ARTIST Summer School in Morocco
Rabat, July 11-16th, 2010

Modeling, Verification and Testing of of Embedded Systems

Speaker : Brian Nielsen

Centre of Embedded Software Systems
Aalborg University, DK

Modeling, Verification, and Testing of of Embedded Systems

Brian Nielsen

Centre of
Embedded Software Systems
Aalborg University, DK
bnielsen@cs.aau.dk

Center for Indlejrede Software Systemer

Course Outline

1. Introduction
2. Modeling
   1. Modelling Embedded systems
   2. Introduction to timed automata (TA)
3. Verification using Uppaal
4. Beyond Verification: Synthesis
   1. Optimal Scheduling & Planning
   2. Controller Synthesis
5. Real-Time Conformance
   1. Testing theory
   2. Real-time extensions of the ioco testing theory
6. Real-Time Test Generation
   1. Off-line generation using model checkers
   2. (optimal) quantitative test-sequences (based on Priced TA)
   3. Online real-time testing
   4. Testing strategies using Timed Games
7. Conclusions

Slides available at
http://www.cs.aau.dk/~bnielsen/rabat2010.pdf

Software Embedded in Everything

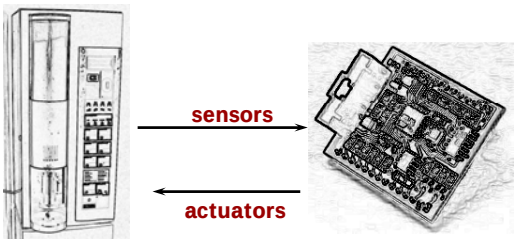
- 80% of all software is embedded
- Demands for
  - increased functionality
  - minimal resources
- Requires interdisciplinary skills
  - Software construction
  - hardware platforms,
  - Scheduling, and resource analysis
  - communication
  - **testing & verification**
- Complex, sometimes buggy
- International focus Area

Why Verification and Testing

- **IMPORTANCE for EMBEDDED SYSTEMS**
  - Often safety critical
  - Often economical critical
  - Hard to patch

- **CHALLENGES for EMBEDDED SYSTEMS**
  - Correctness of embedded systems depend crucially on use of *resources* (real-time, memory, bandwidth, energy).
  - Need for verification of and conformance testing with respect to *quantitative aspects*.

