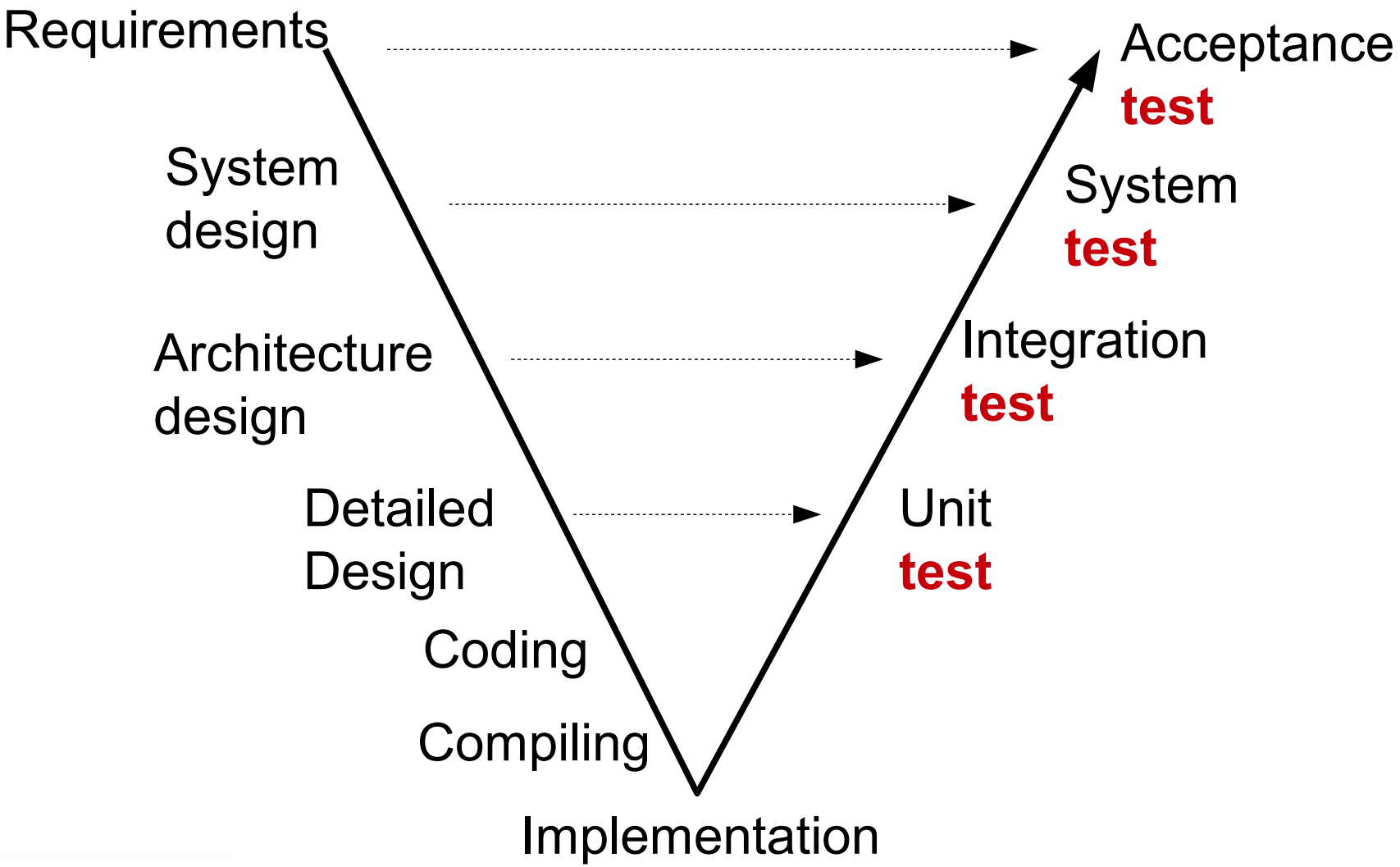
Error, Fault, Failure, Issue



Testing in practice...

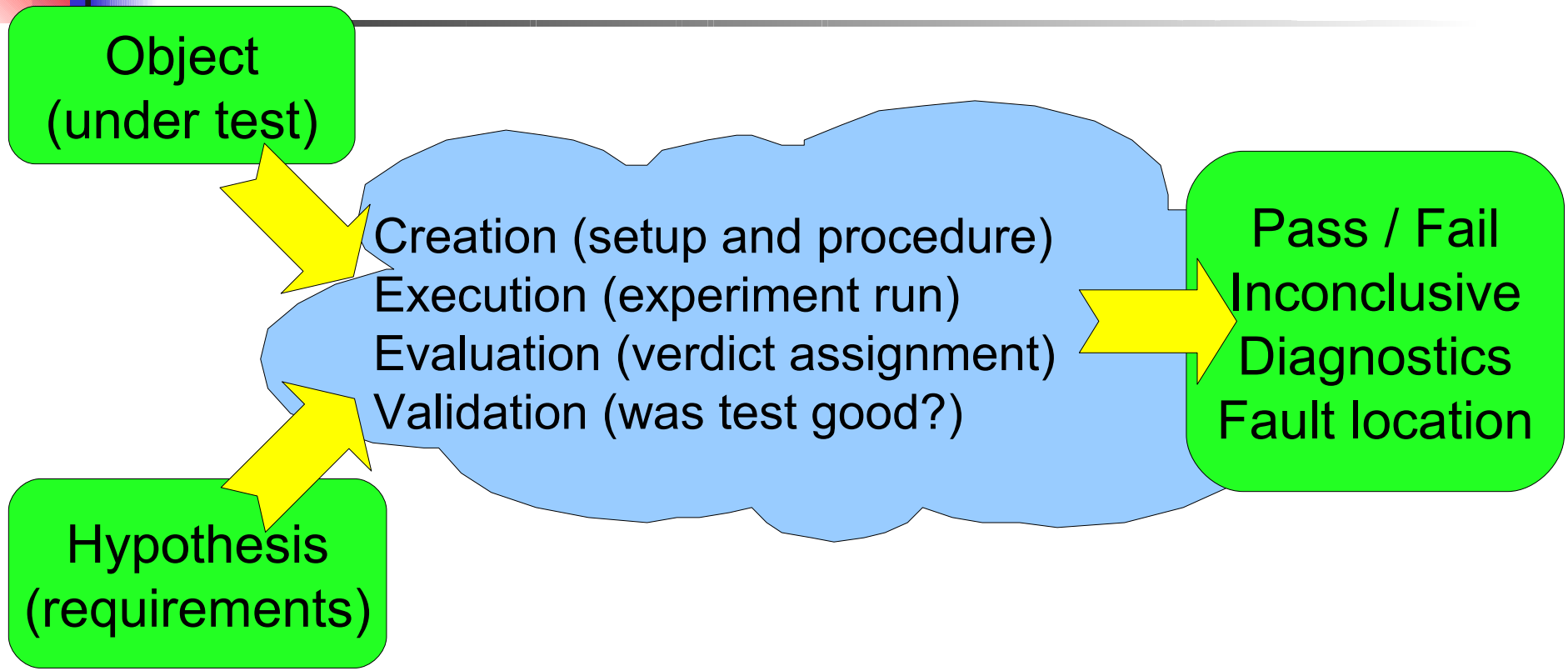
- Primary validation technique used in industry
  - In general avg. 10-20 errors per 1000 LOC
  - 30-50 % of development time and cost in embedded software
- Used to:
  - find errors
  - determine risk of release
- Part of system development life-cycle
- Expensive: tedious, error prone, time consuming

V development model



Center for Indlejrede Software Systemer

Testing Activities



- Usually one testing solutions covers just a few activities, e.g.:
  - Creation may encode verdict based on possible executions
  - Execution may result in coverage profile for validation
  - Execution and evaluation result in monitoring (passive testing)
  - Validation may require executing and evaluating tests on faulty objects

Current development trends

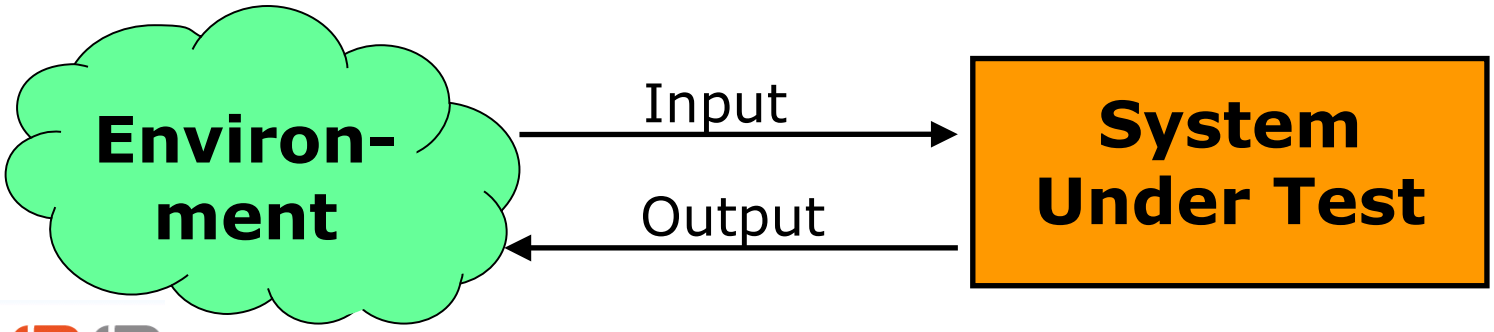
- The key to success is automation
- Automation means implementable tools:
  - elementary steps, precise, unambiguous, terminating...
- Answer: (formal-)model-based development:
  - Requirement specification is a model
  - Test generation from a model
  - Test execution concurrently with a model
  - Test evaluation against a model
  - Test validation by decorating a model/IUT

Uppaal as Modeling Engine

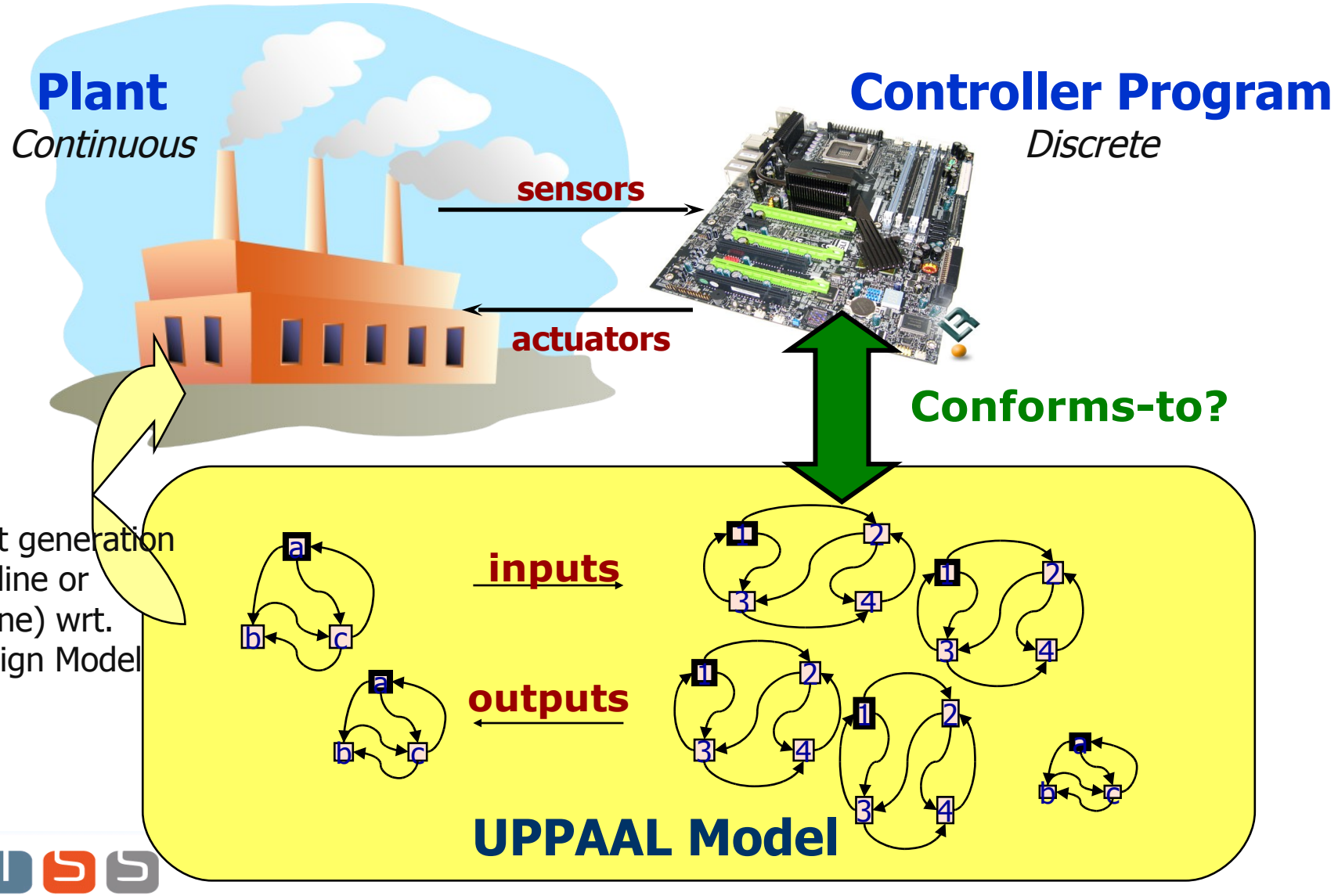
- Timed automata are:
  - Well defined: unambiguous, formal semantics
  - Expressive: cover large class of systems
  - Simple: most analysis are decidable
  - Abstract: allows almost any level of detail
- Uppaal provides
  - Practical language extensions
  - Fast algorithms
  - Compact data structures

Conformance Testing

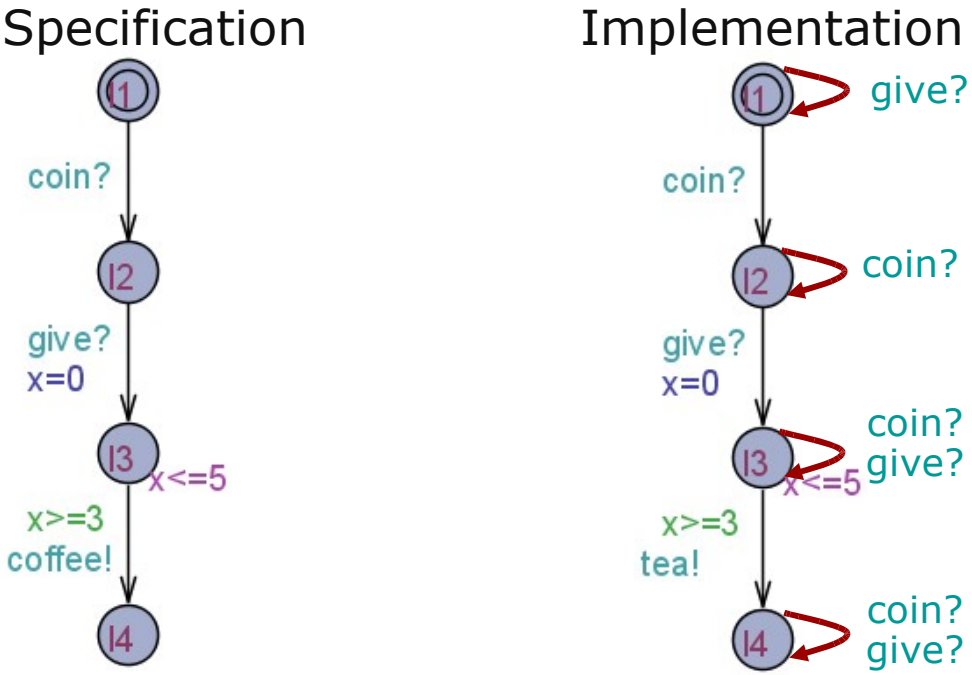
- Correctness is defined via conformance relation
- Mostly system level: isolation is important
- Black-box: state is not observable directly
- Functional: value of response should be correct
- Behavioral: input/output sequences
- Real-time: timing is as important as computed value
- Environment assumptions are explicit



Real-time Model-Based Testing



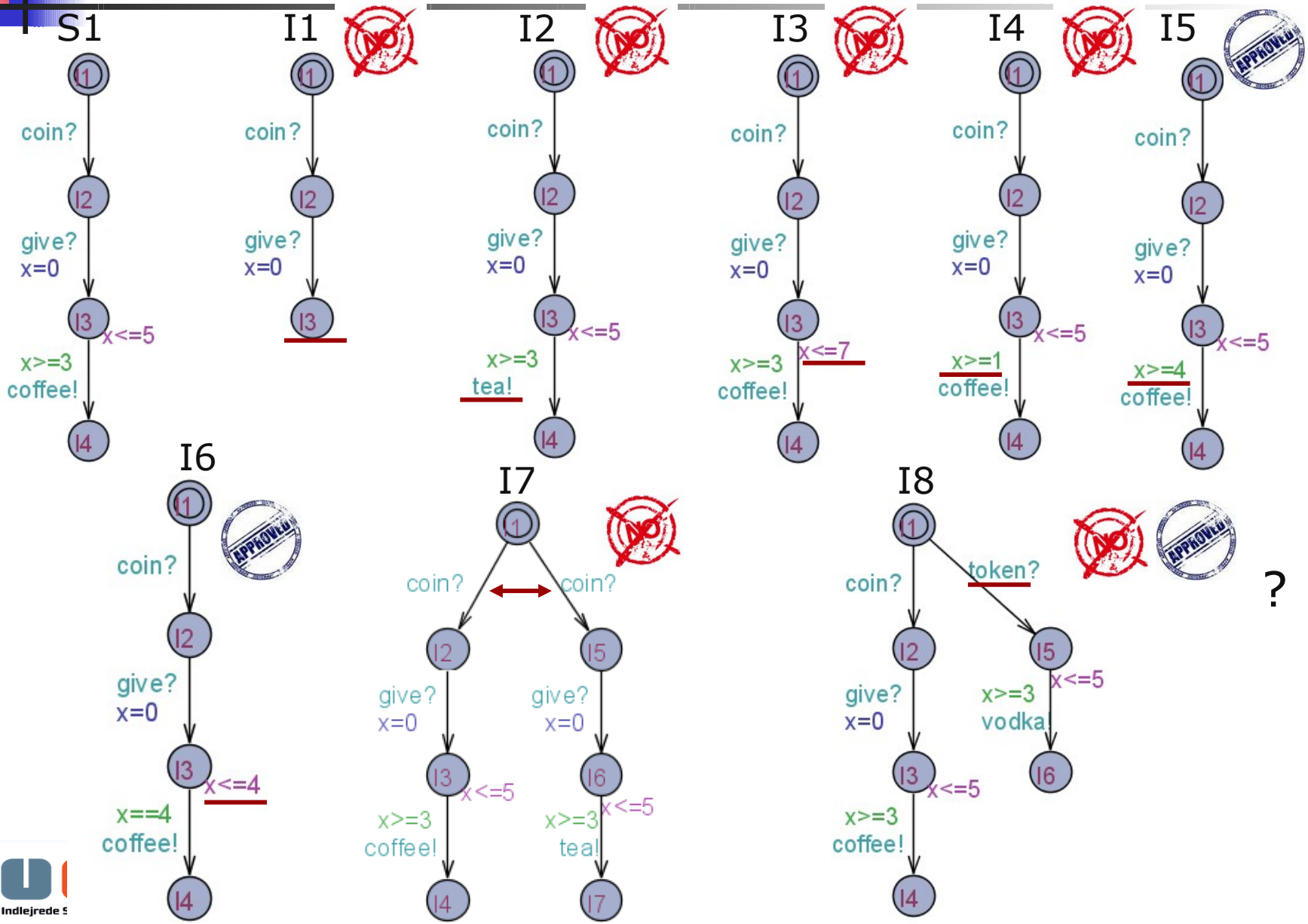
Conformance Relation



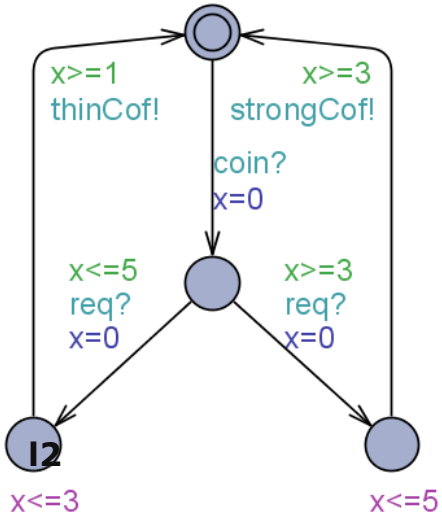
- Timed Automata with Timed-LTS semantics
- **Input** actions (?) are controlled by the environment
- **Output** actions (!) are controlled by the implementation
- Implementations are *input enabled*
- ***Testing hypothesis:*** IUT can be modeled by some (unknown) TA

Center for Indlejrede Software Systemer

Does I_n conform-to S_1 ?



Timed Conformance



- Derived from Tretman's IOCO

- Let **I, S** be timed I/O LTS, P a set of states
- **TTr**(P): the set of *timed traces* from P
  - eg.: σ = coin?.**5**.req?.**2**.thinCoffee!.**9**.coin?
- **Out**(P **after** σ) = possible ***outputs*** and ***delays*** after σ
  - eg. out ({l2,x=1}): {thinCoffee, **0…2**}

- **I rt-ioco S =def**
  - **∀σ ∈ TTr(S): Out(I after σ) ⊆ Out(S after σ)**
  - **TTr(I) ⊆ TTr(S)** *if s and I are input enabled*

- **Intuition**
  - **no illegal output is produced and**
  - **required output is produced (at right time)**

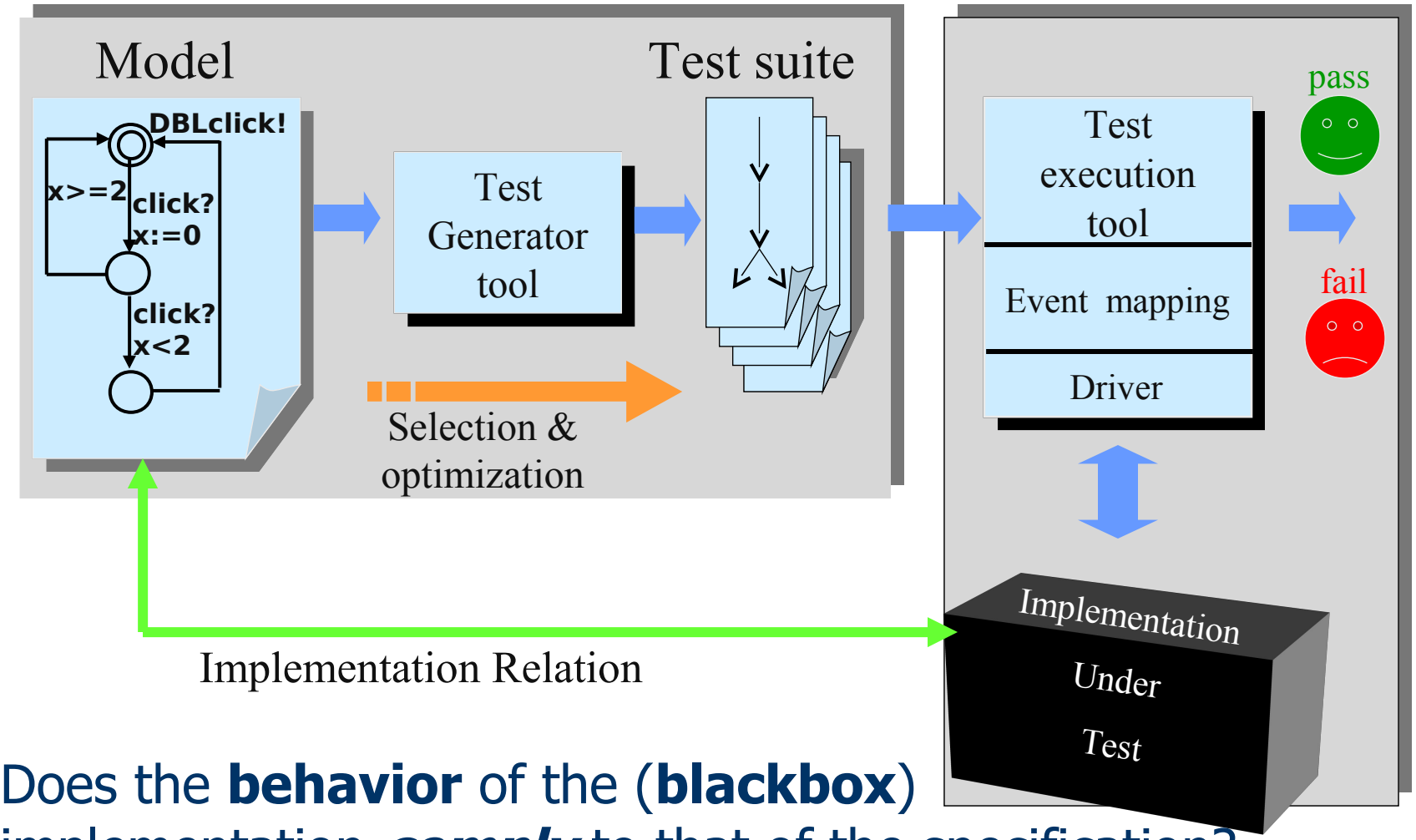
See also [Krichen&Tripakis, Khoumsi] and [Briones '07]

Off-Line Test Generation

Controllable
Timed Automata

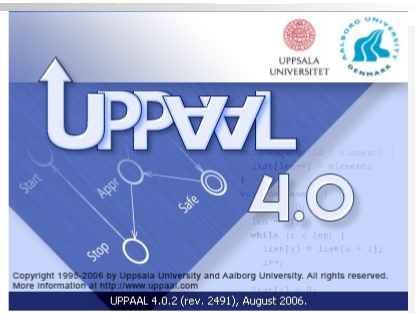
BRICS
Basic Research
in Computer Science

Model Based Conformance Testing



Does the **behavior** of the (**blackbox**) implementation ***comply*** to that of the specification?