Real-time Systems



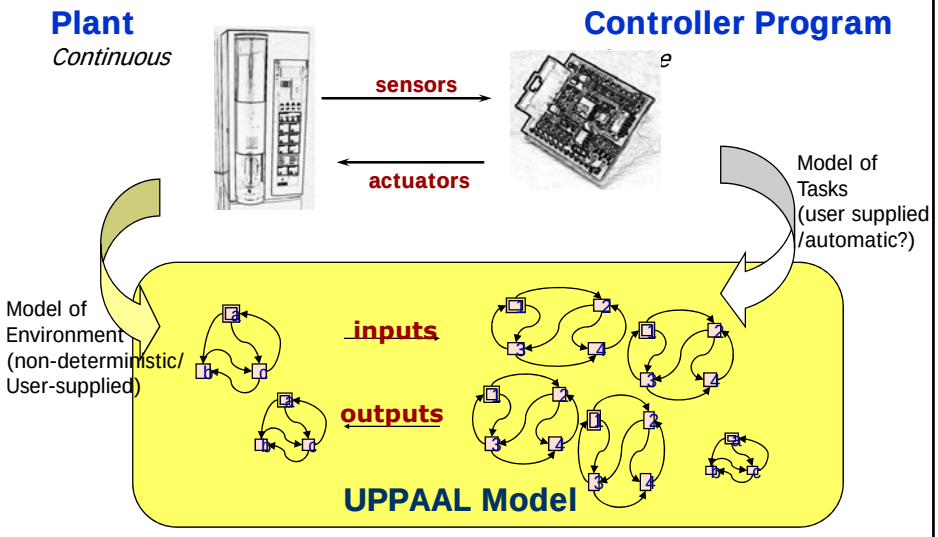
Plant/Env
Continuous

Controller Program
Discrete

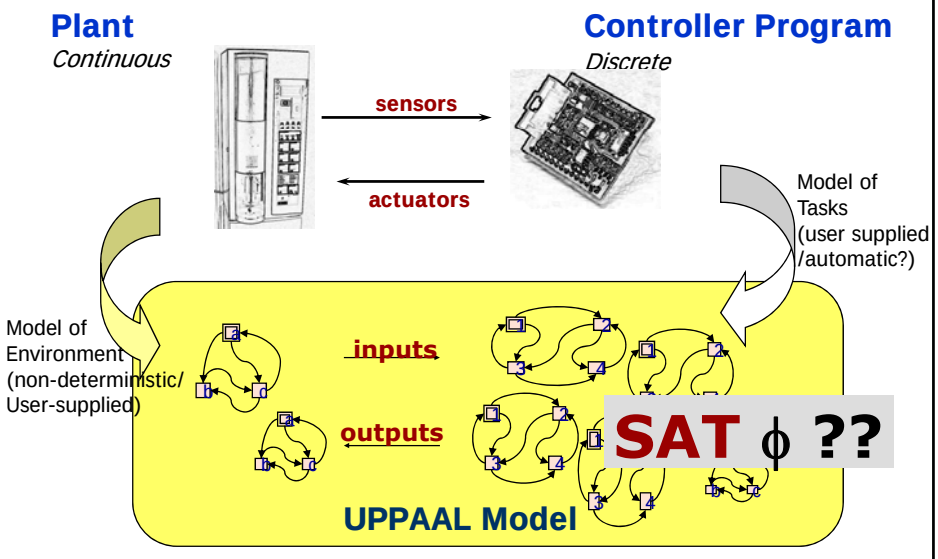
Eg.:
Realtime Protocols
Pump Control
Air Bags
Robots
Cruise Control
ABS
CD Players
Production Lines

Real Time System
A system where correctness not only depends on the logical order of events but also on their **timing**!!

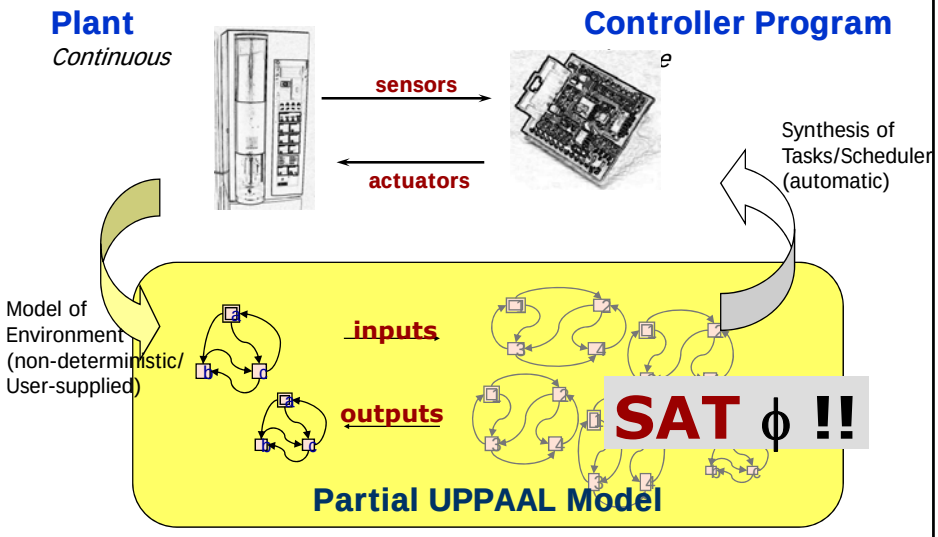
Real-time Modeling



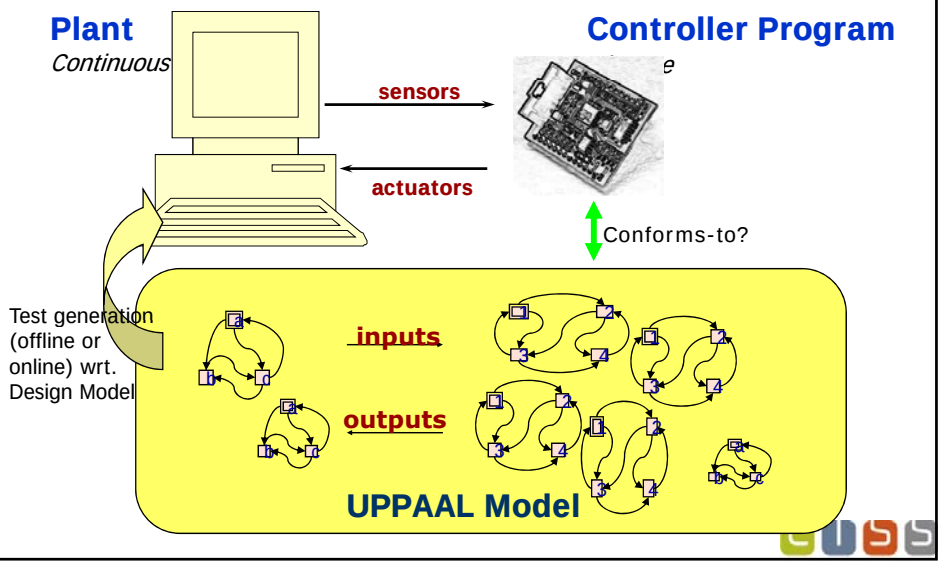
Real-time Model-checking



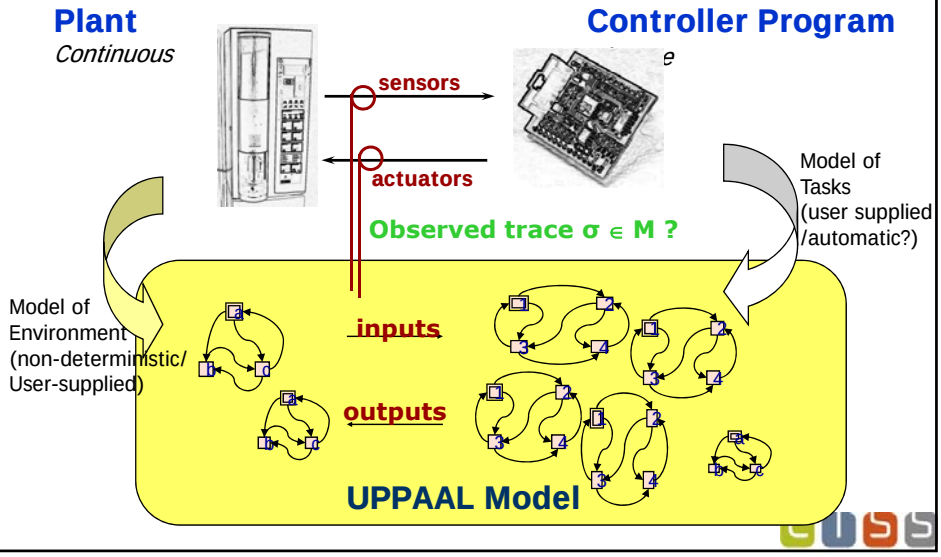
Real-time Controller Synthesis



Real-time Model-Based Testing



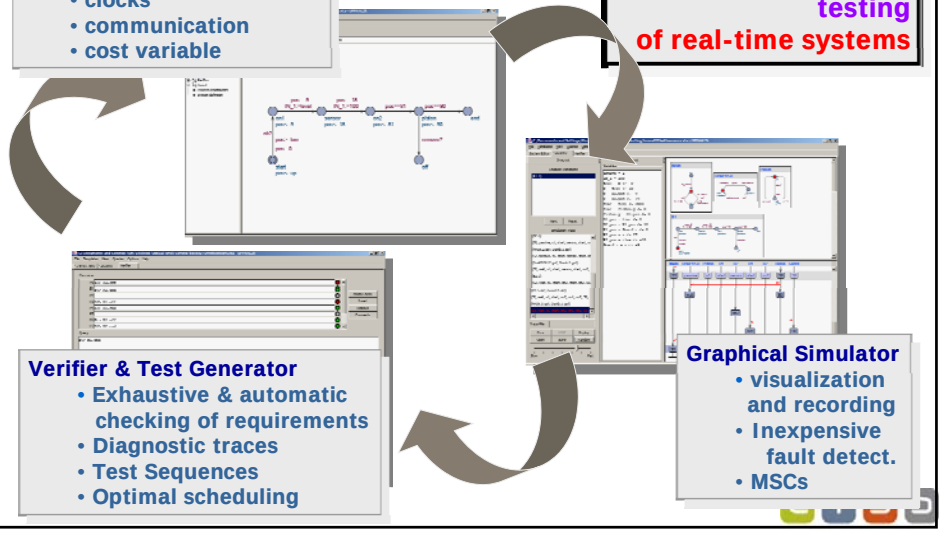
Real-time Monitoring



UPPAAL

Graphical Design Tool
- timed automata
- datatypes & functions
- clocks
- communication
- cost variable