Test generation using model-checking



System model

`myProtocol.xml`

Test purpose
Property

`E<> connection.Established`

Uppaal Model-Checker

Trace (witness)

- Some
- Random
- Shortest
- Fastest

`testConnectionEst.trc`

Use trace scenario as test case??!!

Controllable Timed Automata

Input Enabled:
all inputs can always be accepted.

Assumption about model of SUT

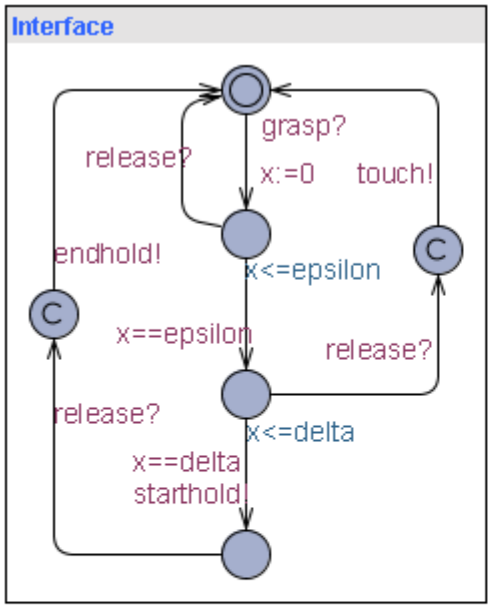
Output Urgent:
enabled outputs will occur immediately.

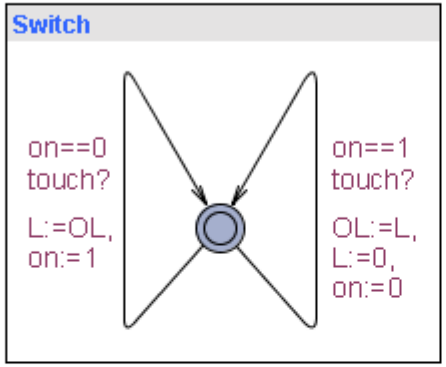
Determinism:
two transitions with same input/output leads to the same state.

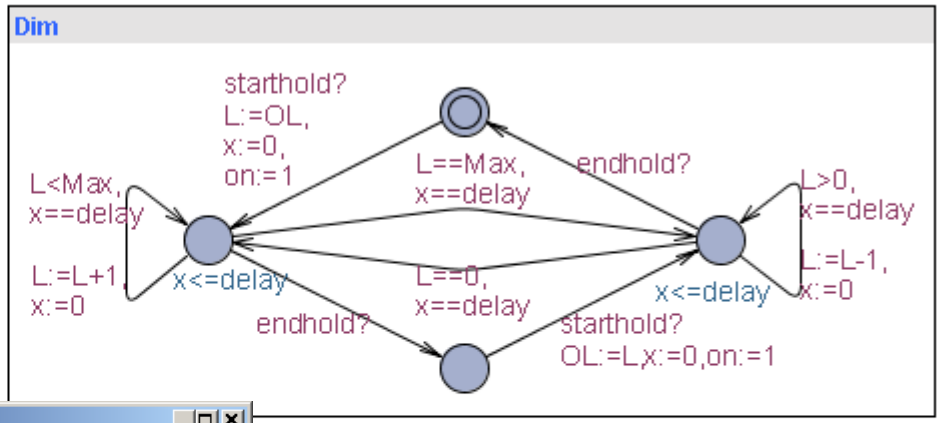
Isolated Outputs:
if an output is enabled, no other output is enabled.

Example

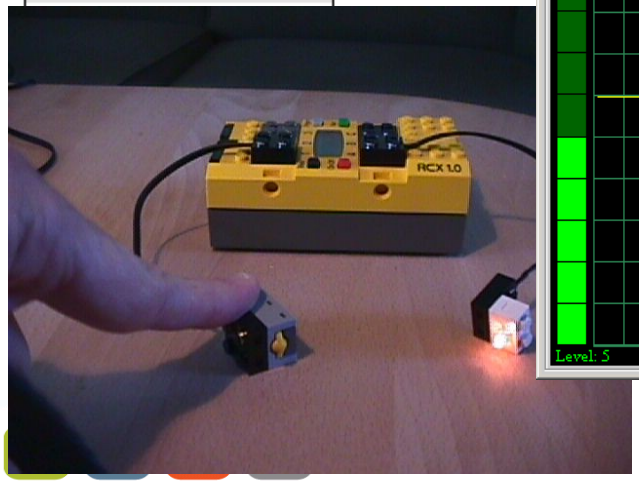
Light Controller



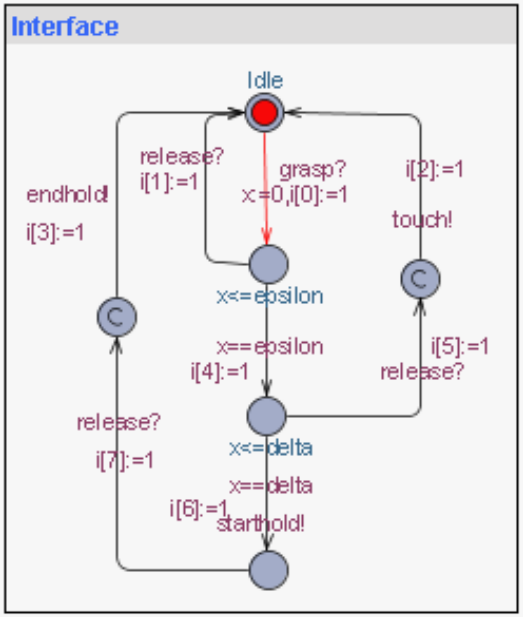


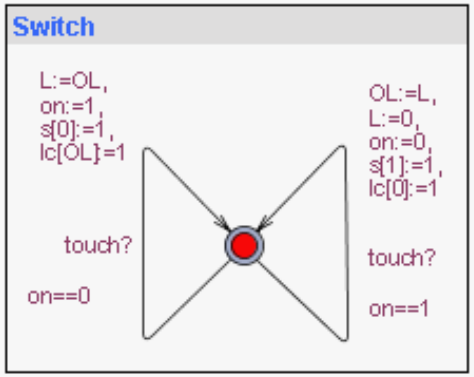


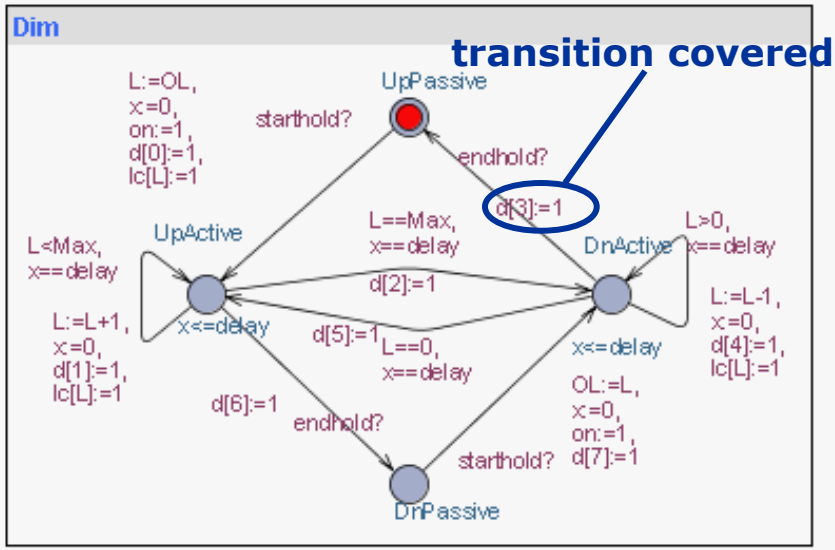
User



Off-Line Testing *= Optimal Reachability*



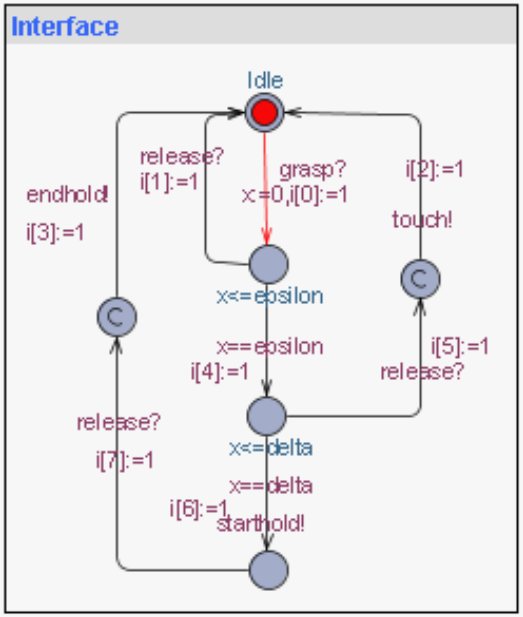


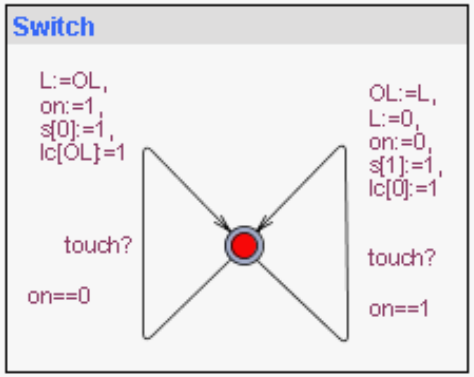


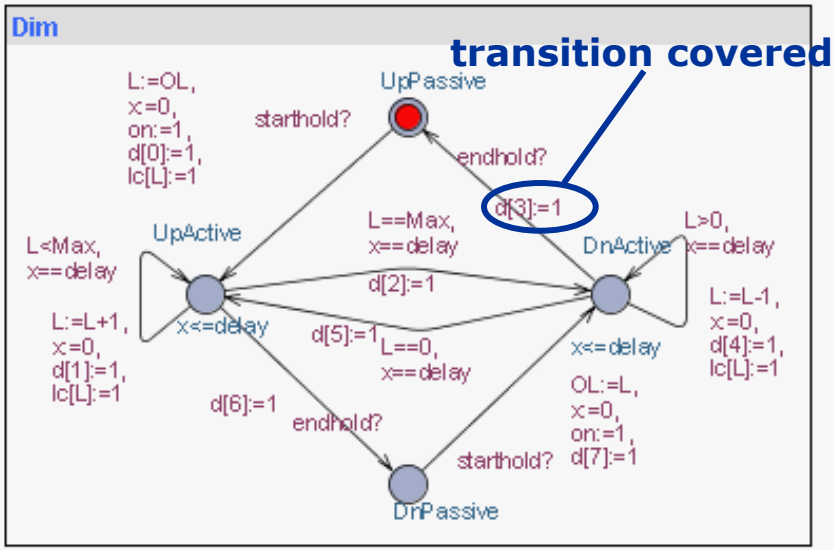
- Specific Test Purposes
- Model Coverage
- Optimal test-suites

Off-Line Testing *= Optimal Reachability*

Fastest Transition Coverage =12600 ms







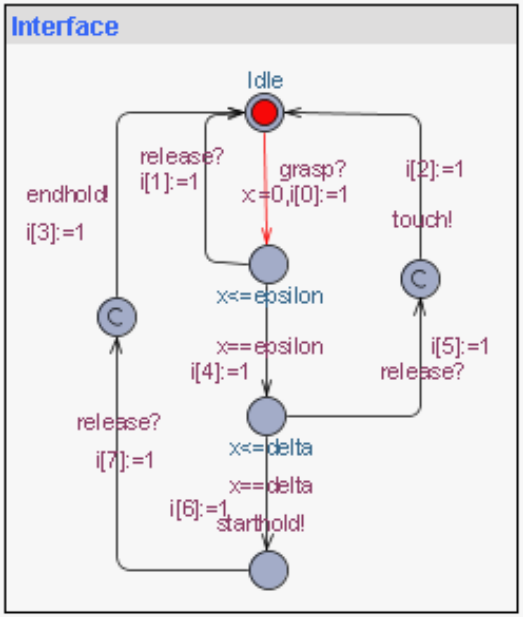

```
out(IGrasp);              //touch:switch light on
silence(200);
out(IRelease);
in(OSetLevel,0);

out(IGrasp); //@200        // touch: switch light off
silence(200);
out(IRelease);//touch
in(OSetLevel,0);

//9
out(IGrasp); //@400        //Bring dimmer from ActiveUp
silence(500); //hold       //To Passive DN (level=0)
in(OSetLevel,0);
out(IRelease);
```

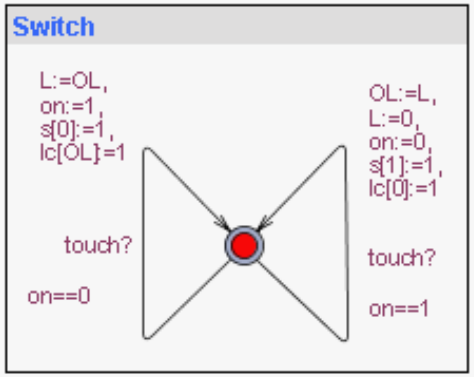
Page 1

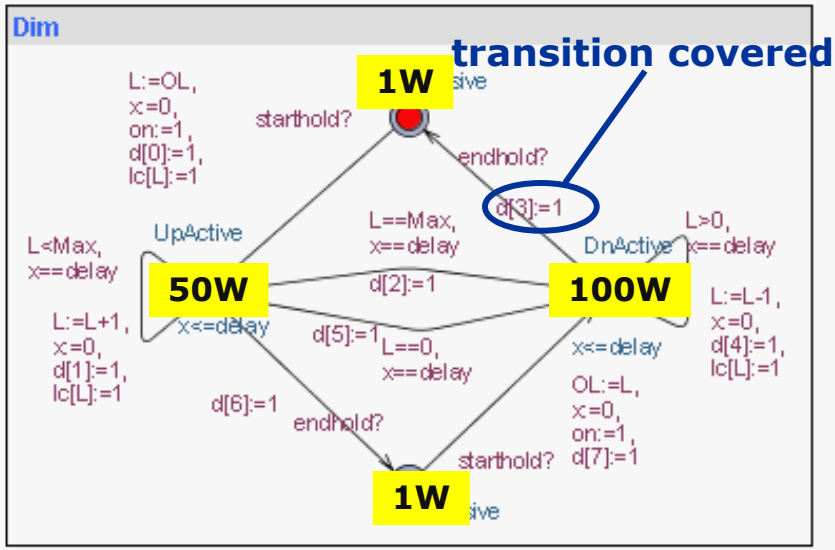
```
//13
out(IGrasp); //@900        // Bring dimmer  PassiveDn->ActiveDN->
silence(500);//hold        // ActiveUP+increase to level 10
silence(1000); in(OSetLevel,1);
silence(1000); in(OSetLevel,2);
silence(1000); in(OSetLevel,3);
silence(1000); in(OSetLevel,4);
silence(1000); in(OSetLevel,5);
silence(1000); in(OSetLevel,6);
silence(1000); in(OSetLevel,7);
silence(1000); in(OSetLevel,8);
silence(1000); in(OSetLevel,9);
silence(1000); in(OSetLevel,10
silence(1000);  in(OSetLevel,9); //bring dimm State to ActiveDN

out(IRelease);            //check release->grasp is ignored
out(IGrasp); //@12400
out(IRelease);
silence(dfTolerance);
```

Page 2

Off-Line Testing *= Optimal Reachability*





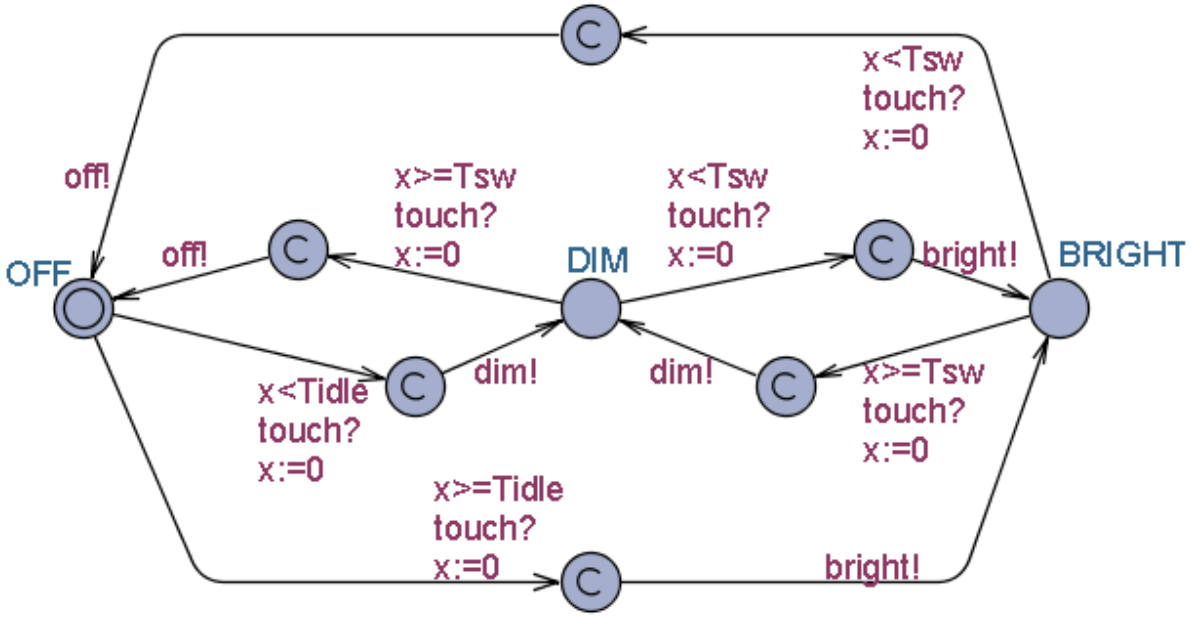


- Specific Test Purposes
- Model Coverage
- Optimal test-suites

Center for Indlejrede Software Systemer

Timed Automata

(E)FSM+clocks+guards+resets

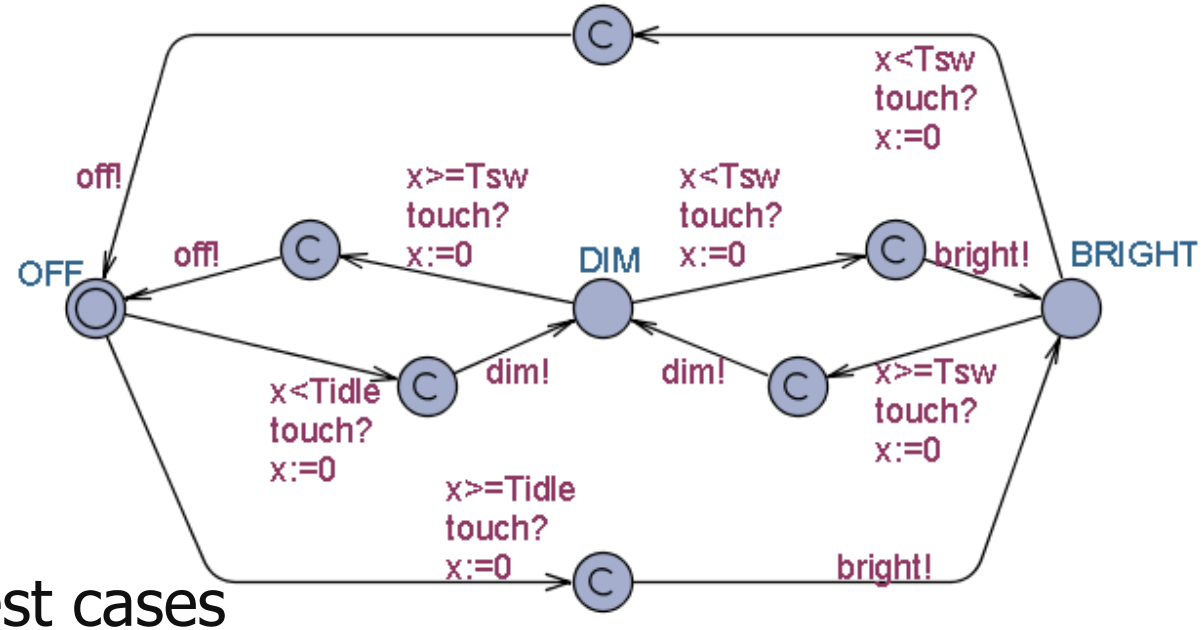



```
T_sw=4
T_idle=20
```

WANT: if touch is issued twice quickly then the light will get brighter; otherwise the light is turned off.

Solution: Add real-valued clock **x**

Timed Tests




```
T_sw=4
T_idle=20
```

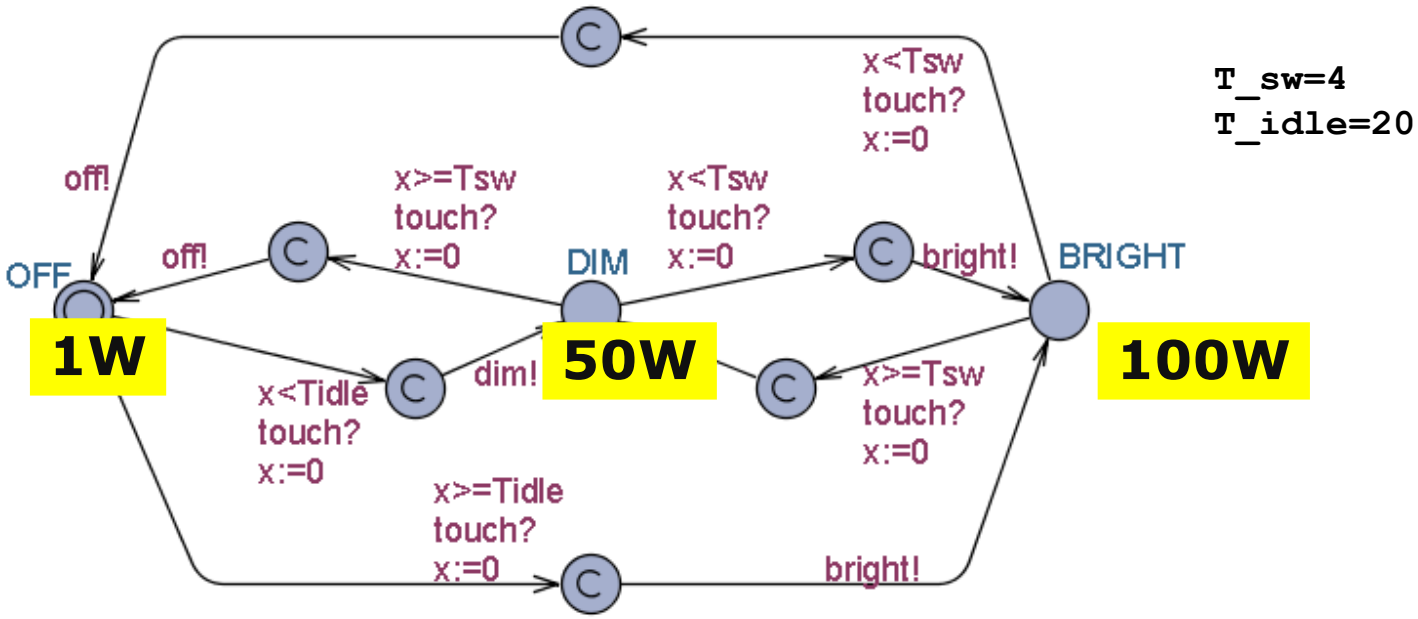
EXAMPLE test cases

```
0·touch!·0·dim?·2·touch!·0·bright?·2·touch!·off?·PASS
0·touch!·0.dim?·2½·touch!·0·bright?·3·touch!·off?·PASS
0·touch!·0·dim?·5touch!·0·off?·PASS
0·touch!·0·dim?·5·touch!·0·off?·50·touch!·0·bright?·6·touch!·0·dim?·PASS
```

INFINITELY MANY SEQUENCES!!!!!!

Center for Indlejrede Software Systemer

Optimal Tests

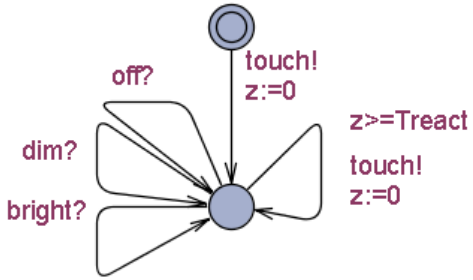


- ***Shortest*** test for bright light??
- ***Fastest*** test for bright light??
- ***Fastest*** edge-covering test suite??
- Least ***power*** consuming test??

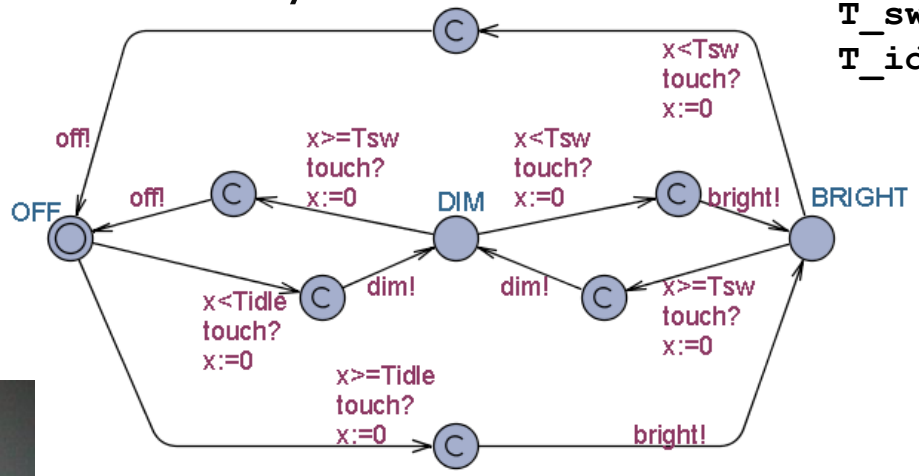
Center for Indlejrede Software Systemer

Simple Light Controller

Environment model



System model



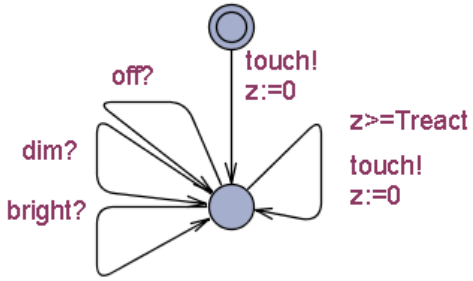

```
T_react=2
T_sw=4
T_idle=20
```

Center for Indlejrede Software Systemer

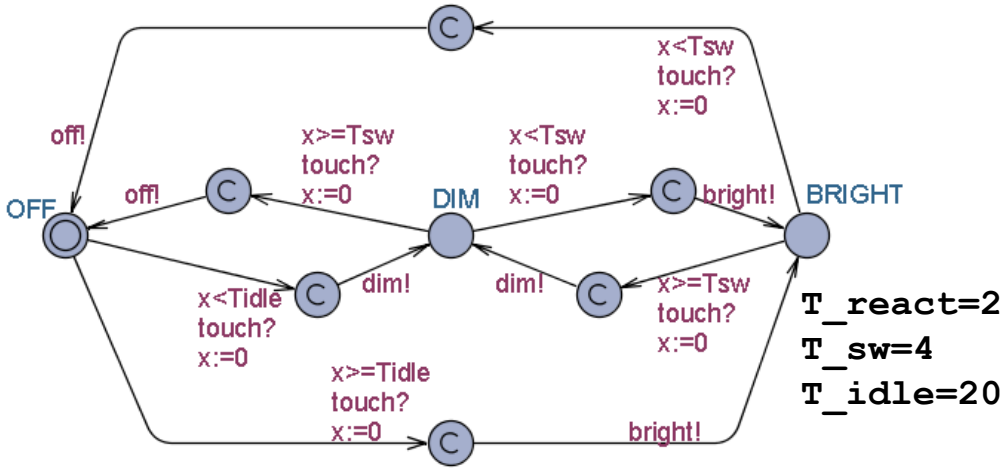
Test Purposes

A specific test objective (or observation) the tester wants to make on SUT

Environment model



System model



TP1: Check that the light can become bright:

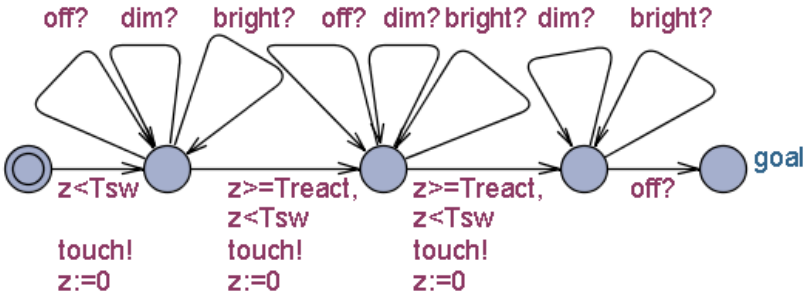
`E<> LightController.bright`

- *Shortest* Test: 20·touch!·0·bright?·PASS
- *Fastest* Test: 0·touch!·0·dim?·2·touch!·0·bright ?·PASS

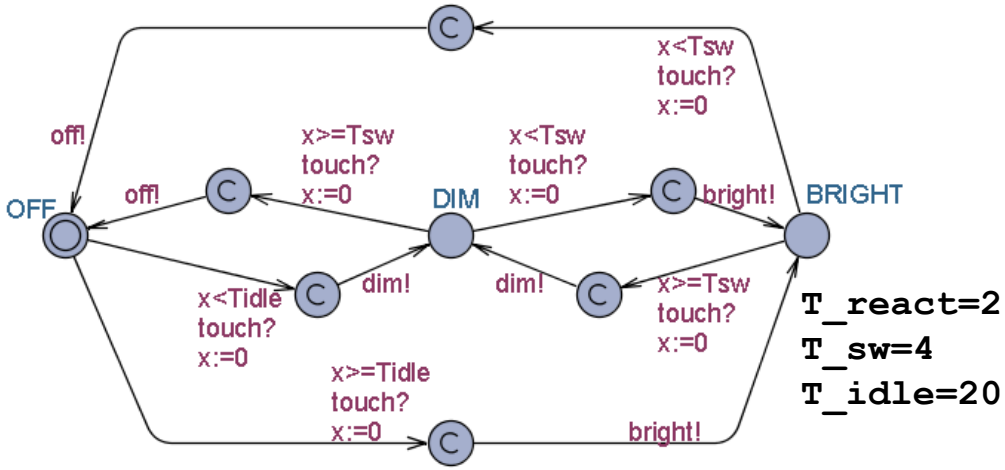
Center for Indlejrede Software Systemer

Test Purposes 2

Environment model*TP2



System model



TP2: Check that the light switches off after three successive touches

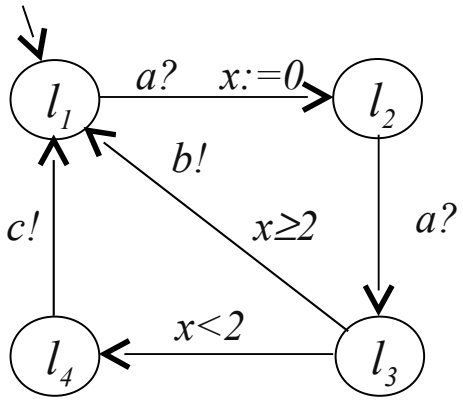
Use restricted environment and **`E<> tpEnv.goal`**

- The fastest test sequence is
0·touch!·0·dim?·2·touch!·0·bright?·2·touch!·0·off?·PASS

Center for Indlejrede Software Systemer

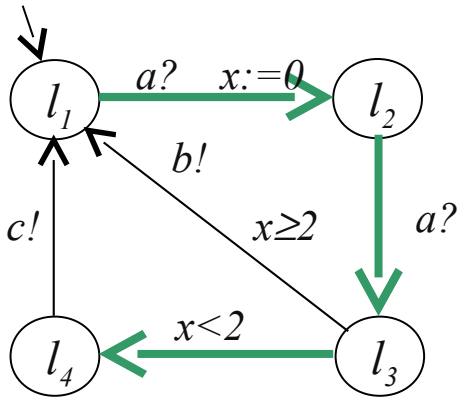
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage



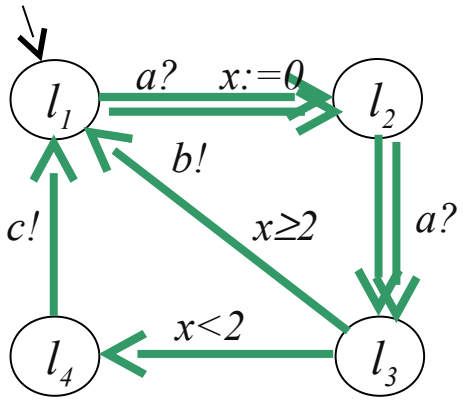
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage



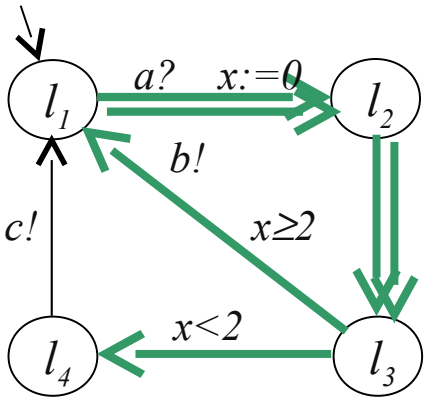
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage



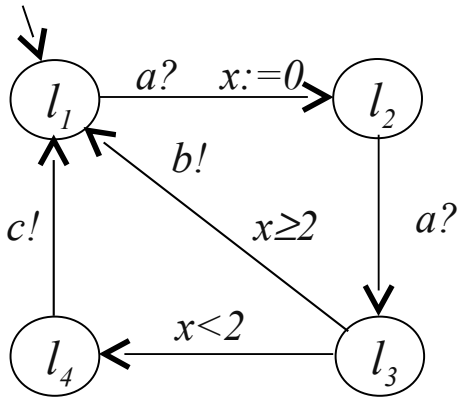
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Location coverage,
  - Edge coverage,
  - Definition/use pair coverage



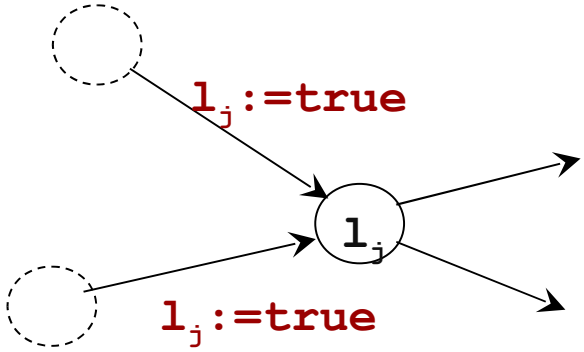
Coverage Based Test Generation

- Multi purpose testing
- Cover measurement
- Examples:
  - Locations coverage,
  - Edge coverage,
  - Definition/use pair coverage
  - All Definition/Use pairs
- Generated by min-cost reachability analysis of annotated graph



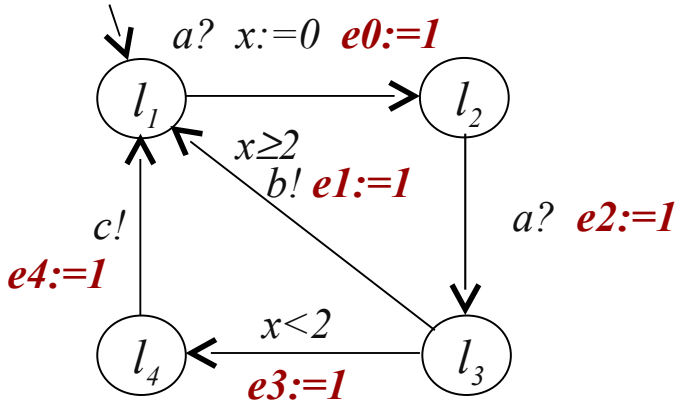
Location Coverage

- Test sequence traversing all locations
- Encoding:
  - Enumerate locations $l_0, \ldots, l_n$
  - Add an auxiliary variable l_i for each location
  - Label each ingoing edge to location i l_i:=true
  - Mark initial visited l_0:=true
- Check: **EF(l_0=true ∧ … ∧ l_n=true)**

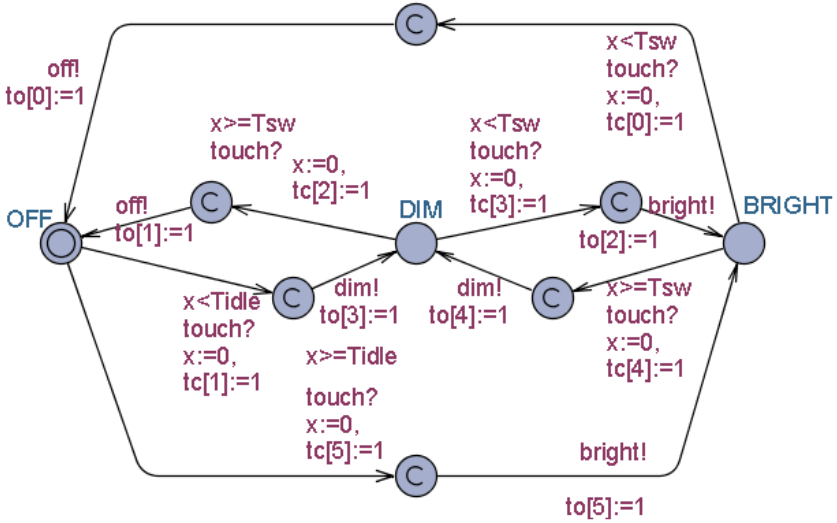


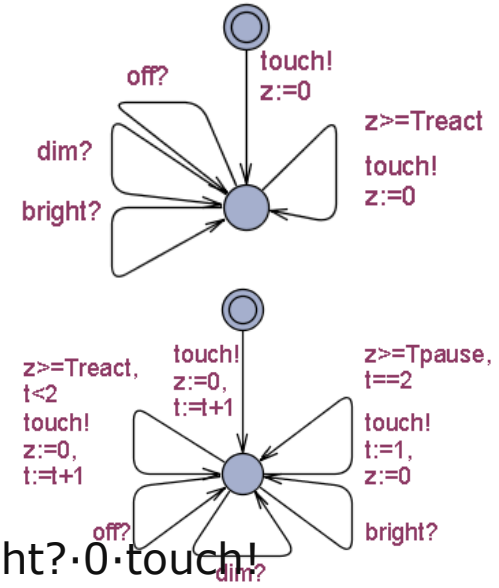
Edge Coverage

- Test sequence traversing all edges
- Encoding:
  - Enumerate edges $e_0, \ldots, e_n$
  - Add auxiliary variable e_i for each edge
  - Label each edge $e_i := true$
- Check: $EF(e_0 = true \wedge \ldots \wedge e_n = true)$



Edge Coverage





EC: *T_react=0* 0·touch!·0·dim?·0·touch!·0·bright?·0·touch!·0·off?·
20·touch!·0·bright?·4·touch!·0·dim?·4·touch!·0·off?·PASS

Time=28

EC': *T_react=2*

0·touch!·0·dim?·4·touch!·0·off?· 20·touch!·0·bright?·
4·touch!·0·dim?·2·touch!·0·bright?·2·touch!·0·off?·PASS

Time=32

EC'': *pausing user T_react=2, T_pause=5*

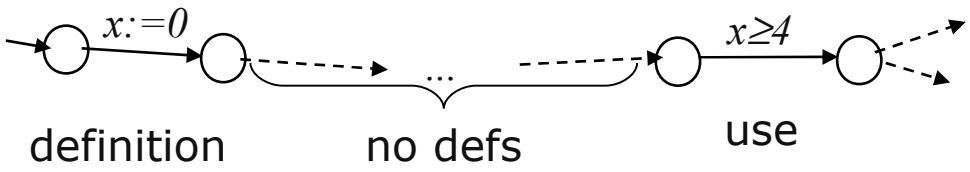
0·touch!·0·dim?·2·touch!·0·bright?·5·touch!·0·dim?·
4·touch!·0·off?·20·touch!·0·bright?·2·touch!·0·off?·PASS

Time=33

Center for Indlejrede Software Systemer

Definition/Use Pair Coverage

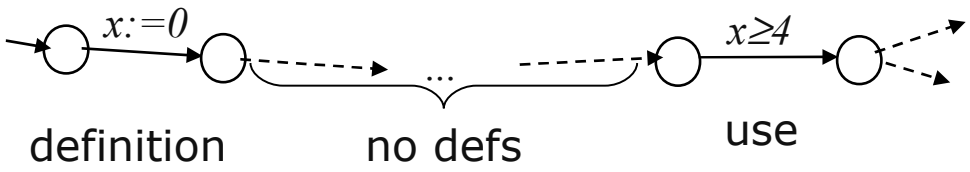
- Dataflow coverage technique
- Def/use pair of variable x:

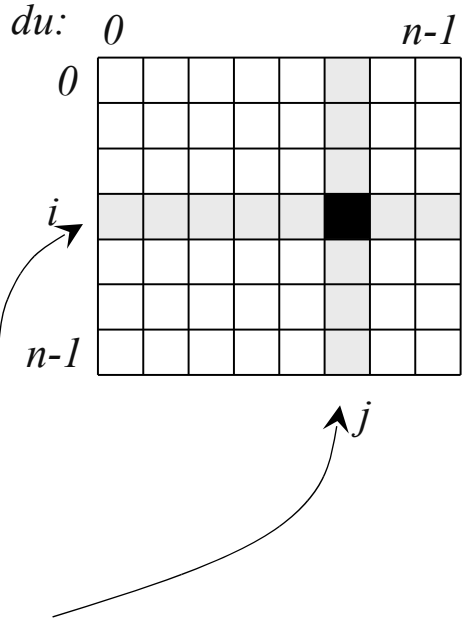


- Encoding:
  - $v_d \in \{false\} \cup \{e_0, \ldots, e_n\}$, initially false
  - Boolean array du of size $|E|$ x $|E|$
  - At definition on edge i: $v_d := e_i$
  - At use on edge j: if(v_d) then $du[v_d, e_j] := true$

Definition/Use Pair Coverage

- Dataflow coverage technique
- Def/use pair of variable x:





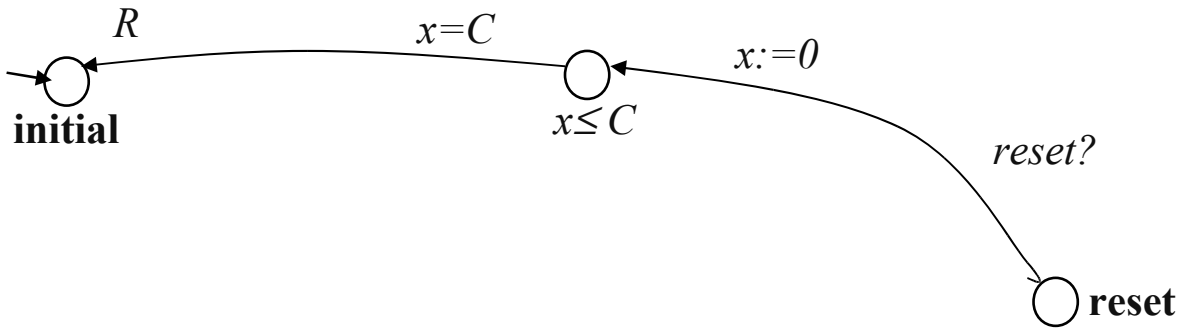
- Encoding:
  - $v_d \in \{false\} \cup \{e_0, \ldots, e_n\}$, initially false
  - Boolean array du of size $|E|$ x $|E|$
  - At definition on edge i: $v_d := e_i$
  - At use on edge j: if(v_d) then $du[v_d, e_j] := true$
- Check:
  - EF(all $du[i,j] = true$)

Uppaal CoVer by Anders Hessel and Paul Pettersson

- Generates coverage-optimal test-suites:
- Uses models with deterministic IUT and non-deterministic environment.
- Provides a language for specifying coverage:
  - locations, edges, variable definition-use
- Internally generates corresponding observer automata.
- Uses observer automata states to encode the state of coverage.
- => automatic model decoration, better control over model, dynamic (more efficient) memory usage
- Applied on Wireless Application Protocol gateway.