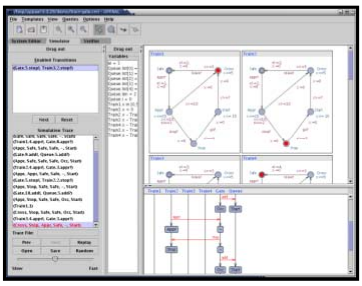
Tool Environment for modeling, simulation, verification, optimization & testing of real-time systems



Verifier & Test Generator
- Exhaustive & automatic checking of requirements
- Diagnostic traces
- Test Sequences
- Optimal scheduling

Graphical Simulator
- visualization and recording
- Inexpensive fault detect.
- MSCs

UPPAAL Tools www.uppaal.com



Uppaal Model-checker:
Efficient reachability analysis of network of timed automata

Uppaal-CORA | Uppaal-TIGA | Uppaal-TRON | ...

- TIGA: Timed games (reachability and safety)
- CORA: Cost Optimal reachability from priced TA
- TRON: Testing Real-time Online

UPPAAL Team

@UPPsala

- **Wang Yi**
- **Paul Pettersson**
- John Håkansson
- Anders Hessel
- Pavel Krcal
- Leonid Mokrushin
- Shi Xiaochun

@AALborg

- **Kim G Lars**
- **Alexandre I**
- **Marius Mikucionis**
- Gerd Behrman
- Arne Skou
- Brian Nielsen
- Jacob I. Rasmussen
- Thomas Chatain

@Elsewhere

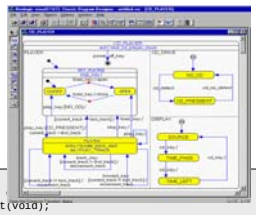
- Emmanuel Fleury, Didier Lime, Johan Bengtsson, Fredrik Larsson, Kåre J Kristoffersen, Tobias Amnell, Thomas Hune, Oliver Möller, Elena Fersman, Carsten Weise, David Griffioen, Ansgar Fehnker, Frits Vandraager, Theo Ruys, Pedro D'Argenio, J-P Katoen, Jan Tretmans, Judi Romijn, Ed Brinksma, Martijn Hendriks, Klaus Havelund, Franck Cassez, Magnus Lindahl, Francois Laroussinie, Patricia Bouyer, Augusto Burgueno, H. Bowmann, D. Latella, M. Massink, G. Faconti, Kristina Lundqvist, Lars Asplund, Justin Pearson...

Why Verification and Testing

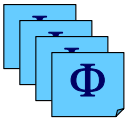
- 30-40% of production time is currently spend on elaborate, ad-hoc testing.
- High potential for improving testing methods and tools.
- Time-to-market may be shortened considerable by verification of early designs.
- Quantitative aspects essential for ES.

Verification and Testing

Model



Req



```
/* Wait for
void OS_Wait(void);

/* Operating system visualSTATE process. Mimics a OS process for a
 * visualSTATE system. In this implementation this is the mainloop
 * interfacing to the visualSTATE basic API. */
void OS_VS_Process(void);

/* Define completion code variable. */
unsigned char cc;

void HandleError(unsigned char ccArg)
{
  printf("Error code %c detected, exiting application.\n", ccArg);
  exit(ccArg);
}

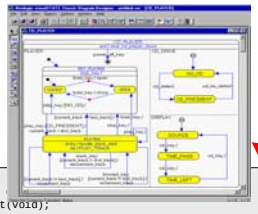

/* In d-241 we only use the OS_Wait call. It is used to simulate a
 * system. It purpose is to generate events. How this is done is up to
 * you.
 */
void OS_Wait(void)
{
  /*  Ignore the parameters; just retrieve events from the keyboard and
   *  put them into the queue. When EVENT_UNDEFINED is read from the
   *  keyboard, return to the calling process. */
  SEM_EVENT_TYPE event;
  int num;
```

Code

Running System

Verification and Testing

Model



```
/* Wait for
void OS_Wait(void);

/* Operating system visualSTATE process. Mimics a OS process for
 * visualSTATE system. In this implementation this is the mainloop
 * interfacing to the visualSTATE basic API. */
void OS_VS_Process(void);

/* Define completion code variable. */
unsigned char cc;

void HandleError(unsigned char ccArg)
{
  printf("Error code %c detected, exiting application.\n", ccArg);
  exit(ccArg);
}


/* In d-241 we only use the OS_Wait call. It is used to simulate a
 * system. It purpose is to generate events. How this is done is up to
 * you.
 */
void OS_Wait(void)
{
  /*  Ignore the parameters; just retrieve events from the keyboard and
   *  put them into the queue. When EVENT_UNDEFINED is read from the
   *  keyboard, return to the calling process. */
  SEM_EVENT_TYPE event;
  int num;
```

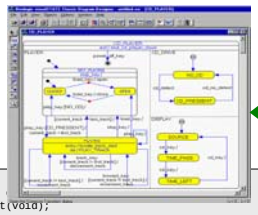
Code

Φ Req

Running System

Verification and Testing

Model



```
/* Wait for
void OS_Wait(void);

/* Operating system visualSTATE process. Mimics a OS process for a
 * visualSTATE system. In this implementation this is the mainloop
 * interfacing to the visualSTATE basic API. */
void OS_VS_Process(void);

/* Define completion code variable. */
unsigned char cc;

void HandleError(unsigned char ccArg)
{
  printf("Error code %c detected, exiting application.\n", ccArg);
  exit(ccArg);
}


/* In d-241 we only use the OS_Wait call. It is used to simul
 * system. It purpose is to generate events. How this is don
 * you.
 */
void OS_Wait(void)
{
  /*  Ignore the parameters; just retrieve events from the keyboard and
   *  put them into the queue. When EVENT_UNDEFINED is read from the
   *  keyboard, return to the calling process. */
  SEM_EVENT_TYPE event;
  int num;
```

Code

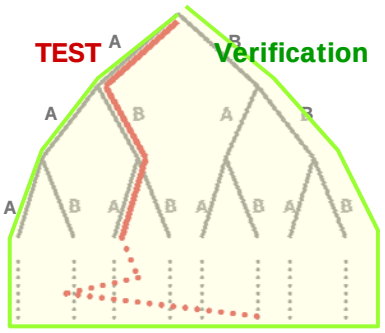
Φ Req

Running System

Test versus Verification

E F E E G H ... H A

T1 T3 T5 T1 ... T4 T3



2^n sequences of length n

Deadlock identified using **Verification**
After sequence of 2000 telegrams / < 1min

UPPAAL

Comparison

Verification

- Abstract models
- Exhaustive "proof"
- Limited size

Testing

- Checks the actual implementation
- Only few executions checked
- But is the most direct method
- Any system size*

*) Model-based test generators does not always scale

ARTIST Summer School in Morocco
Rabat, July 11-16th, 2010

Modeling, Verification and Testing of of Embedded Systems

Speaker : Brian Nielsen

Centre of Embedded Software Systems
Aalborg University, DK

Modeling, Verification, and Testing of of Embedded Systems

Brian Nielsen

Centre of
Embedded Software Systems
Aalborg University, DK

bnielsen@cs.aau.dk

Center for Indlejrede Software Systemer

Course Outline

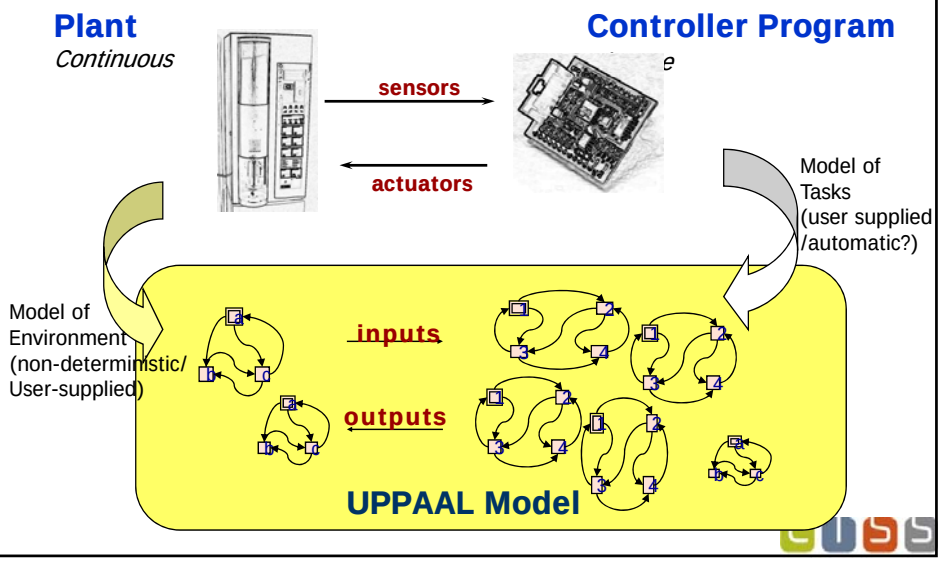
1. Introduction
2. Modeling
   1. Modelling Embedded systems
   2. Introduction to timed automata (TA)
3. Verification using Uppaal
4. Beyond Verification: Synthesis
   1. Optimal Scheduling & Planning
   2. Controller Synthesis
5. Real-Time Conformance
   1. Testing theory
   2. Real-time extensions of the ioco testing theory
6. Real-Time Test Generation
   1. Off-line generation using model checkers
   2. (optimal) quantitative test-sequences (based on Priced TA)
   3. Online real-time testing
   4. Testing strategies using Timed Games
7. Conclusions