Test Suite Generation

- In general a set of test cases is needed to cover a test criteria
- Add global reset of SUT and environment model and associate a cost (of system reset)

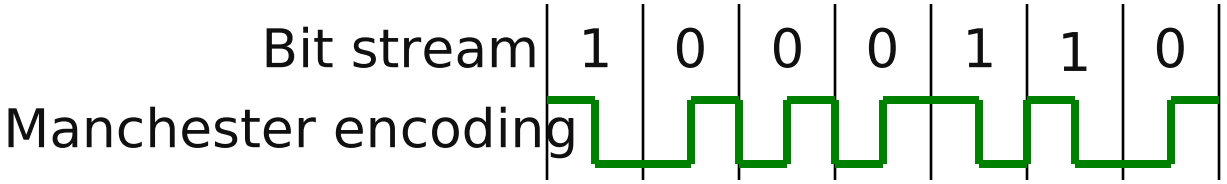


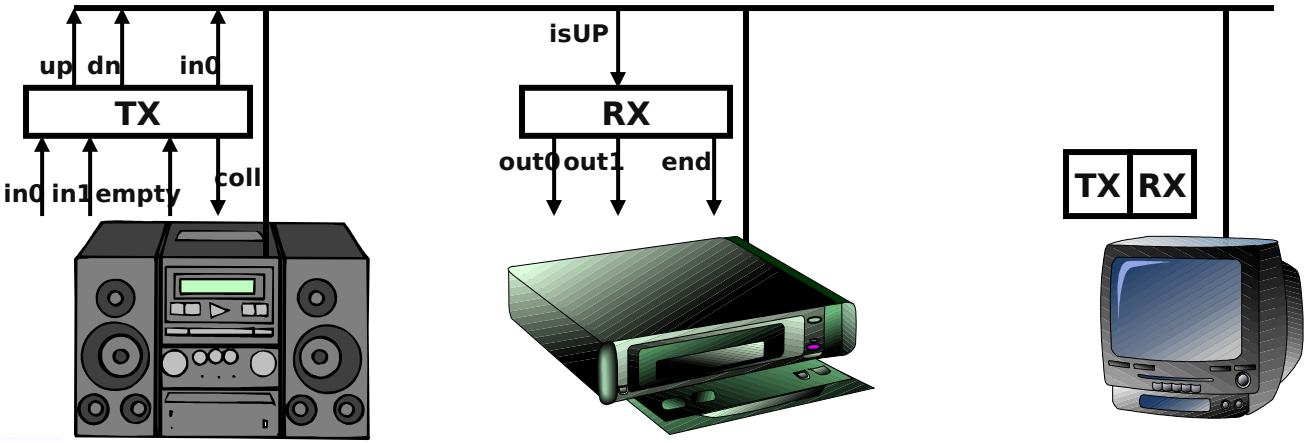
- Same encodings and min-cost reachability
- Test sequence $\sigma = \varepsilon_0, i_0, \ldots, \varepsilon_l, i_l,$ **reset** $\varepsilon_2, i_2, \ldots, \varepsilon_0, i_0,$ **reset**$, \varepsilon_l, i_l, \varepsilon_2, i_2, \ldots$
- Test suite $T = \{\sigma_1, \ldots, \sigma_n\}$ with minimum cost

$\underbrace{\qquad\qquad}_{\sigma_i}$

The Philips Audio Protocol

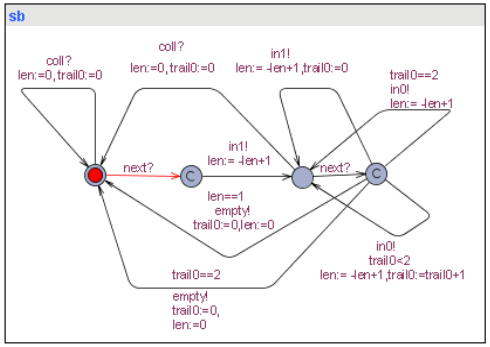
- A bus based protocol for exchanging control messages between audio components
  - Collisions
  - Tolerance on timing events

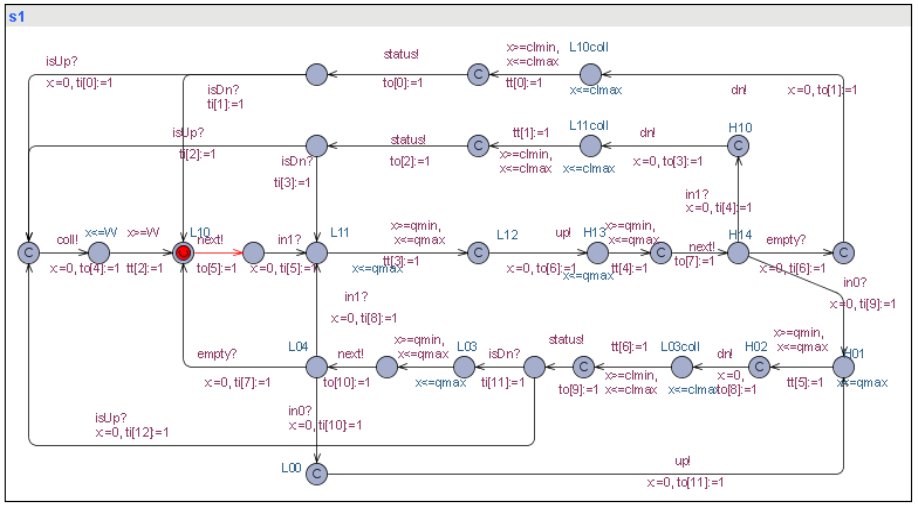


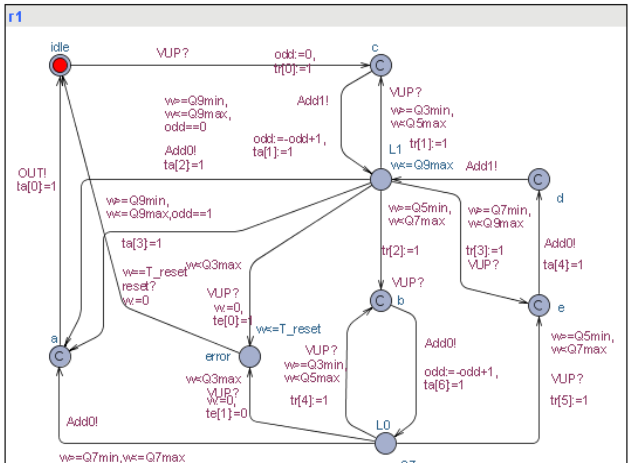


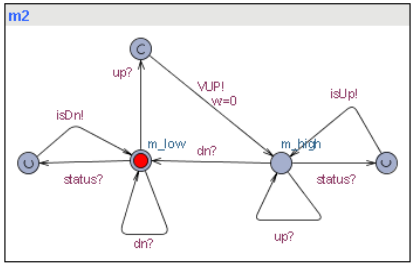
Center for Indlejrede Software Systemer

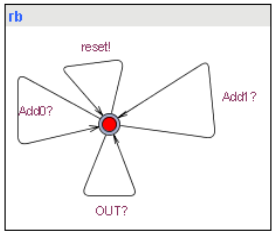
Philips Audio Protocol











Center for Indlejrede Software Systemer

Benchmark Example

- Philips Audio Protocol

Coverage Criterion	Execution time (μs)	Generation time (s)	Memory usage (KB)
Edge_{Sender}	212350	2.2	9416
$\text{Edge}_{Receiver}$	18981	1.2	4984
$\text{Edge}_{Sender,Bus,Receiver}$	114227	129.0	331408

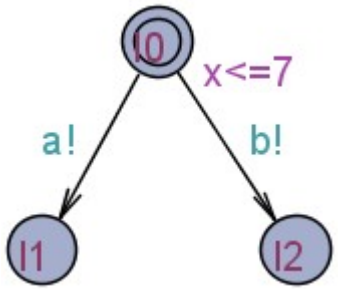
Off-Line Test Generation

Observable Timed Automata

BRICS
Basic Research
in Computer Science

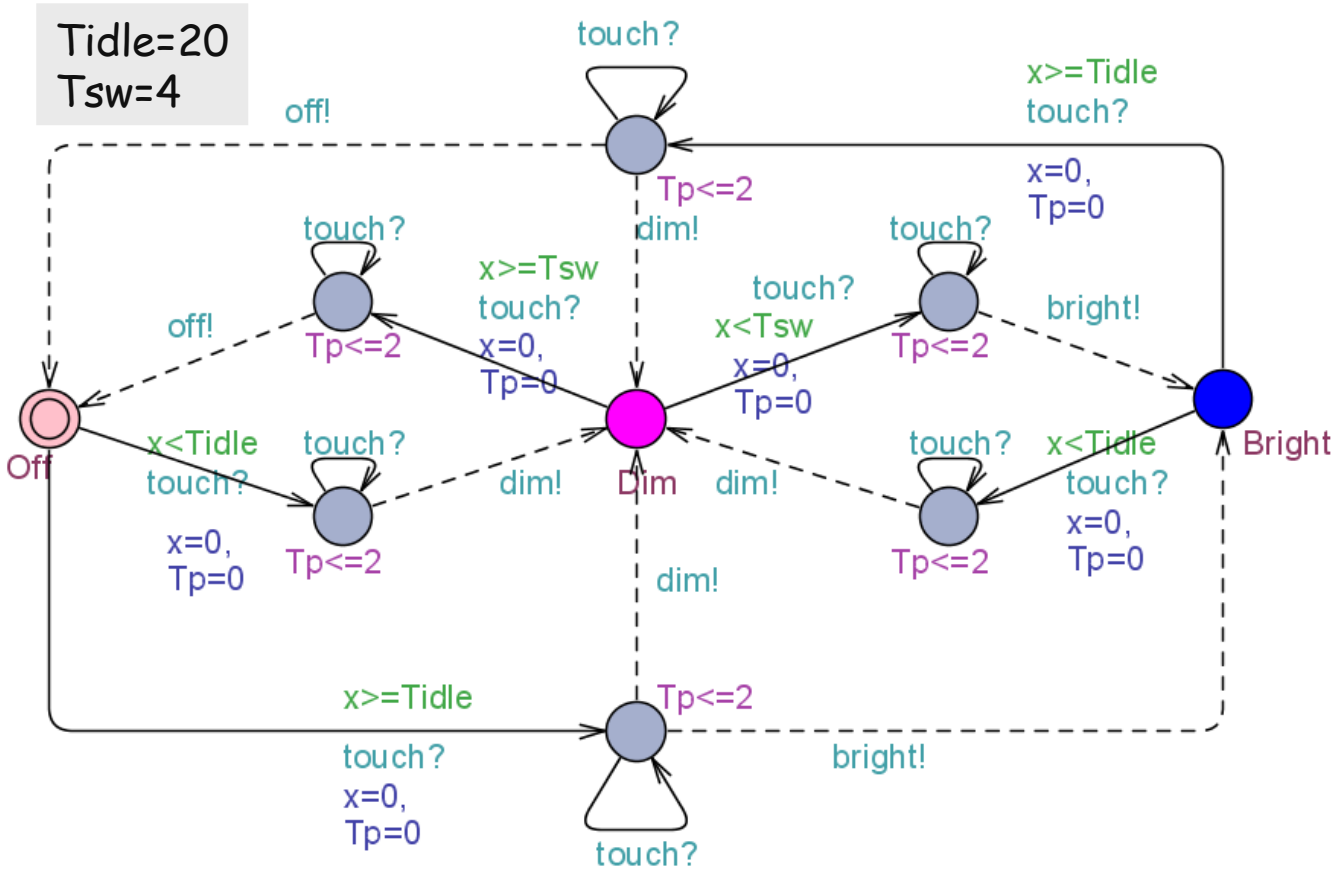
Observable Timed Automata

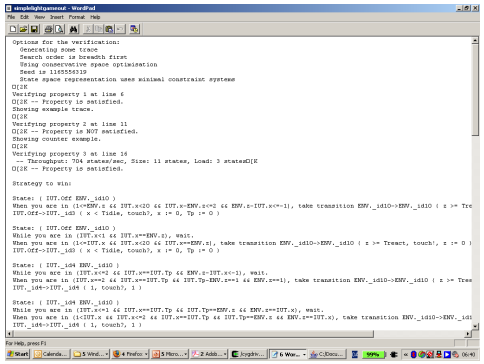
- **Determinism**:
  two transitions with same input/output leads to the same state
- **Input Enabled**:
  all inputs can always be accepted
- **Time Uncertainty of outputs**:
  timing of outputs uncontrollable by tester
- **Uncontrollable output**:
  IUT controls which enabled output will occur in what order



Center for Indlejrede Software Systemer

Timed Games and Testing

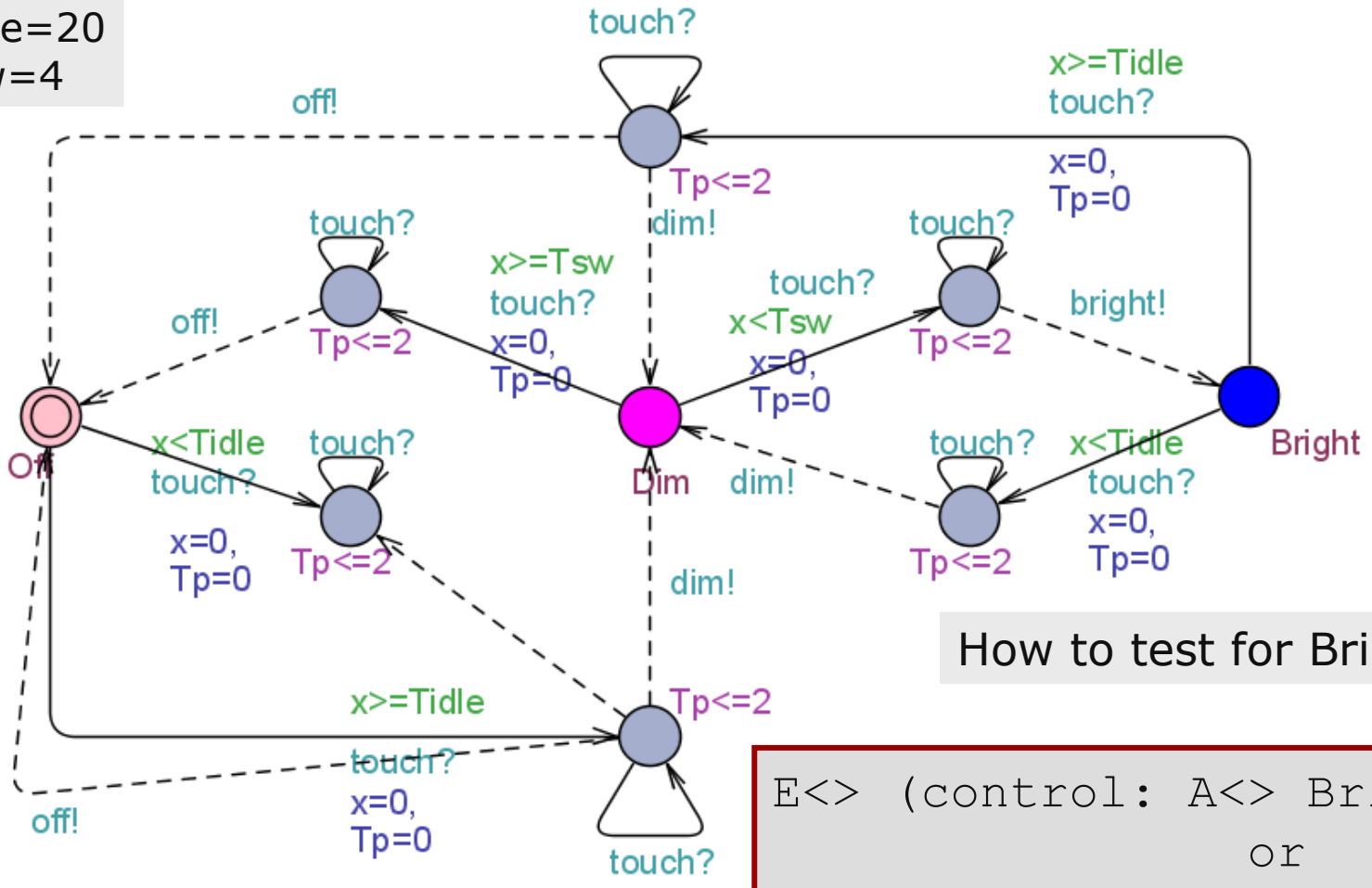




Off-line test-case generation =
Compute winning strategy for reaching **Bright**
Assign verdicts st. lost game means IUT not conforming

A trick light control

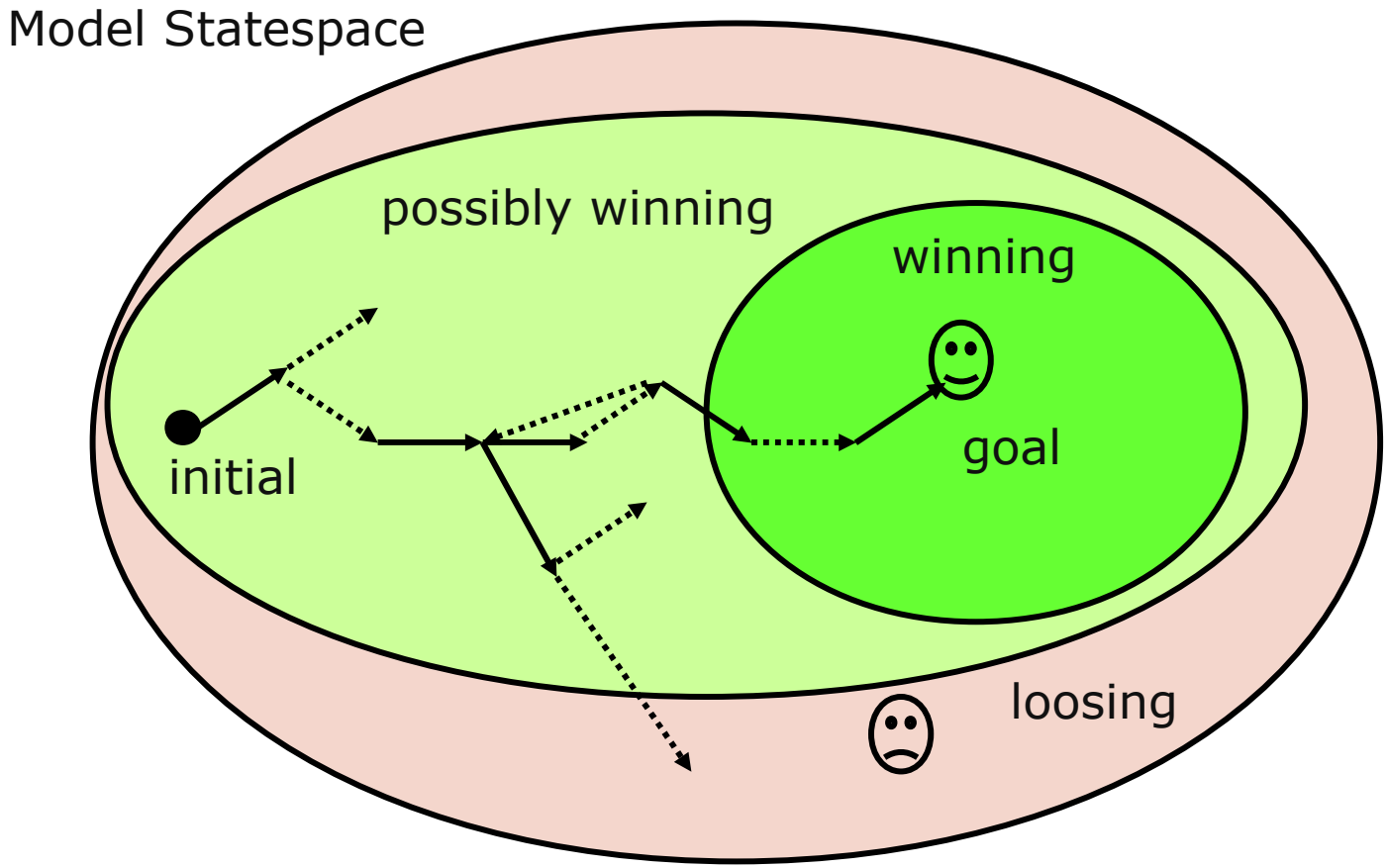
Tidle=20
Tsw=4



How to test for Bright ?

```
E<> (control: A<> Bright)
                or
<<c,u>> ◊ (<<c>> ◊ Bright)
```

Cooperative Strategies



- Play the game (execute test) while time available or game is lost
- Possibly using ranomized online testing

On-Line Testing

emulation, execution and evaluation

BRICS
Basic Research
in Computer Science

RELATED SITES: UPPAAL | COVER | TORX | TIMES | UPPAAL CORA

UPPAAL TRON

UPPAAL FOR TESTING REALTIME SYSTEMS ONLINE

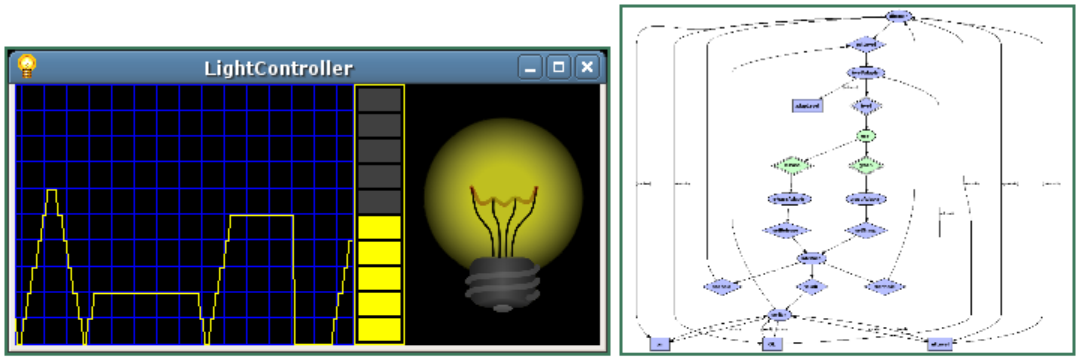
Main Page | Introduction | Adaptation | Testing | Publications | Examples | Download | Authors

Welcome!

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays, etc.).

UPPAAL TRON is a testing tool, based on UPPAAL engine, suited for black-box conformance testing of timed systems, mainly targeted for embedded software commonly found in various controllers. By online we mean that tests are derived, executed and checked simultaneously while maintaining the connection to the system in real-time.

Below are a screenshot of smart-lamp controller Java applet demo and automatically generated signal flow diagram of the system model:



Like UPPAAL, UPPAAL TRON is free for non-profit use, e.g. for evaluation, research, and teaching purposes. For more details please read the license.

Latest News

Version 1.5 released

17 June 2009

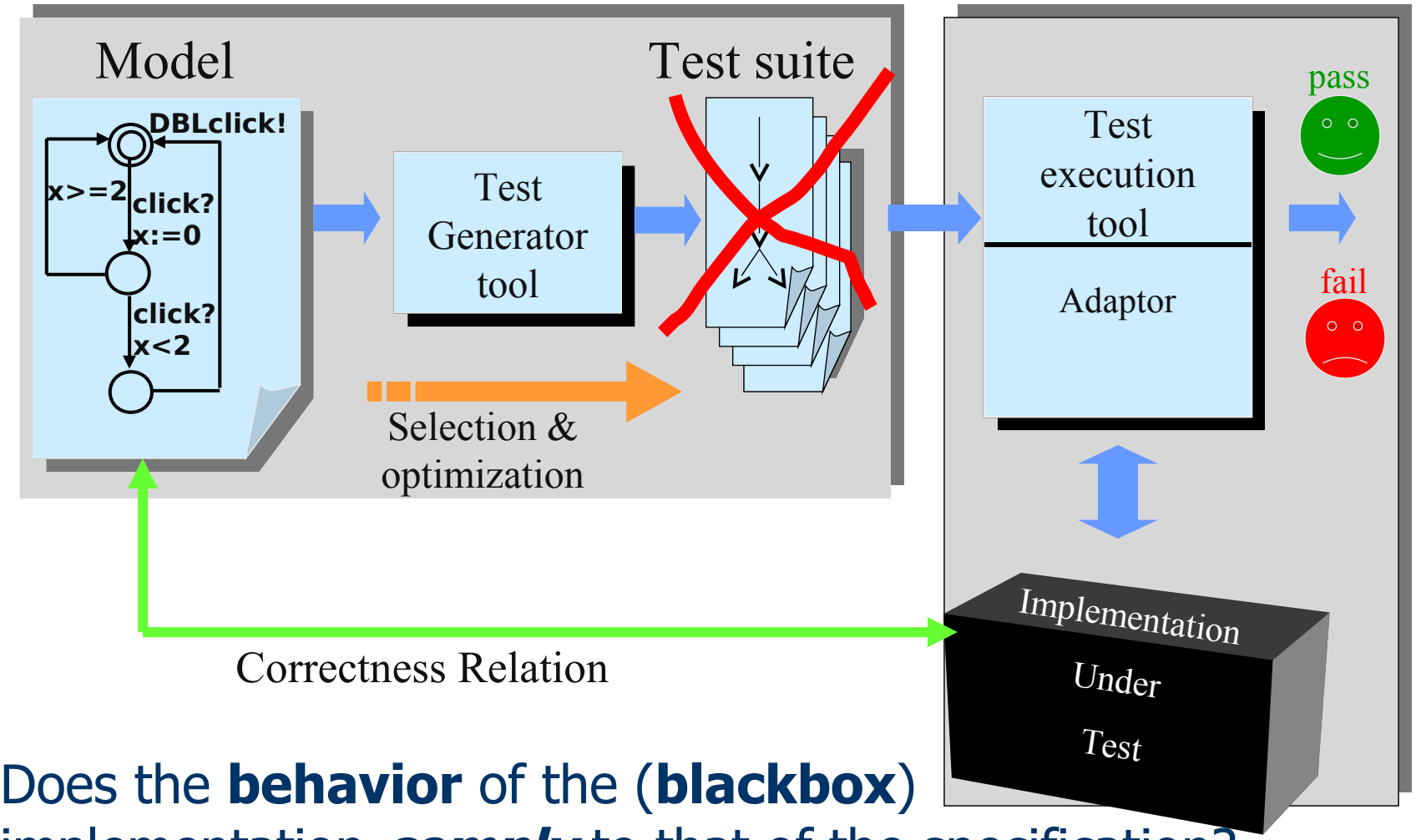
New preview release with fixes for usability issues. A new user-manual draft is available for building custom adapters. See Linux README and Windows README for details.