Modeling & Verification

What can it be used for?

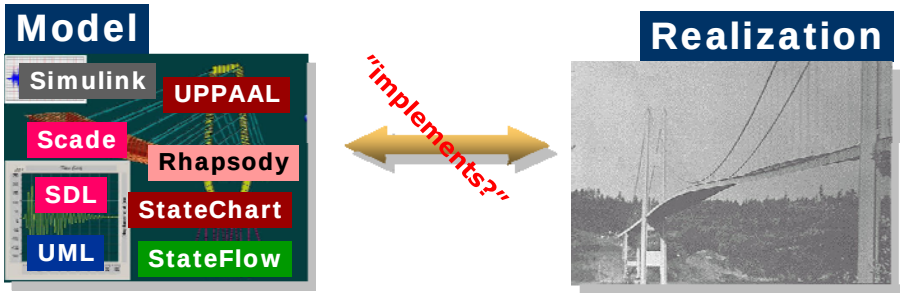
Center for Indlejrede Software Systemer

Real-time Modeling

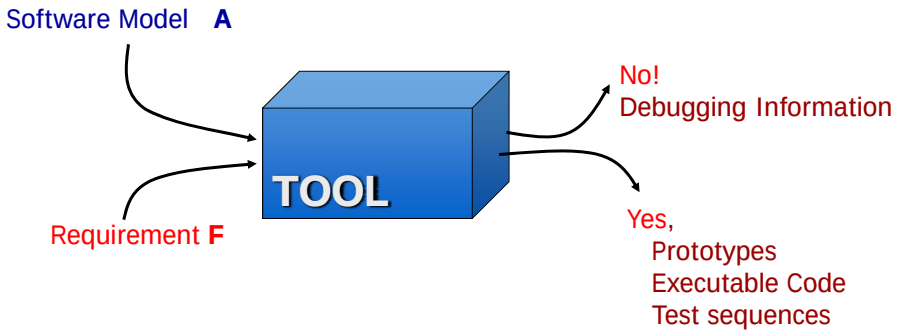


Models

- A model is a simplified representation of the real world.
- User gains confidence in the adequacy and validity of a proposed system.
- Models selected aspects. Removes irrelevant details.
- Early design exploration.

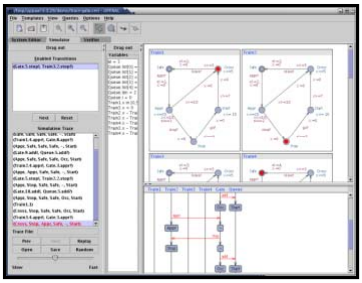


Modelling and Analysis

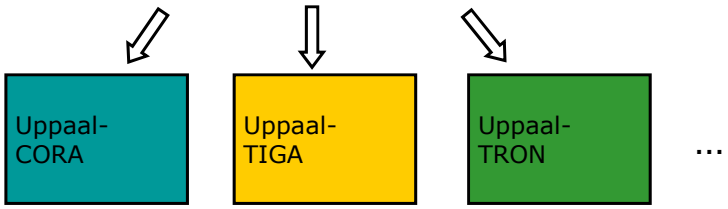


Tools: UPPAAL, visualSTATE,
ESTEREL, SPIN, Statemate, FormalCheck,
VeriSoft, Java Pathfinder,...

UPPAAL Tools www.uppaal.com



Uppaal Model-checker:
Efficient reachability analysis of network of timed automata



- TIGA: Timed games (reachability and safety)
- CORA: Cost Optimal reachability from priced TA
- TRON: Testing Real-time Online

Home-Banking?