More News »

For comments or questions about this website, please email the author
Updated » Wednesday, 17-Jun-2009 19:44:27 CEST

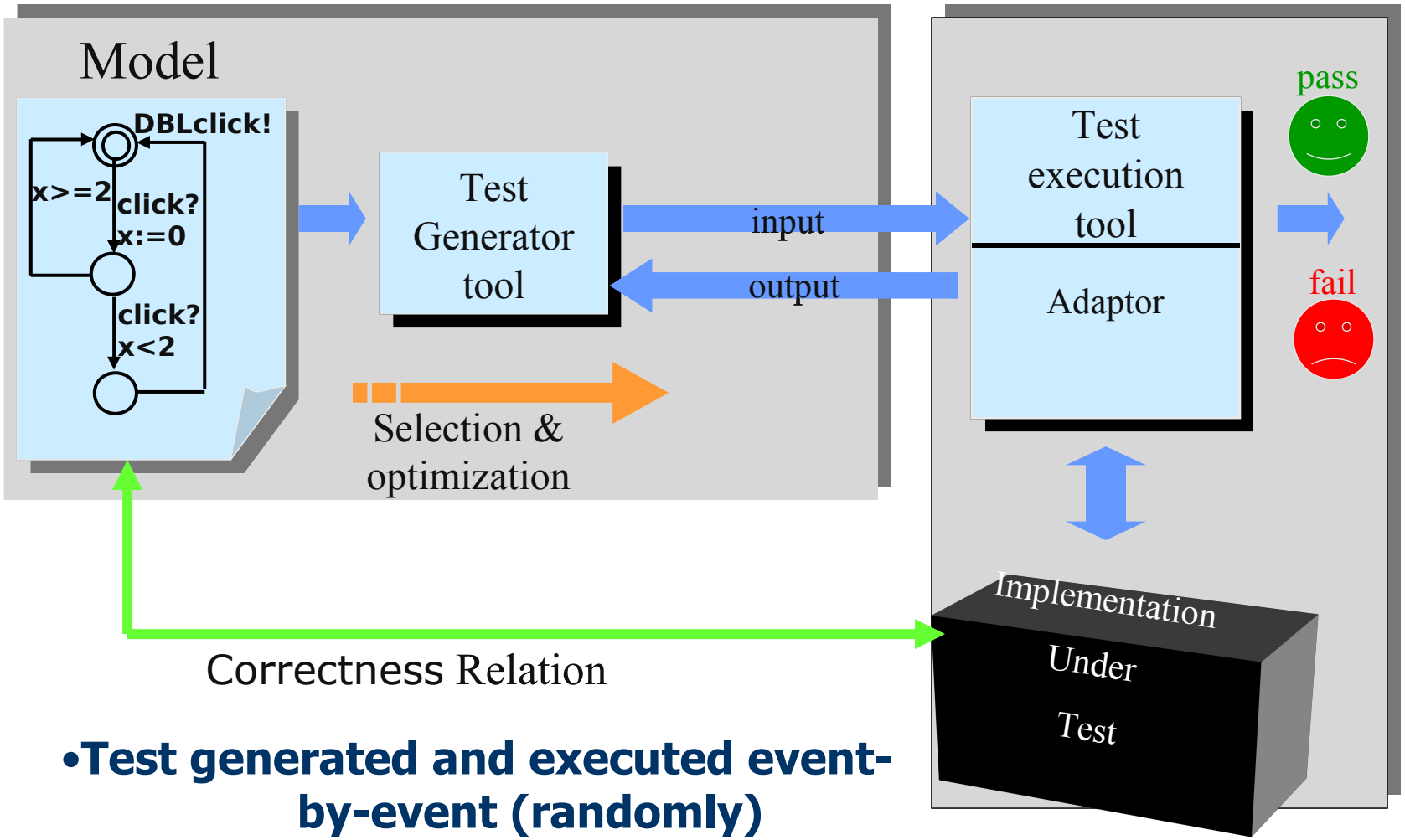
Center for Indlejrede Software Systemer

Automated Model Based Conformance Testing



Does the **behavior** of the (**blackbox**) implementation ***comply*** to that of the specification?

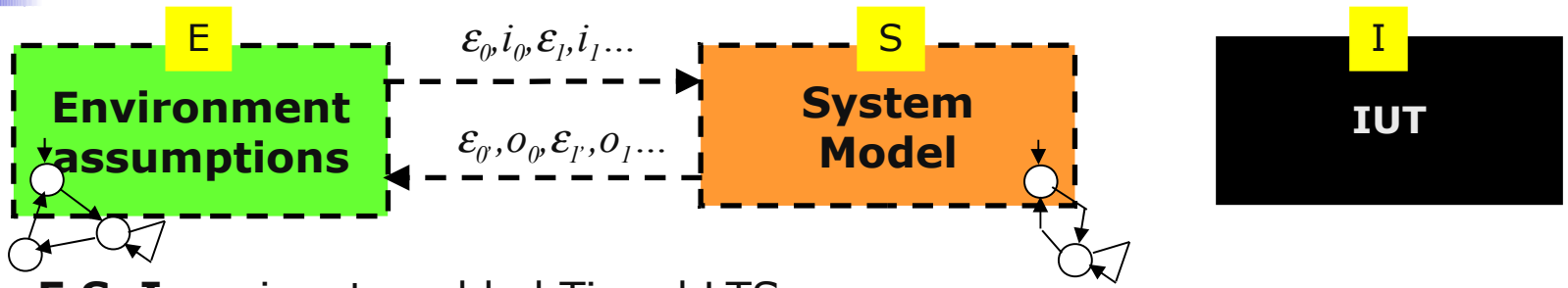
Online Testing



- **Test generated and executed event-by-event (randomly)**
- **A.K.A on-the-fly testing**

Implementation relation

Relativized real-time io-conformance



- **E,S, I** are input enabled Timed LTS
- **Let** *P* **be a set of states**
- **TTr**(*P*): the set of *timed traces* from states in *P*
- *P* **after** σ = the set of states reachable after timed trace σ
- **Out**(*P*) = possible outputs and delays from states in *P*

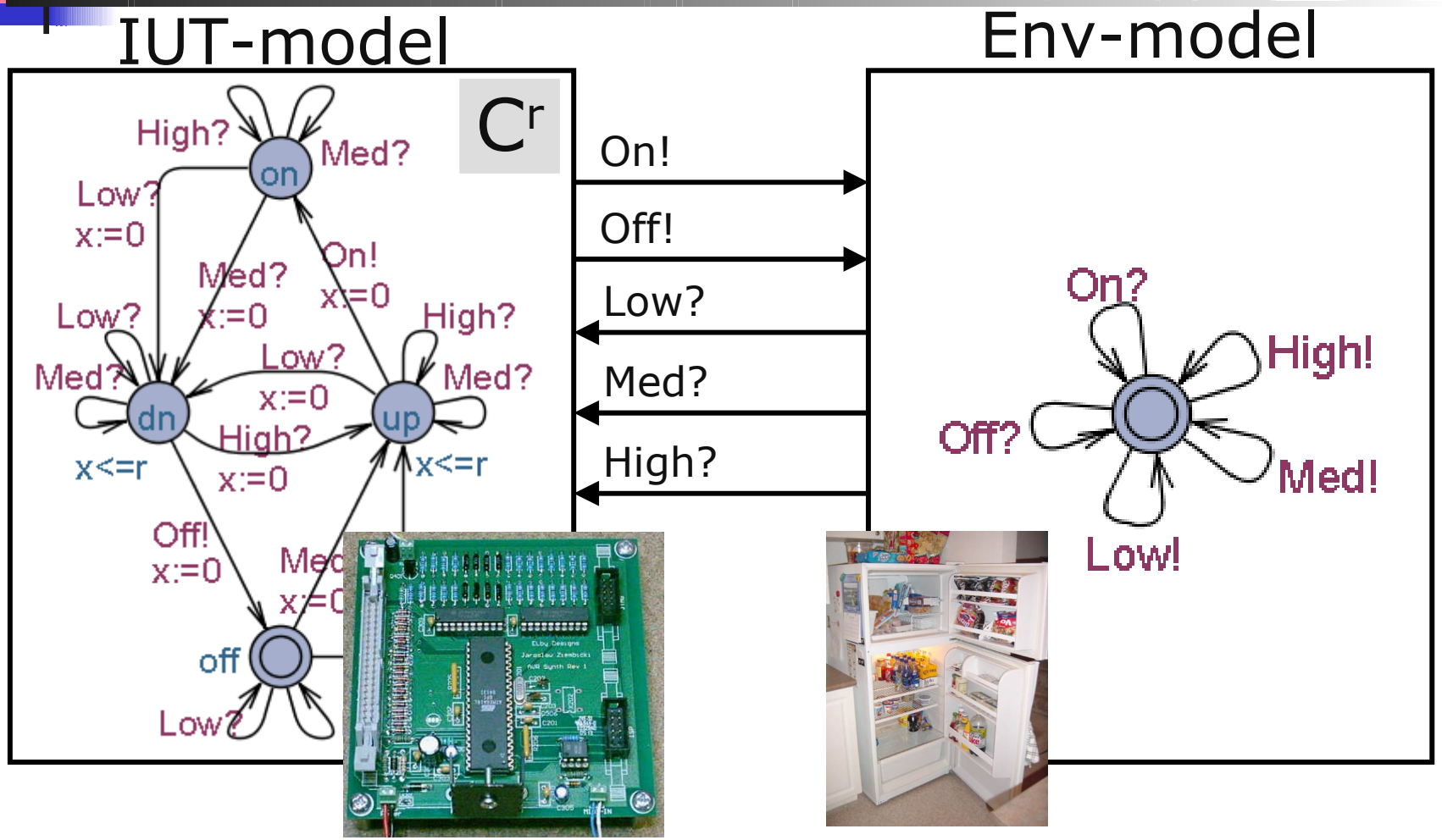
- I rt-ioco$_E$ S $=_{def}$

  $\forall \sigma \in TTr(E)$: Out((E,I) after σ) $\subseteq$ Out((E,S) after σ)

- I rt-ioco$_E$ s iff TTr(I) $\cap$ TTr(E) $\subseteq$ TTr(S) $\cap$ TTr(E) *// input enabled*

- **Intuition, for all assumed environment behaviors, the IUT**
  - **never produces illegal output, and**
  - **always produces required output in time**

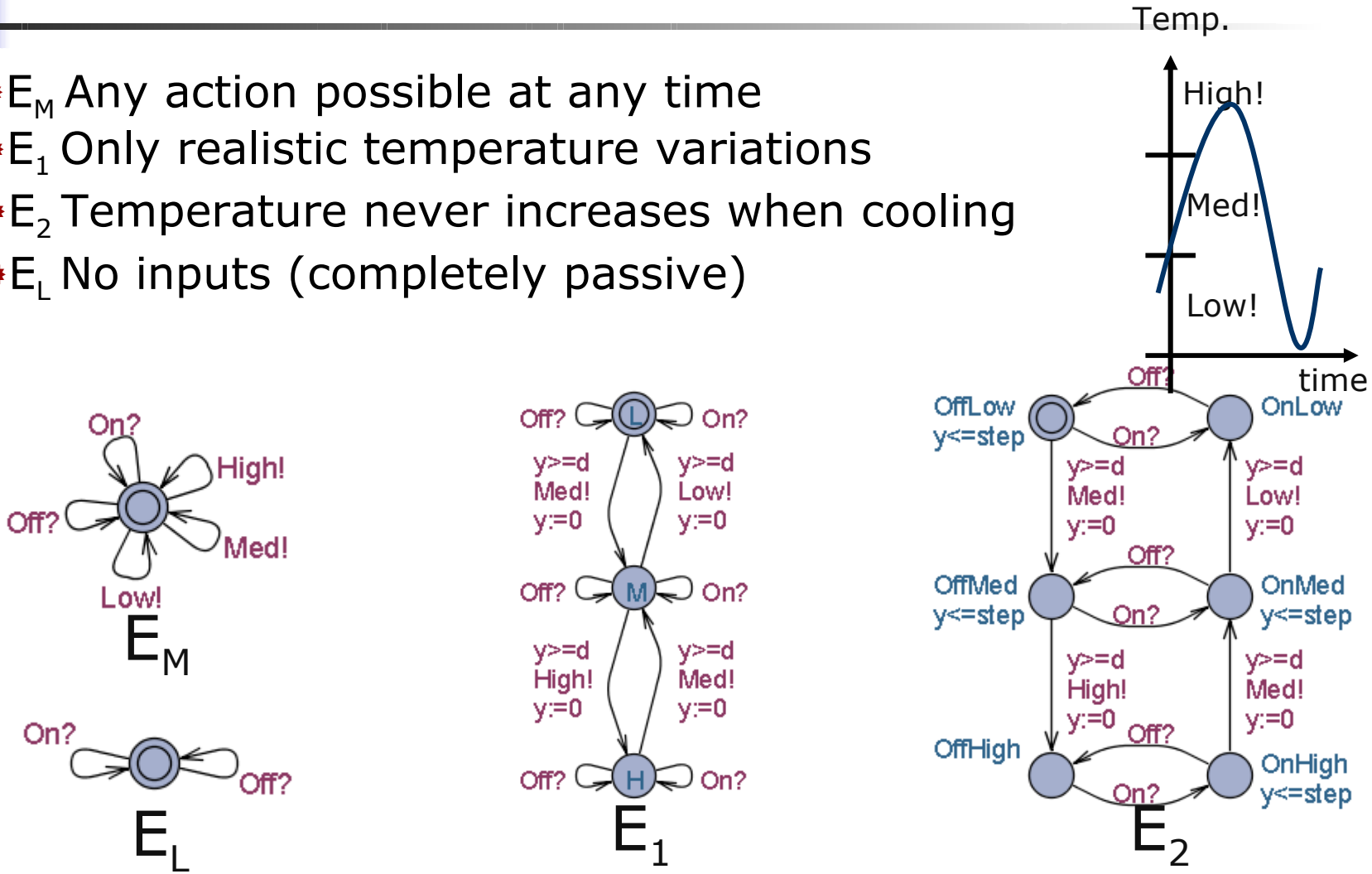
Sample Cooling Controller



- When T is high (low) switch on (off) cooling within r secs.
- When T is medium cooling may be either on or off (impl freedom)

Environment Modeling

- E_M Any action possible at any time
- E_1 Only realistic temperature variations
- E_2 Temperature never increases when cooling
- E_L No inputs (completely passive)



$$E_L \sqsubseteq E_2 \sqsubseteq E_1 \sqsubseteq E_M$$

Re-use Testing Effort

- Given I, E, S
- Assume I rt-ioco$_E$ S

1. Given new (weaker) system specification S'

$$\text{If } S \sqsubseteq S' \text{ then } I \text{ rt-ioco}_E S'$$

2. Given new (stronger) environment specification E'

$$\text{If } E' \sqsubseteq E \text{ then } I \text{ rt-ioco}_{E'} S$$

Online Testing Algorithm

BRICS
Basic Research
in Computer Science

Algorithm Idea: State-set tracking

- Dynamically compute all potential states that the model M can reach after the timed trace $\varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2, \ldots$

  [Tripakis] Failure Diagnosis

- *Z=M* ***after*** *$(\varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2)$*

- *If Z= ∅ the IUT has made a computation not in model:* ***FAIL***

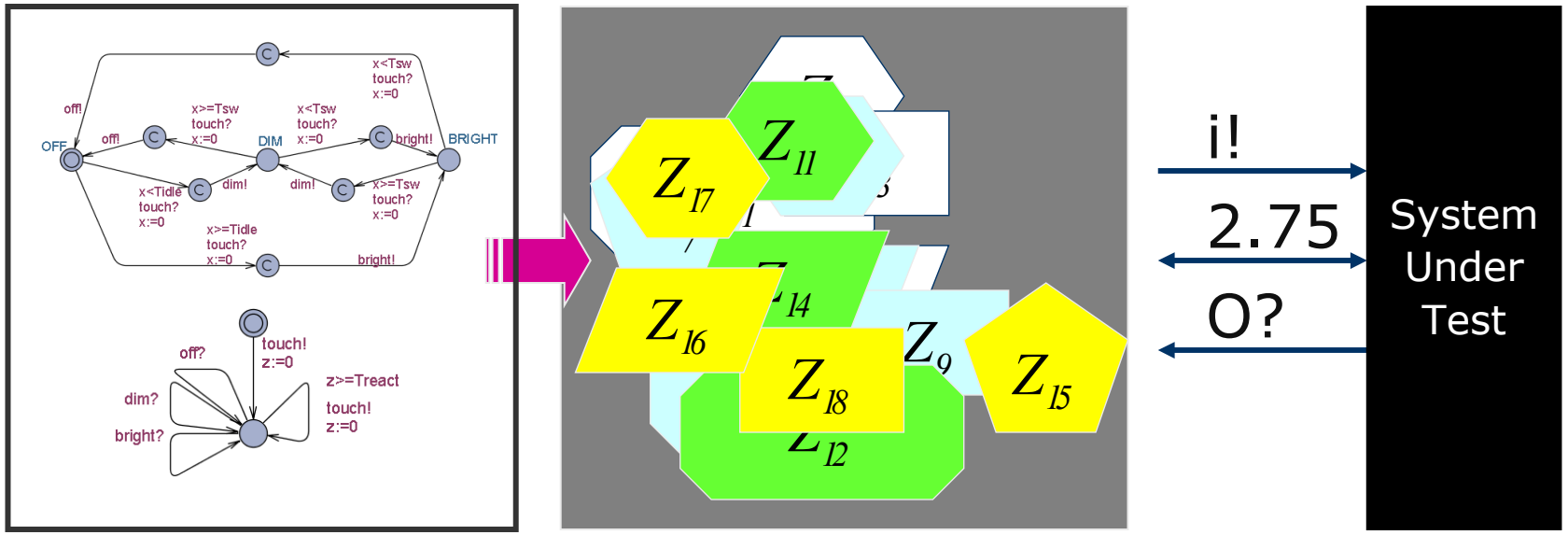
- *i is a relevant input in Env iff I ∈ EnvOutput(Z)*

Online State Estimation

Timed Automata Specification

State-set explorer:

maintain and analyse a *set* of *symbolic* states in real time!



(Abstract) Online Algorithm

Algorithm *TestGenExe (S, E, IUT, T)* **returns** {**pass**, **fail**)

Z := {(s0, e0)}.

while Z ≠ ∅ **and** ♯iterations ≤ T **do either** randomly:

1. // offer an input

   if *EnvOutput(Z)* ≠ ∅

   randomly choose *i*∈ *EnvOutput(Z)*

   send i to IUT

   Z := Z *After* i

1. // wait d for an output

   randomly choose d∈ *Delays(Z)*

   wait (for d time units or output o at d′ ≤ d)

   if o occurred **then**

   Z := Z *After* d′

   Z := Z *After* o // may become ∅ (⇒fail)

   else

   Z := Z *After* d // no output within d delay

3. *restart*:

   Z := {(s0, e0)}, **reset** IUT //reset and restart

if Z = ∅ **then return fail else return pass**

Center for Indlejrede Software Systemer

(Abstract) Online Algorithm

Algorithm *TestGenExe (S, E, IUT, T)* **returns** {**pass**, **fail**)
Z := {(s0, e0)}.

while Z ≠ ∅ ∧ ♯iterations ≤ T **do either** randomly:

1. // offer an input

   if *EnvOutput(Z)* ≠ ∅
   randomly choose *i* ∈ *EnvOutput(Z)*
   send i to IUT
   Z := Z *After*

1. // wait d for an

   randomly choos

   wait (for d time

   if o occurred **th**

   Z := Z *After*

   Z := Z *After* o // may become ∅ (⇒fail)

   else

   Z := Z *After* d // no output within d delay

3. *restart*:

   Z := {(s0, e0)}, **reset** IUT //reset and restart

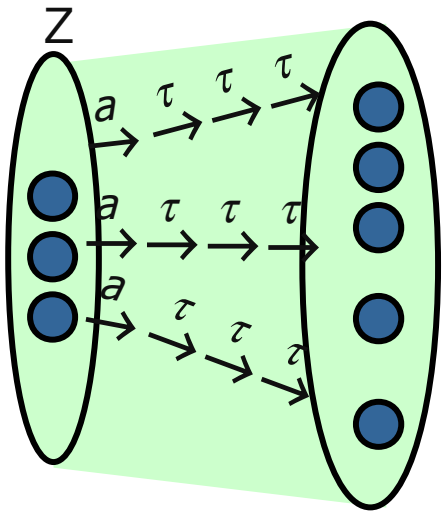
if Z = ∅ **then return fail else return pass**

- Sound (fail => non-conforming)
- Complete (non-conforming => fail as T → ∞, (under some technical assumptions))

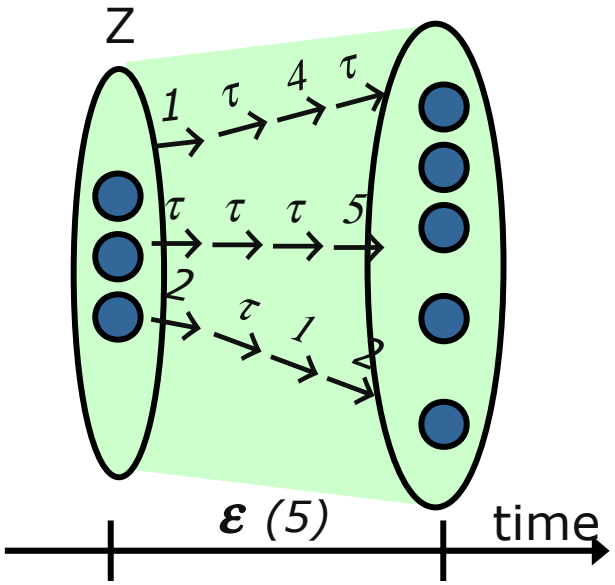
Center for Indlejrede Software Systemer

State-set Operations

Z* after *a: *possible states after* ***action a*** *(and* τ^**)*

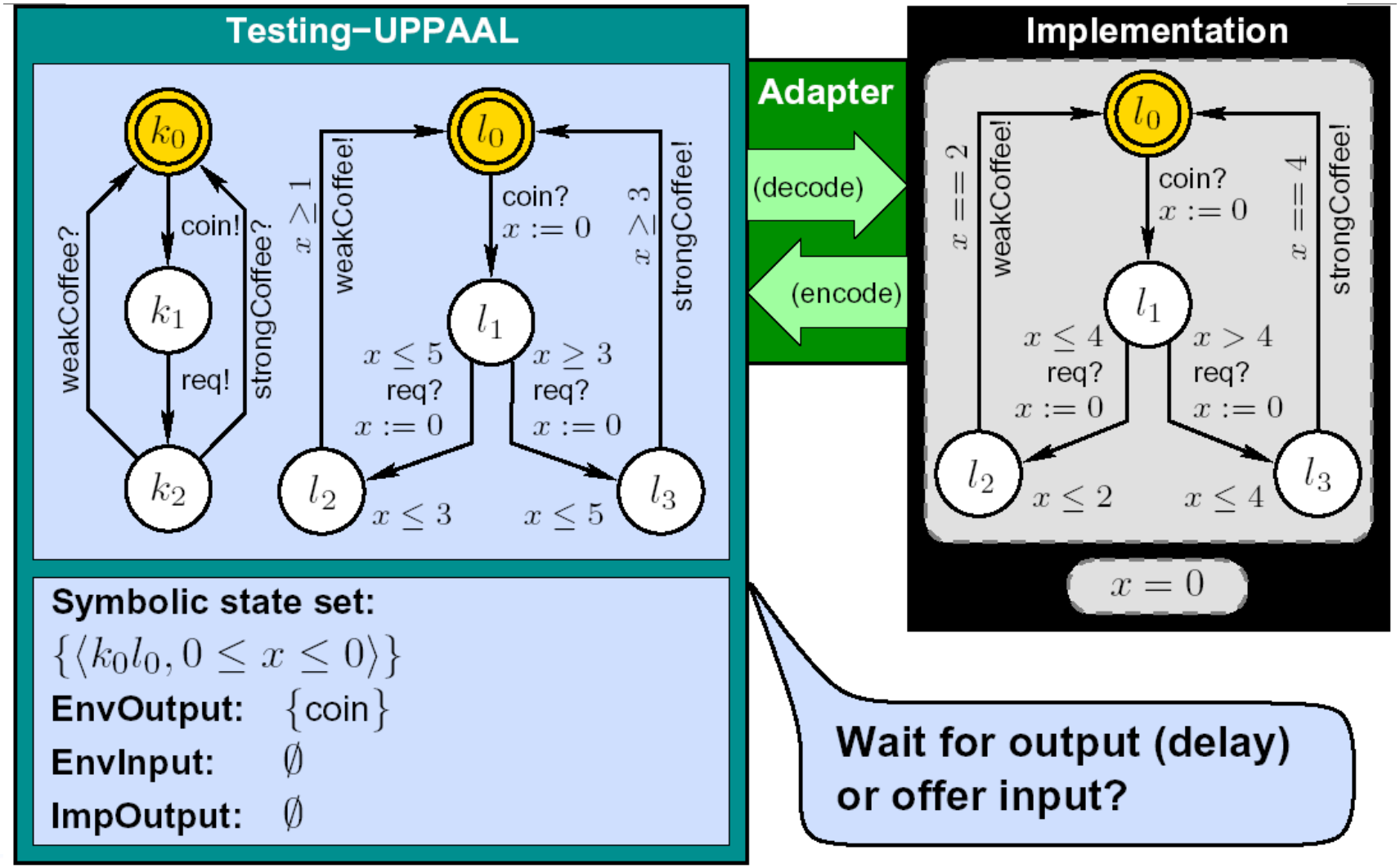


Z* after ε** :*possible states after* τ^* *and* ε_i*, totaling a* ***delay *of* ε

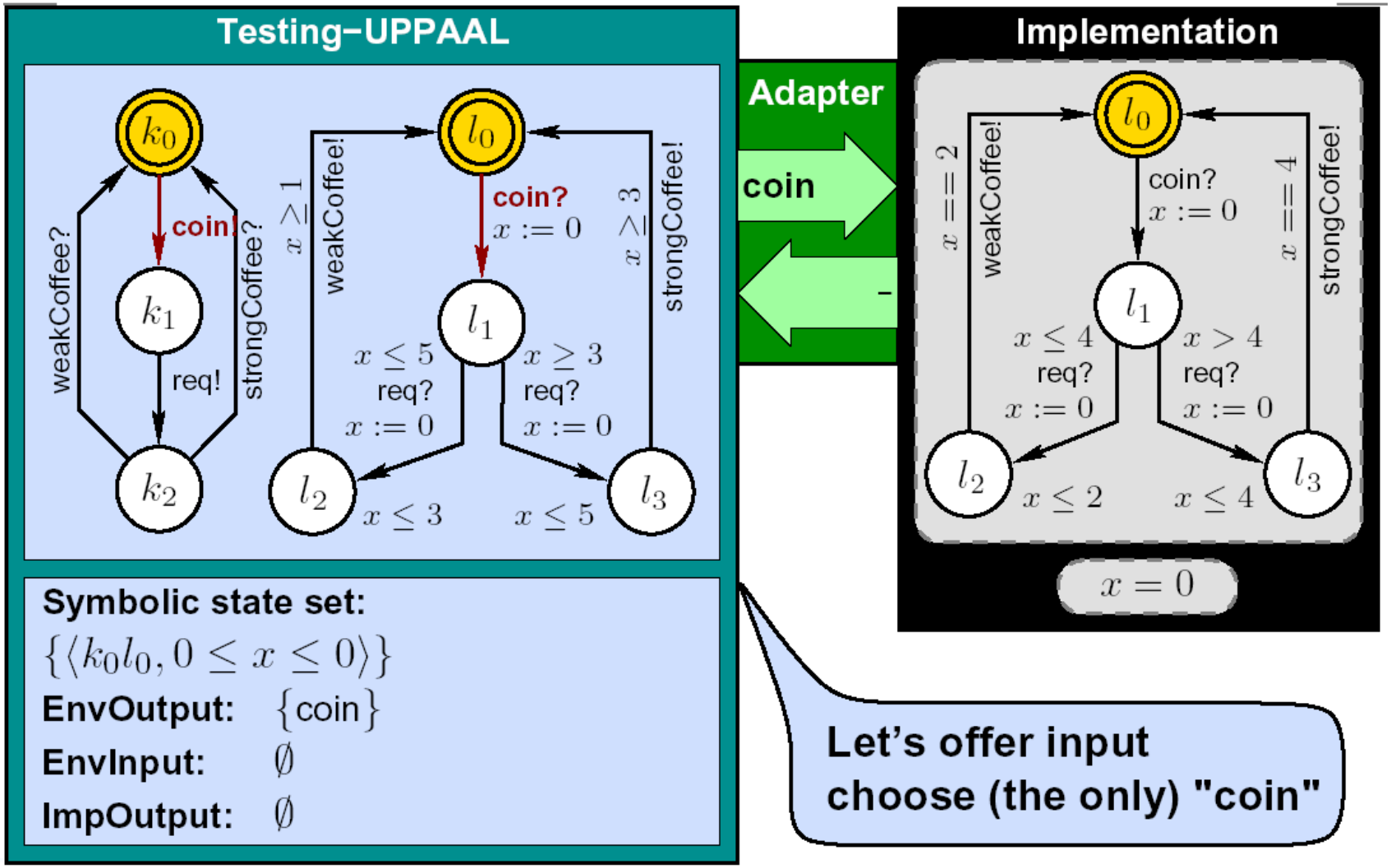


- Can be computed efficiently using the symbolic data structures and algorithms in Uppaal