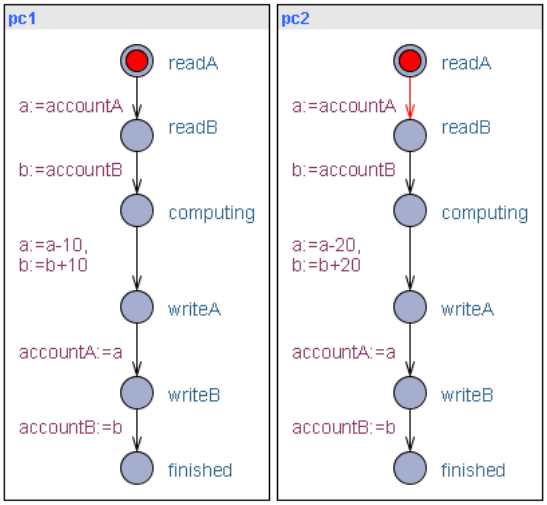
```
int accountA, accountB; //Shared global variables
//Two concurrent bank costumers

Thread costumer1 () {          Thread costumer2 () {
  int a,b; //local tmp copy      int a,b;

  a=accountA;                    a=accountA;
  b=accountB;                    b=accountB;
  a=a-10;b=b+10;                 a=a-20; b=b+20;
  accountA=a;                    accountA=a;
  accountB=b;                    accountB=b;
}                              }
```

- Initially accountA=accountB=100
- Can money be lost after the transactions?

Home Banking



A[] (pc1.finished and pc2.finished) imply (accountA+accountB==200)?

Home Banking

```
int accountA, accountB; //Shared global variables
Semaphore A,B;          //Protected by sem A,B
//Two concurrent bank costumers

Thread costumer1 () {
  int a,b; //local tmp copy

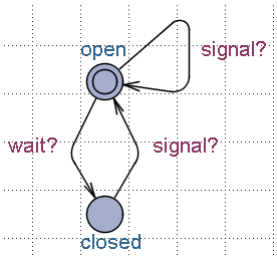
  wait(A);
  wait(B);
  a=accountA;
  b=accountB;
  a=a-10;b=b+10;
  accountA=a;
  accountB=b;
  signal(A);
  signal(B);
}
```

```
Thread costumer2 () {
  int a,b;

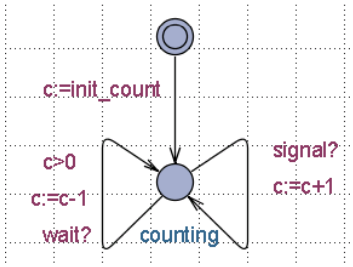
  wait(B);
  wait(A);
  a=accountA;
  b=accountB;
  a=a-20; b=b+20;
  accountA=a;
  accountB=b;
  signal(B);
  signal(A);
}
```

Semaphore Model

Binary Semaphore

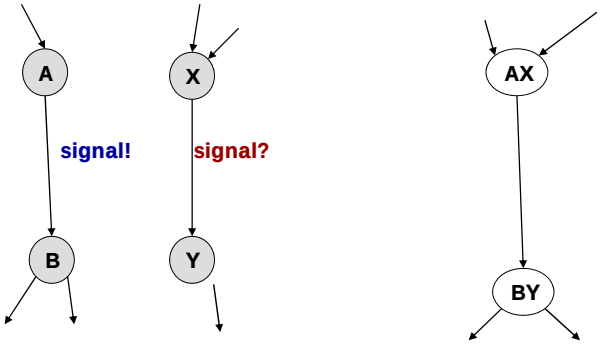


Counting Semaphore



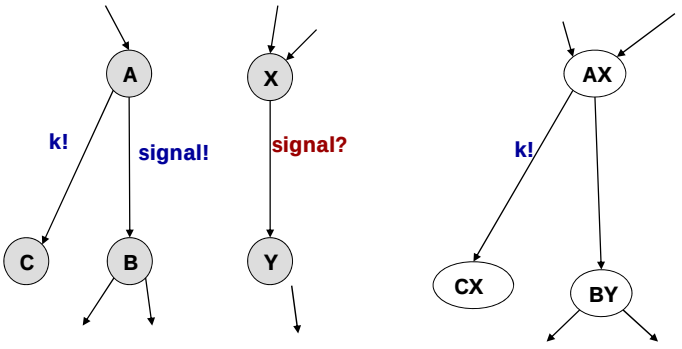
Composition

IO Automater (2-vejs synkronisering)