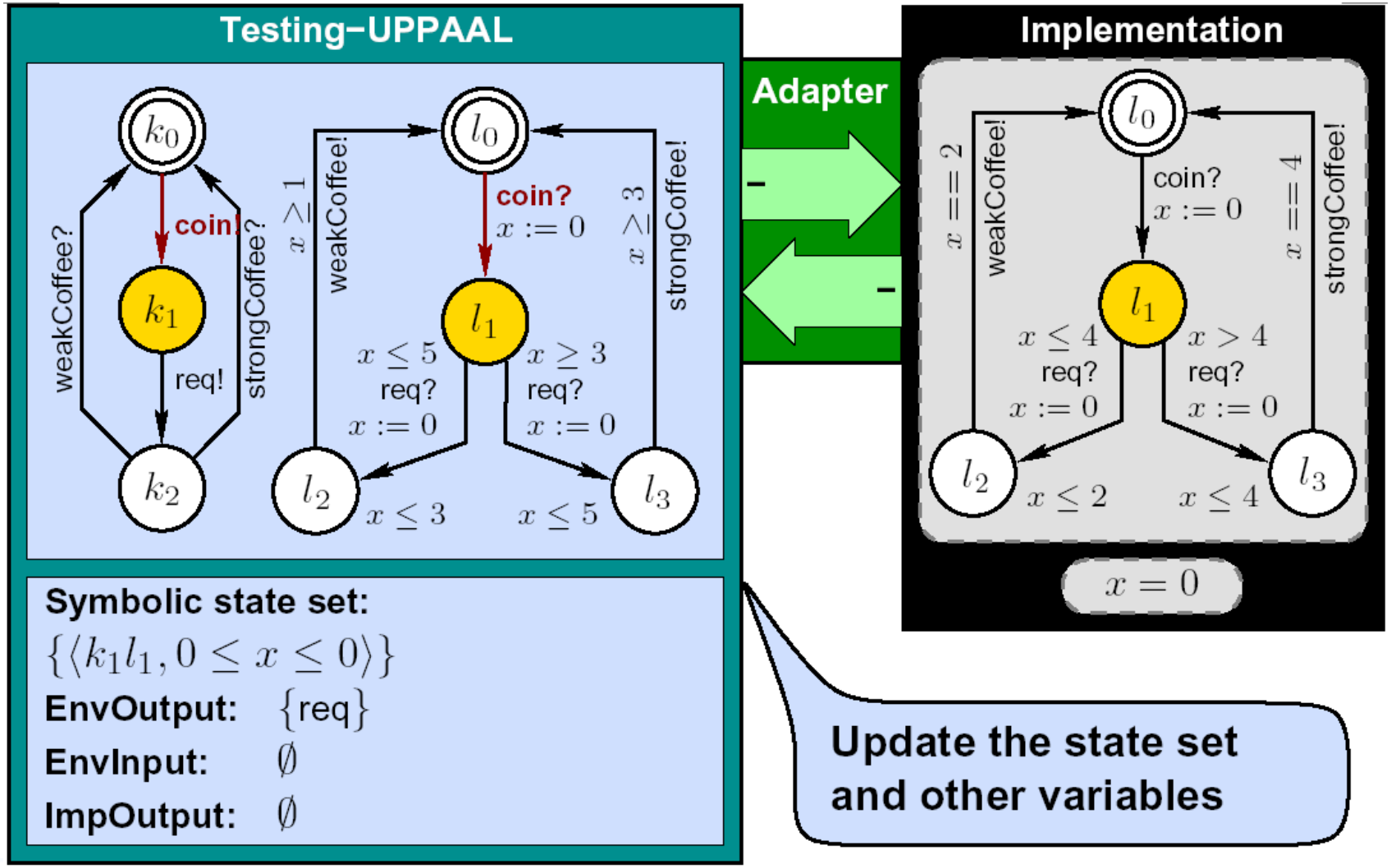
Online Testing Example



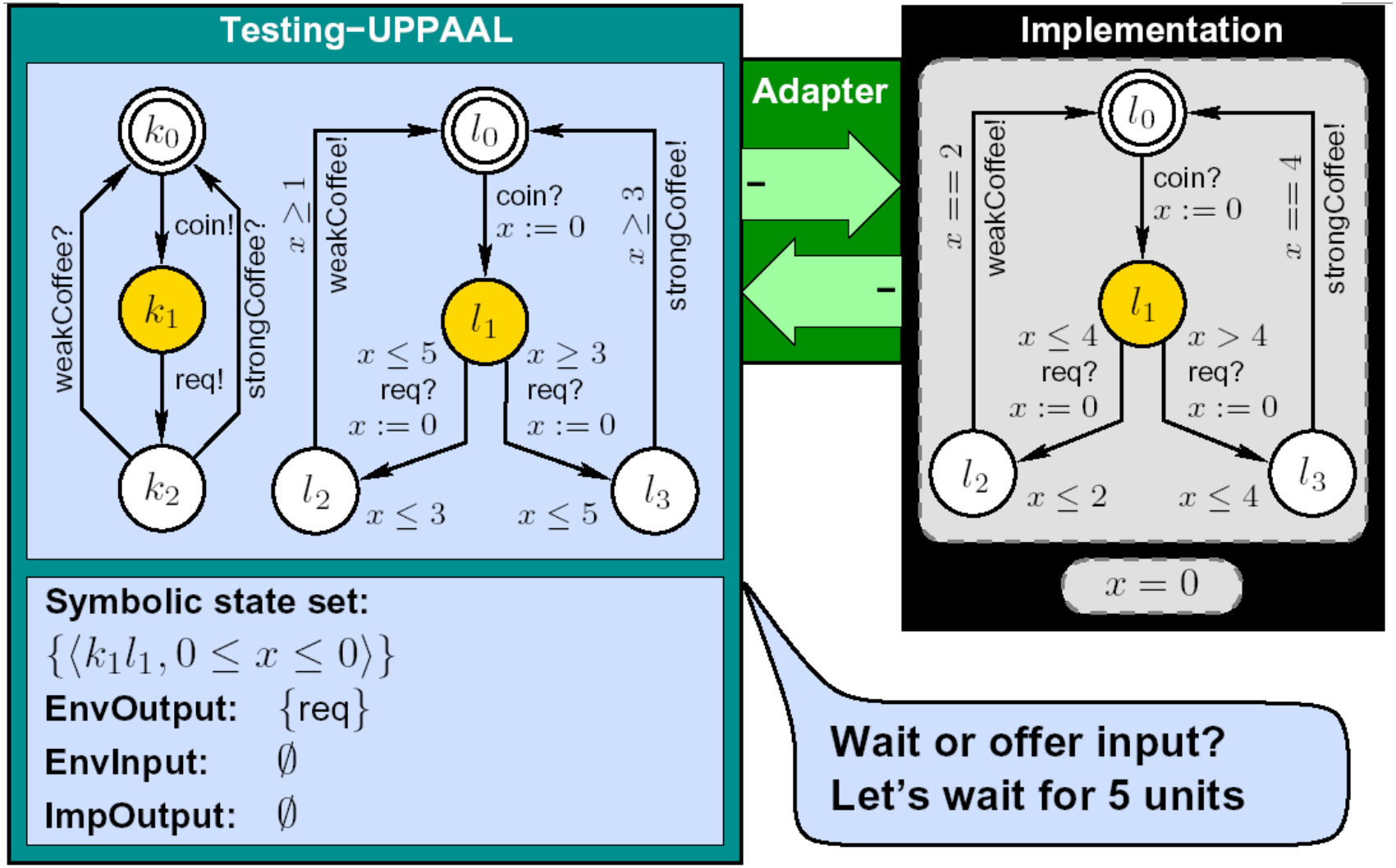
Online Testing



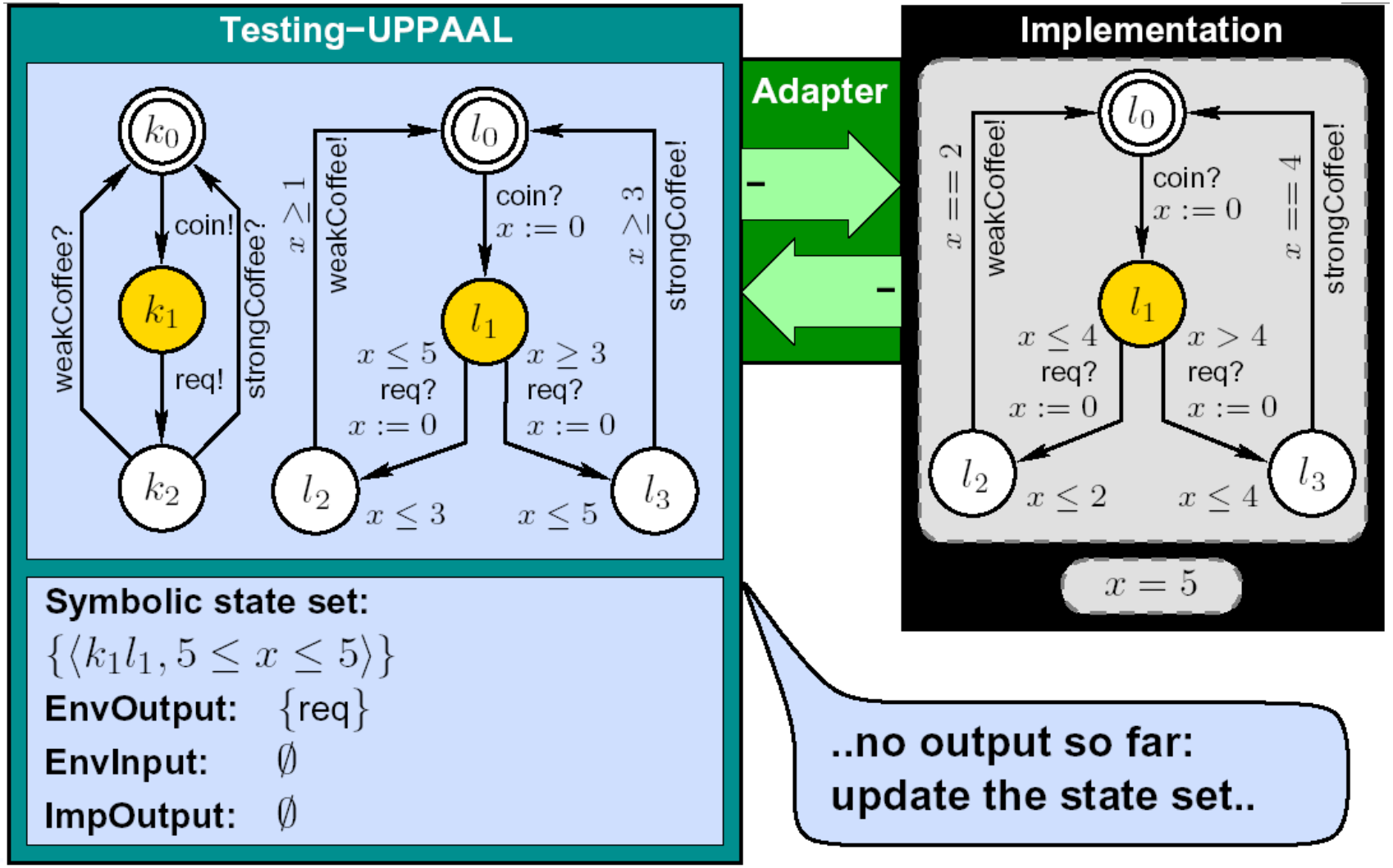
Online Testing



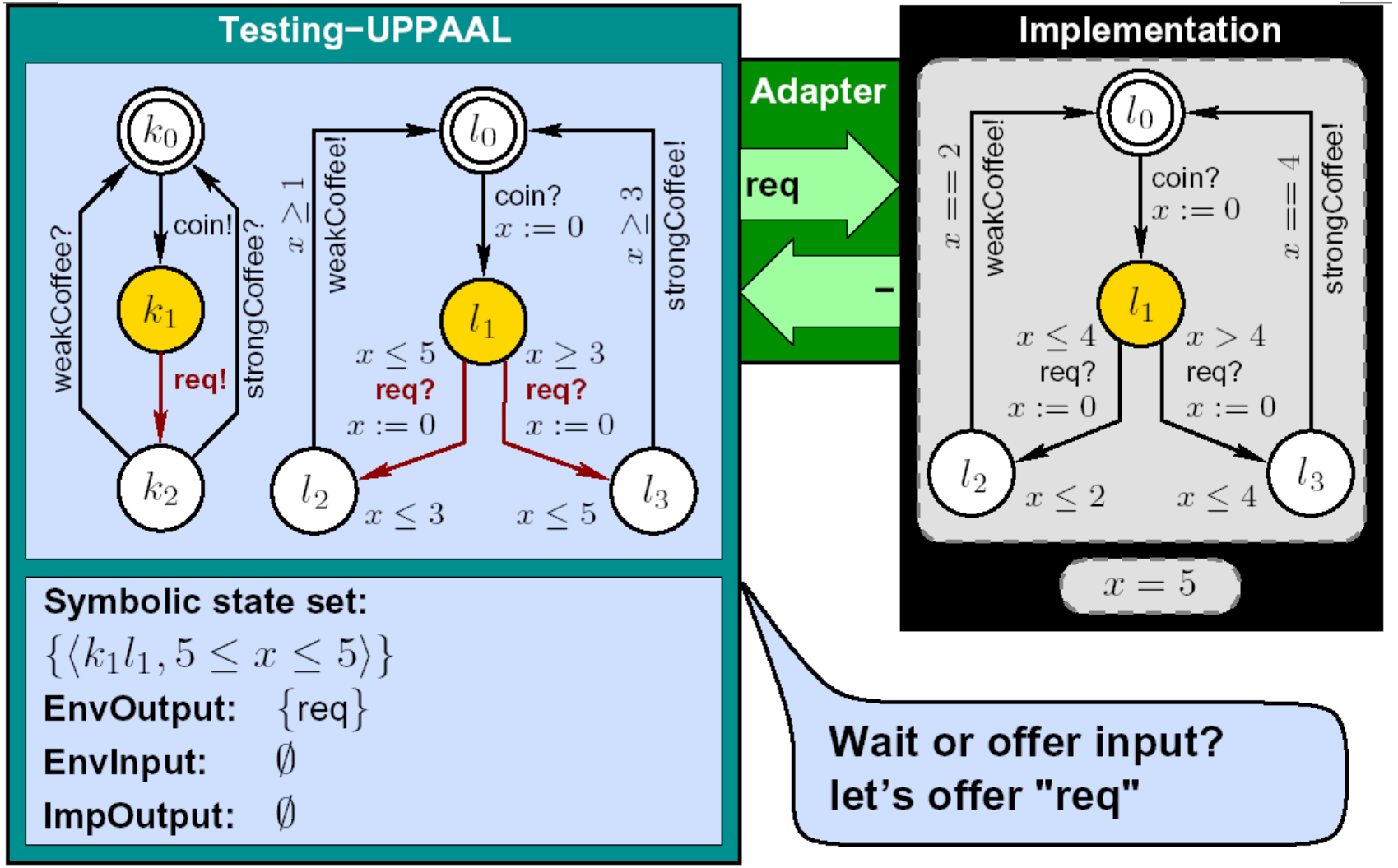
Online Testing



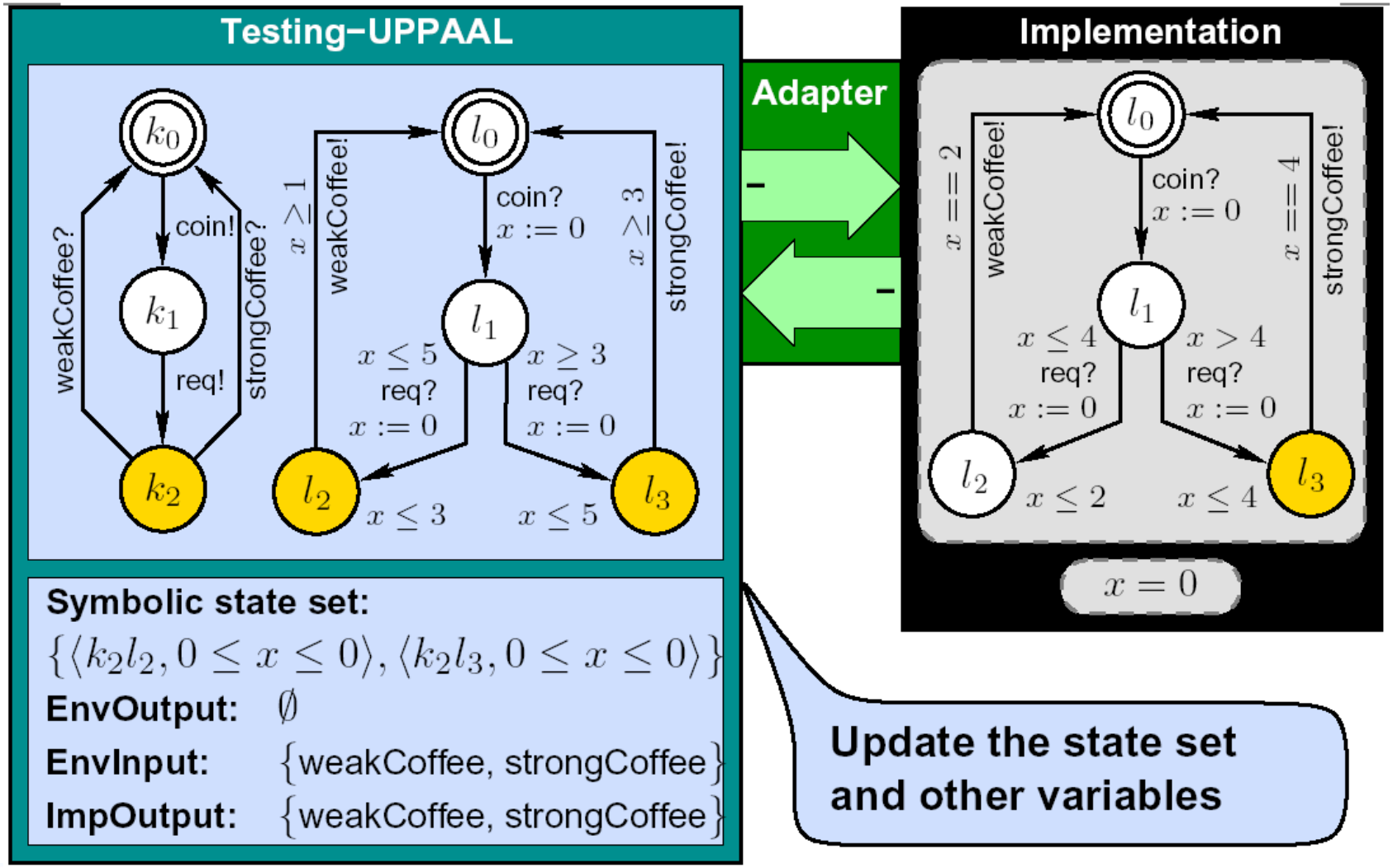
Online Testing



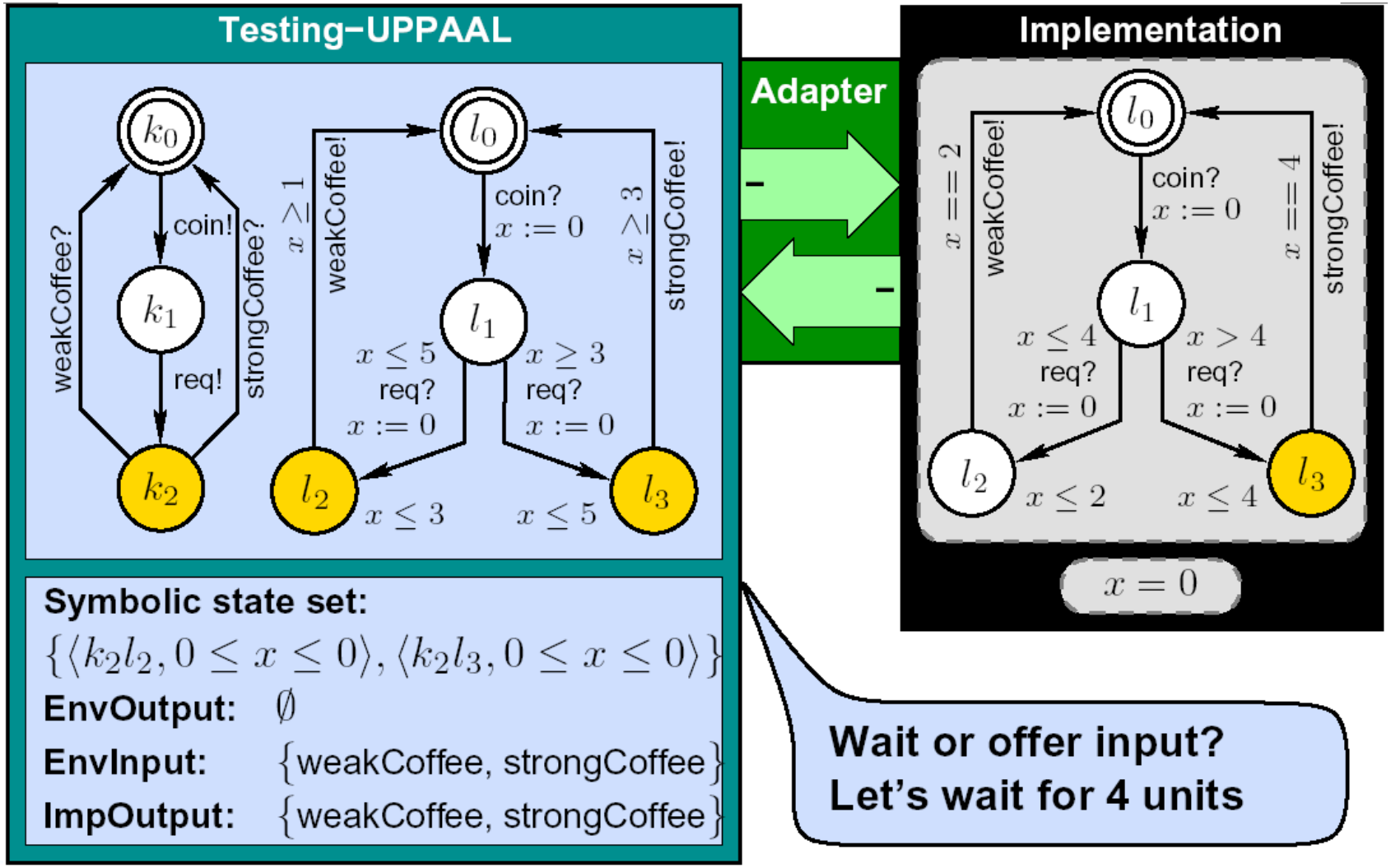
Online Testing



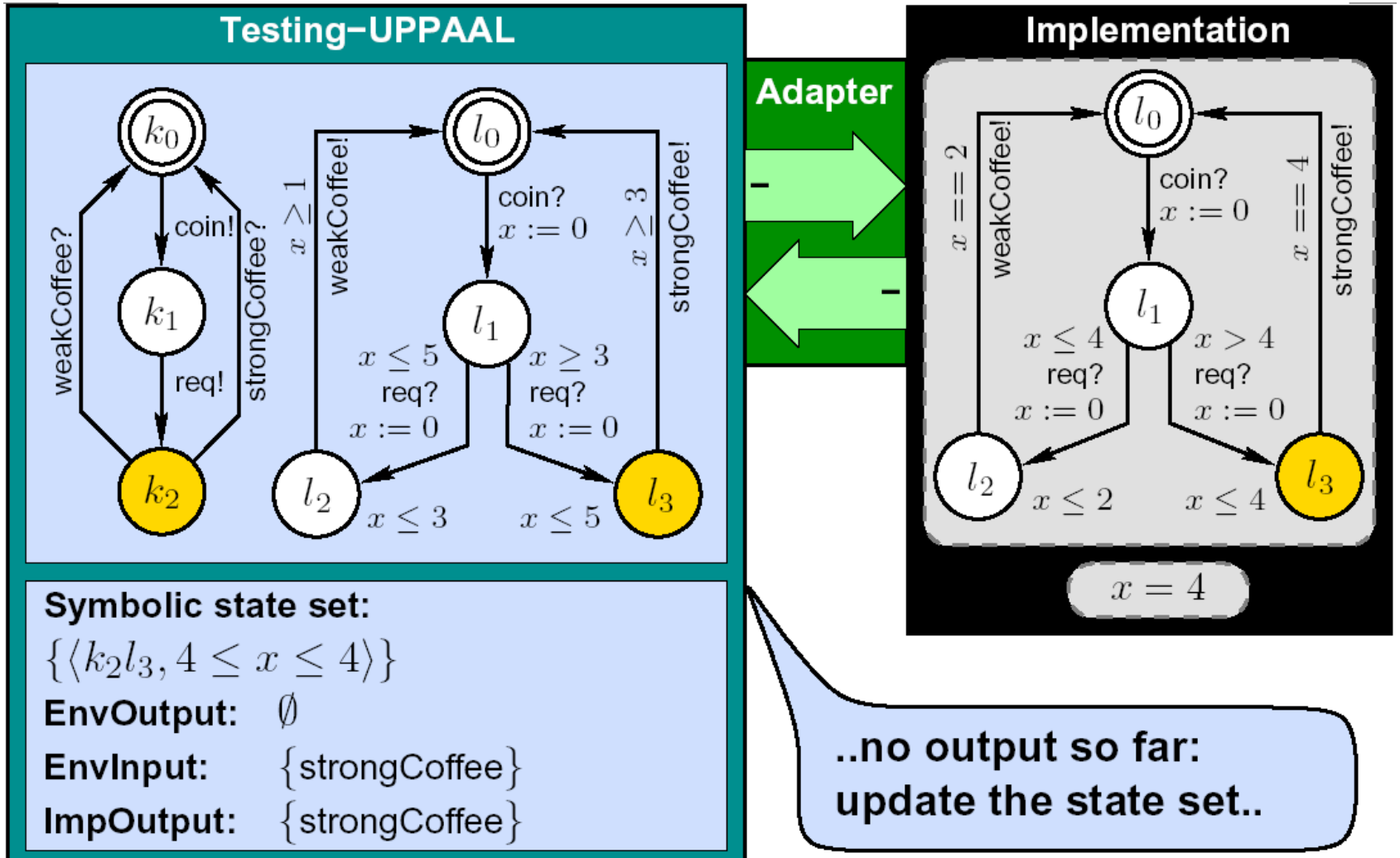
Online Testing



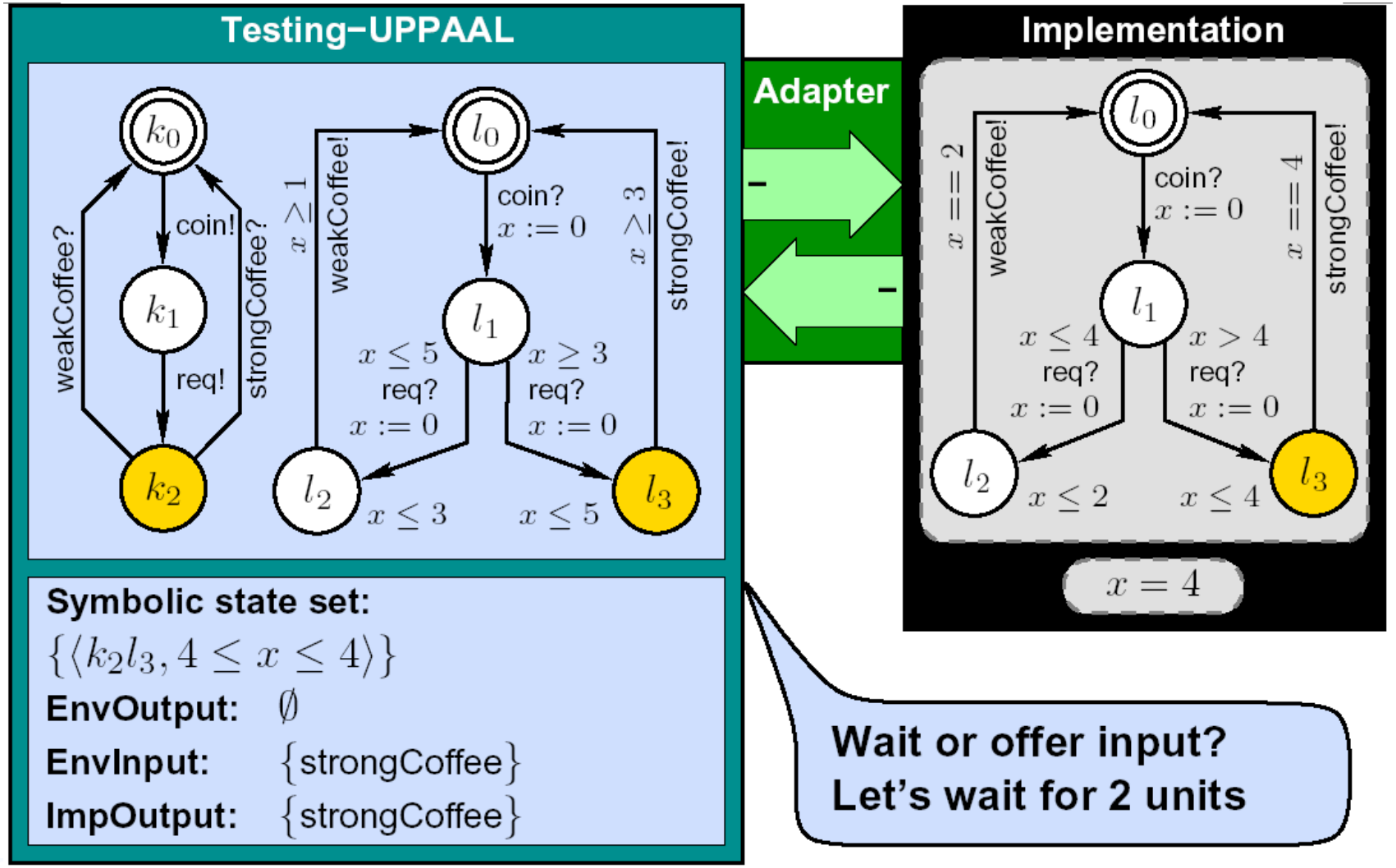
Online Testing



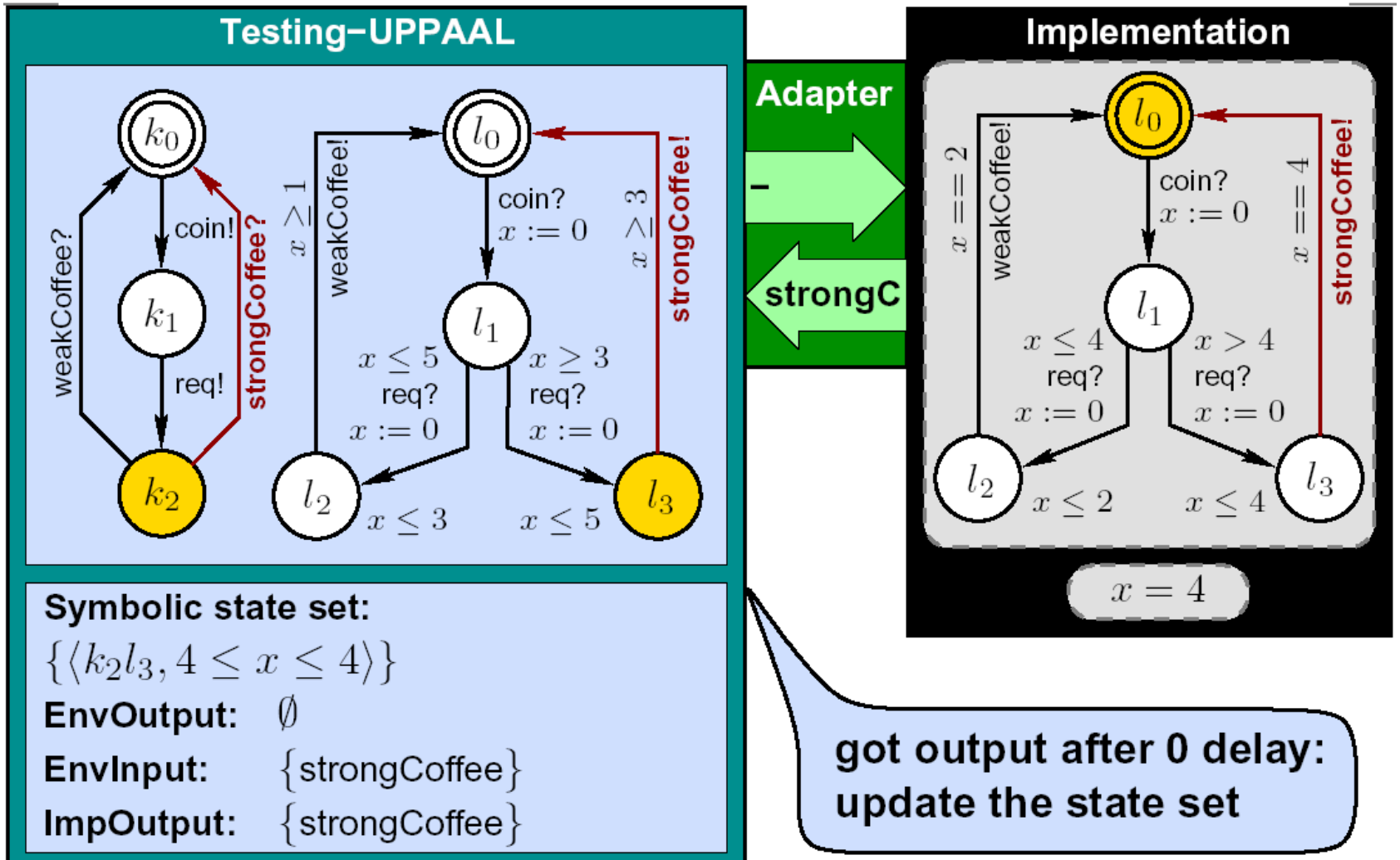
Online Testing



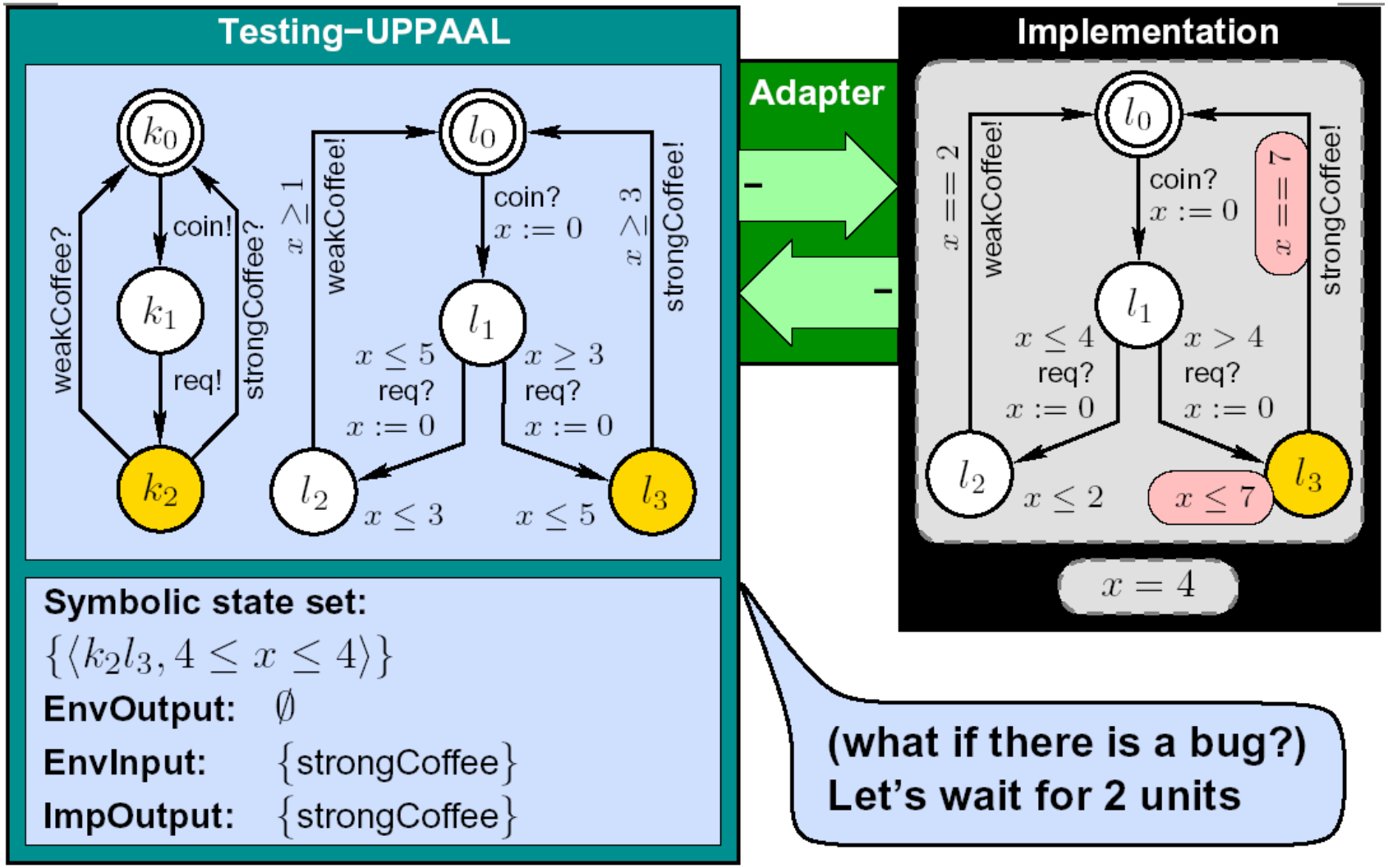
Online Testing



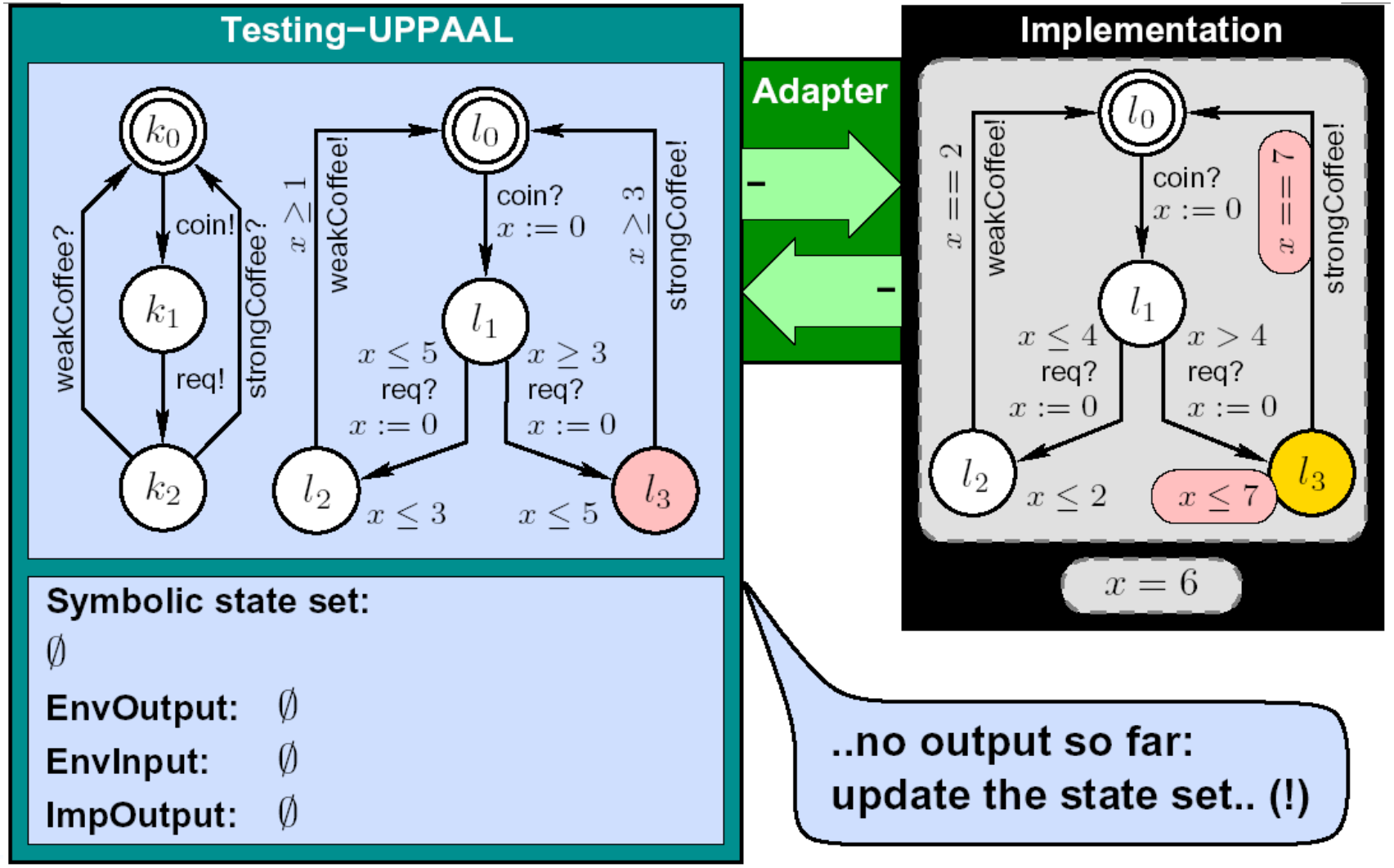
Online Testing



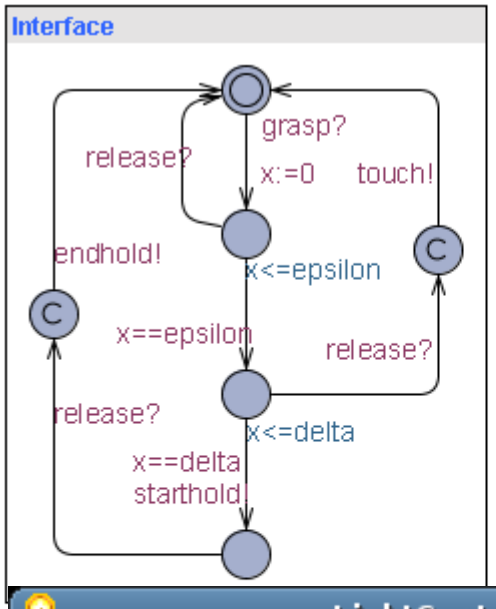
Online Testing

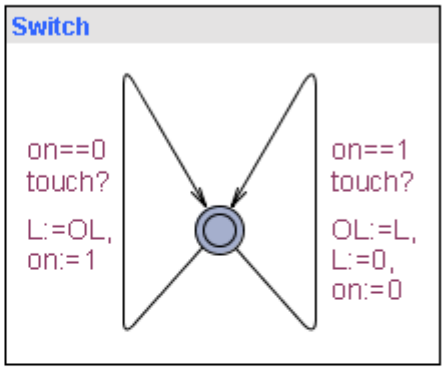


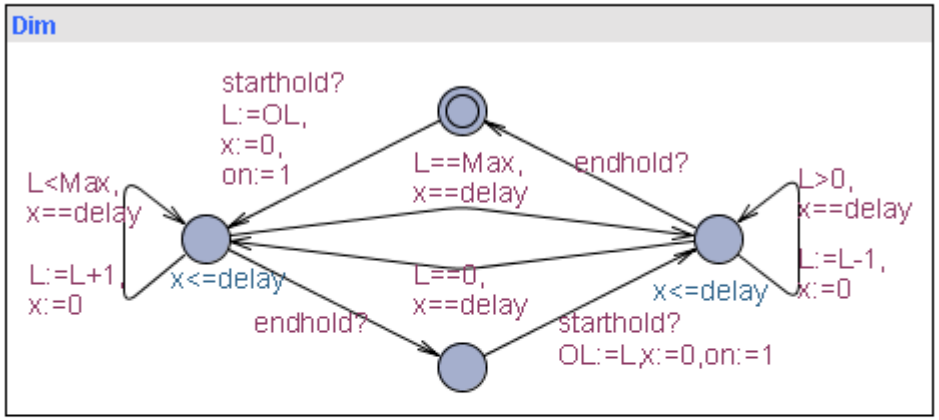
Online Testing

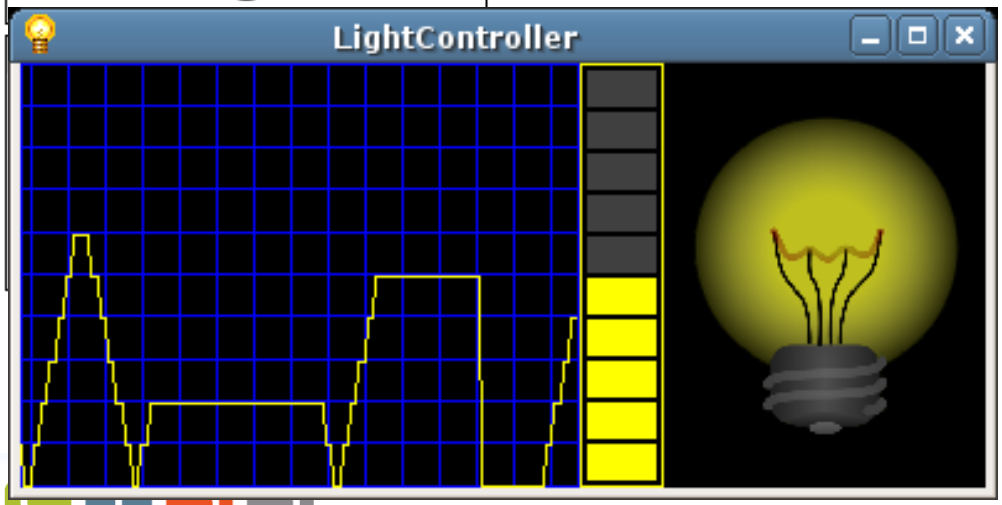


DEMO Touch-sensitive Light-Controller



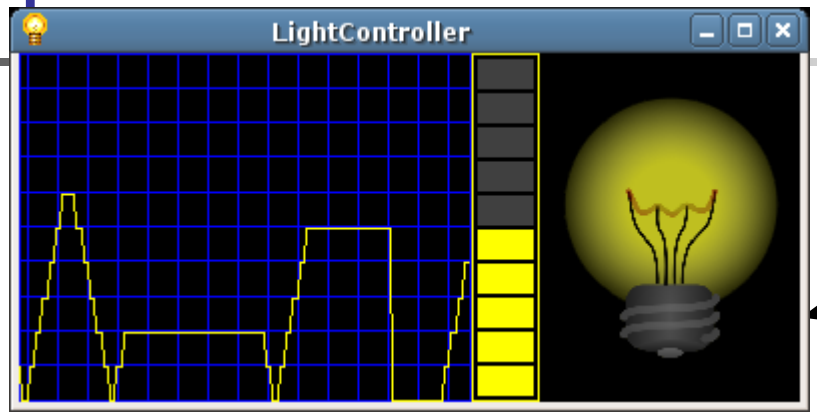


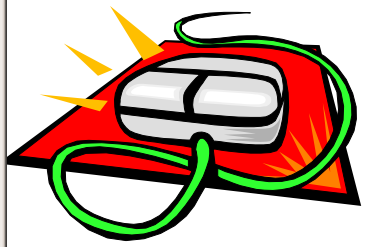




Center for Indlejrede Software Systemer

Test Setup





mousePress
mouseRelease

TRON
Testing
Host

tcp/ip

- Real-time
- Simulated time

Test
Fixture

LightControllerGUI

setLevel

grasp
release

setLevel

grasp
release

LightController

JavaVM+w2k/Linux

Center for Indlejrede Software Systemer

Mutants

- Mutant: Non-conforming program version with a seeded error
  - M1 incorrectly implements switch

```
synchronized public void handleTouch() {
    if(lightState==lightOff) {
      setLevel(oldLevel);
      lightState=lightOn;
    }
     else { //was missing
   if(lightState==lightOn){
     oldLevel=level;
     setLevel(0);
     lightState=lightOff;
  }
```

  - M2 violates a deadline

Industrial Cooling Plants

Industrial Application:

Danfoss Electronic Cooling Controller

Sensor Input

- air temperature sensor
- defrost temperature sensor
- (door open sensor)

Keypad Input

- 2 buttons (~40 user settable parameters)

Output Relays

- compressor relay
- defrost relay
- alarm relay
- (fan relay)

Display Output

- alarm / error indication
- mode indication
- current calculated temperature

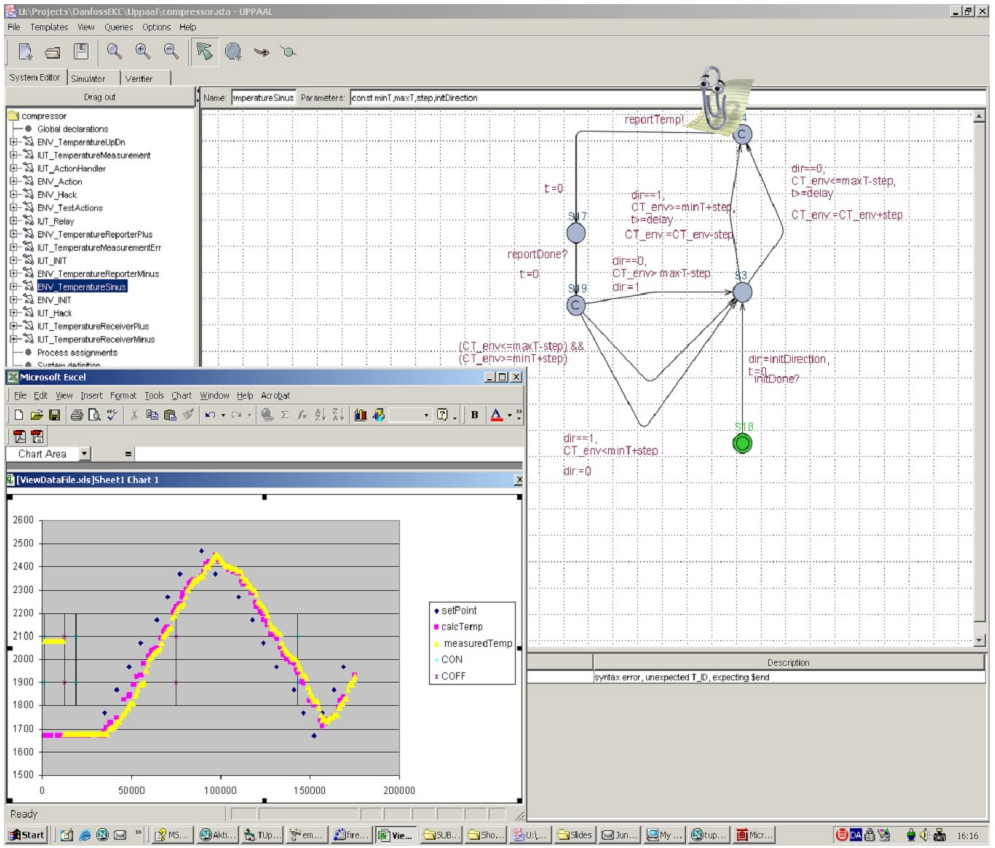
- Optional real-time clock or LON network module

Industrial Application:

Danfoss Electronic Cooling Controller

Sensor Input

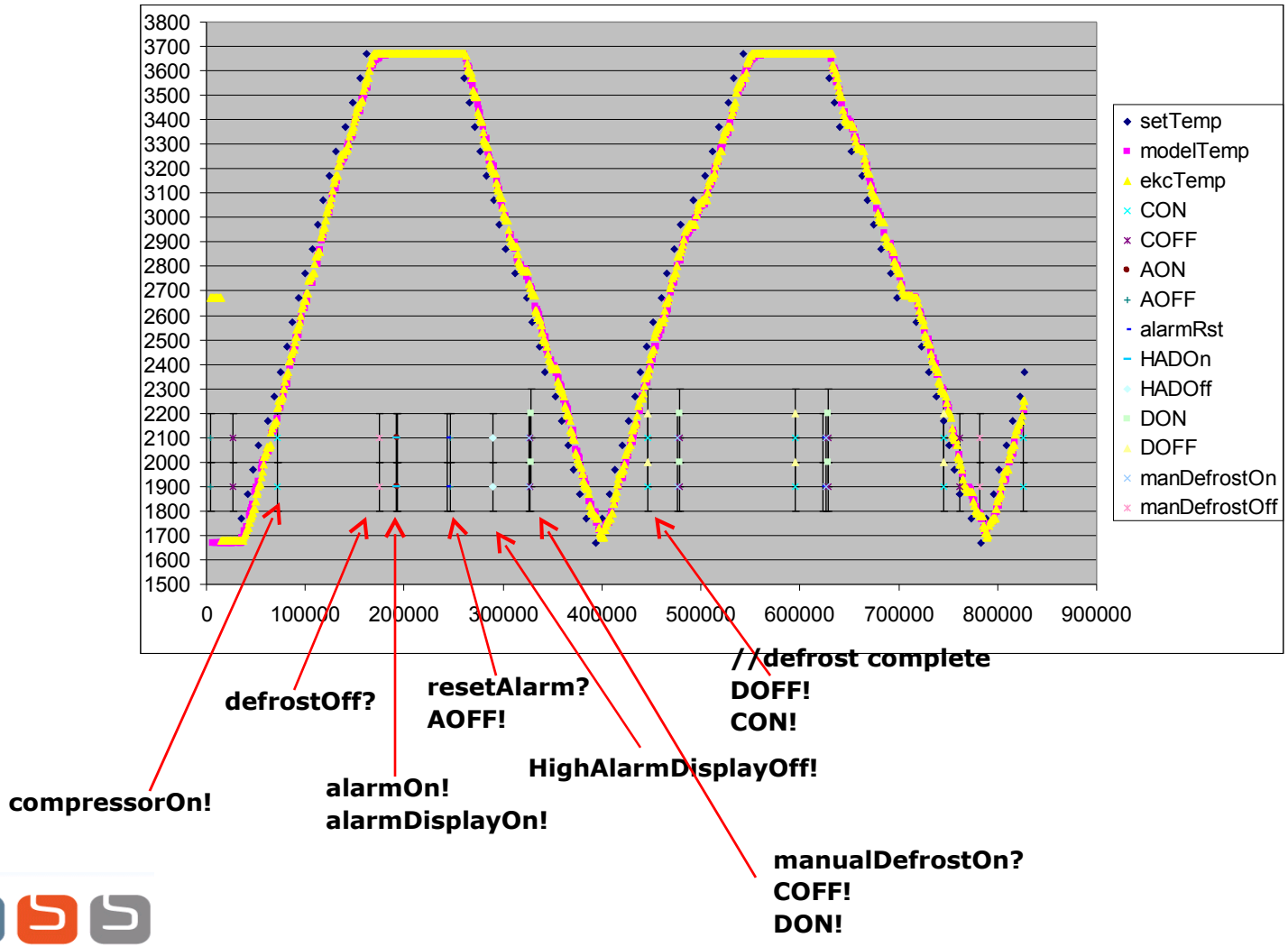
- air temperature sensor



- Optional real-time clock or LON network module

Example Test Run

(log visualization)



Center for Indlejrede Software Systemer

Model-based Testing of Real Time Systems

Conclusions

BRICS
Basic Research
in Computer Science

Conclusions

- Testing real-time systems is theoretically and practically challenging
- Promising techniques and tools
- Explicit environment modeling
  - Realism and guiding
  - Separation of concerns
  - Modularity
  - Creative tool uses
  - Theoretical properties
- Real-time online testing from timed automata is feasible, but
  - Many open research issues

Research Problems

- Testing Theory
- Timed games with partial observability
- Hybrid extensions
- Other Quantitative Properties
- Probabilistic Extensions, Performance testing
- Efficient data structures and algorithms for state set computation
- Diagnosis & Debugging
- Guiding and Coverage Measurement
- Real-Time execution of TRON
- Adapter Abstraction, IUT clock synchronization
- Further Industrial Cases

Related Work

- Formal Testing Frameworks
  - [Brinksma, Tretmans]
- Real-Time Implementation Relations
  - [Khoumsi'03, Briones'04, Krichen'04]
- Symbolic Reachability analysis of Timed Automata
  - [Dill'89, Larsen'97,…]
- Online state-set computation
  - [Tripakis'02]
- Online Testing
  - [Tretmans'99, Peleska'02, Krichen'04]

Uppaal TRON Tutorial

modeling, implementing and testing

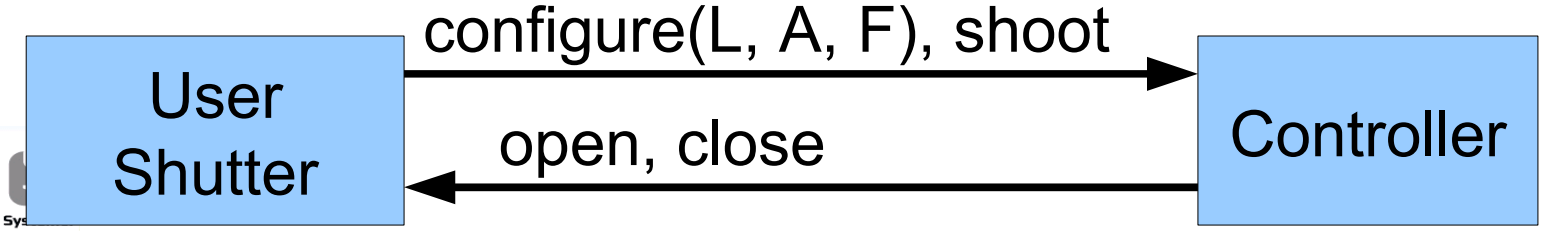
BRICS
Basic Research
in Computer Science

Test the setup

- Check Java version by "java -version"
  - Should be Sun Java 6 or 5
- Check ant version by "ant -version"
  - Should be 1.7.0 or 1.7.1
- Unpack (copy) Uppaal TRON 1.5 distribution
- Go to "java" directory of Uppaal TRON distribution
- Recompile:
  - "ant clean jar"
- Run test of smartlamp:
  - Linux: "start-light-v.sh & start-test-v.sh"
  - Windows: "start-test-v.bat"
- Try commands from Makefile (Linux only):
  - "make test-light-s"
  - "make test-dummy-s"

Task: model, implement, test

- Simple shutter controller for photo camera:
  - When user presses “configure” sensors are read:
    - luminance (L): 10^0 – 10^5 lux
    - aperture (A): 1.41, 2, 2.83, 4, 5.66, 8, 11.31, 16, 22.63, 32
    - focal length (F): 0.05m
  - Controller then computes exposition (E): 1ms - 2s
    - E=3.14*(10^2)*((F/2*A)^2)/L (in seconds)
  - When user presses “shoot”:
    - Shutter should get signal “open” within 1ms
    - Shutter should remain open for at least E-0.1ms time
    - Shutter should get signal “close” within E time after “open”



Test Specification

- Run tests at least against three environments:
  - Human behavior:
    - Slow user, use clock guards to slow down user
    - All tests should pass
  - Extreme but still realistic behavior:
    - Fast user, use clock invariants to force more inputs
    - Test may fail:
      - Find out why test fail
      - Advanced: find boundary conditions that test does not fail.
  - Unrestricted:
    - Allow any input at any time
    - It is OK for tests to fail, explain why tests fail.

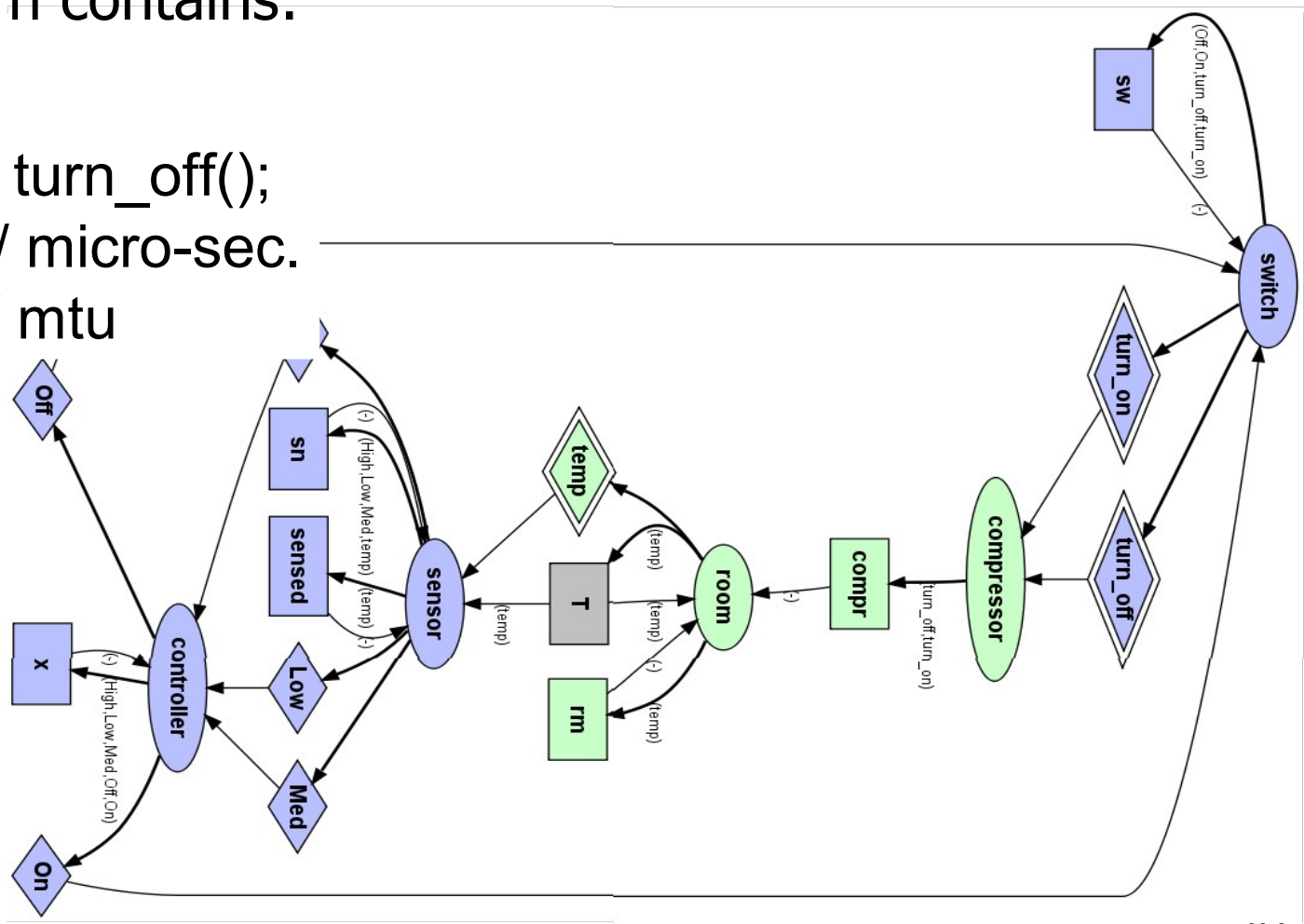
Tips for Modeling

- The model consists of a closed system:
  - All sending and receiving synchronizations are paired: for every sender there is a receiver.
- Model is partitioned into:
  - Processes for environment
  - Processes for implementation
  - All communication between env. and IUT goes only with observable channel synchronization
  - Adapter processes belong to implementation
- Use "tron -i dot" option to get signal flow diagram

Model partition example

- tron -v 3 -i dot fridge.xml < fridge.trn | dot -Tpng -o fridge.png
- Where fridge.trn contains:

```
input temp(T);
output turn_on(), turn_off();
precision 1000; // micro-sec.
timeout 10000; // mtu
```



Input Enabledness

- IUT model should be input enabled:
  - If not, diagnostics might be misleading
- Environment model should be input enabled:
  - If not, there might be “Inconclusive” verdicts
- Use loop edges with caution when consuming redundant observable actions

Tips for Programming in Java

- Use dummy example as a stub in java dir:
  - see src/com/uppaal/dummy/Dummy.java
  - execute() method is the implementation code
- DummyInterface.java declares inputs
- DummyListener.java declares outputs
- TestIOHandler.java handles the communication:
  - configure(Reporter) registers observable channels
  - run() translates tester's input messages to Dummy calls
  - reportMyOutput() translates calls into output messages to tester

Thread programming

- Monitor synchronization:

Producer:

```
…
lock.lock();
queue.add(item);
cond.signalAll();
lock.unlock();
...
```

Consumer:

```
…
lock.lock()
while (queue.isEmpty())
        cond.await();
queue.remove();
lock.unlock();
…
```

- Consumer and producer share queue
- queue is protected by lock
- Consumer blocks and releases lock in await if queue is empty
- cond is associated and surrounded with lock
- cond wakes consumer up which reacquires lock and continues

Testing using Uppaal TRON

- TRON can be run in:
  - Real-world-time (requires adapter models, see smartlamp)
  - Virtual time (provides "lab-conditions" and releases from scheduling and communication latency problems)
- Virtual time (option "-Q"):
  - Uses single virtual clock
  - All threads are registered at this clock (use VirtualThread instead of Thread)
  - All time-related calls are forwarded to this clock
  - Virtual clock is incremented only when all threads agree to wait
  - May deadlock if at least one thread misbehaves
  - Possible Zeno behavior (time does not progress)

Tips for running online tests

- If test fails and diagnostics at the end is not enough:
  - Inspect the last good state set provided by TRON:
    - Validate model:
      - Would **you** expect such state to be reachable?
      - No? How such state can be reached? Use UPPAAL verifier to find out.
    - Would/did implementation do as the model? Why not?
  - Add "System.out.print" in your implementation
  - Ask TRON for more verbose messages:
    - add "-D test.log" option to save test trace to test.log
    - use "-v 10" instead of "-v 9" to see engine events
- If test passes unexpectedly (e.g. TRON is too passive):
  - Use "-P" option with larger/smaller intervals, or set to "-P random"

More Details

- Options are documented in the user manual

  http://www.cs.aau.dk/~marius/tron/manual.pdf

- When all else fail...
  - See commands in Makefile rules
  - See smartlamp example source code
  - Write an email to: **marius@cs.aau.dk**

Thanks for your attention!