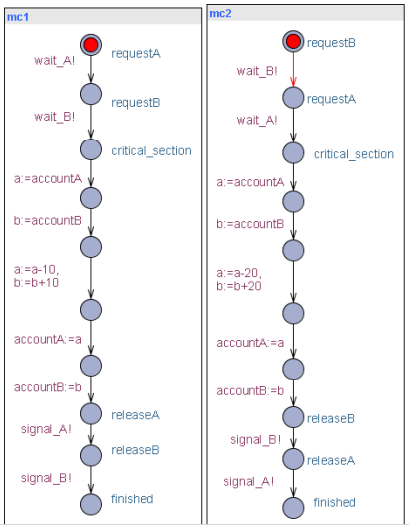


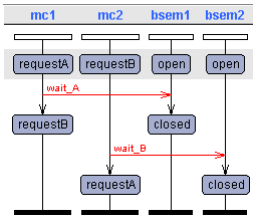
Composition

IO Automater (2-vejs synkronisering)



Semaphore Solution?



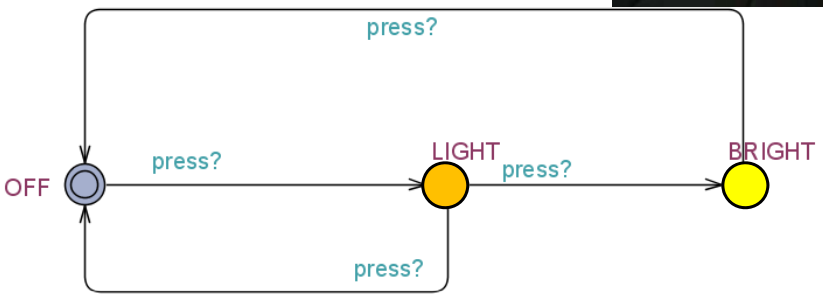


1. Consistency? (Balance)
2. Race conditions?
3. Deadlock?

1. `A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200)` ✓
2. `E<> mc1.critical_section and mc2.critical_section` ✓
3. `A[] not (mc1.finished and mc2.finished) imply not deadlock` ÷

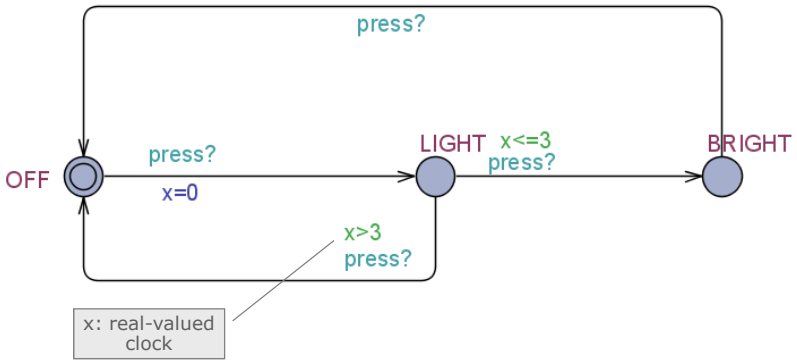
Modeling Function

Simple Light Control



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

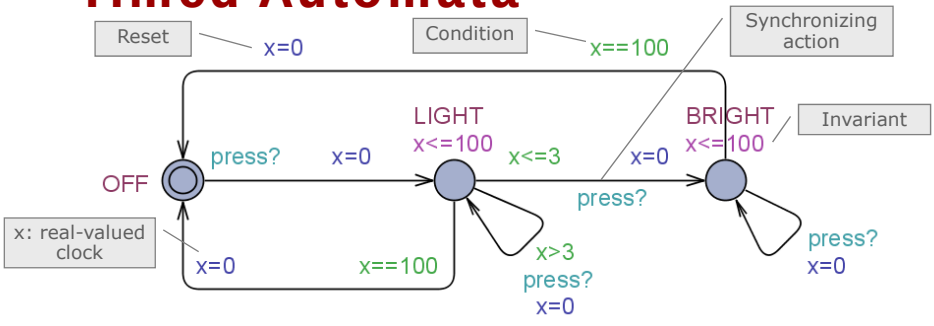
Modeling Quantities: Time



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

Modeling Quantities: Timed Automata

Alur&Dill

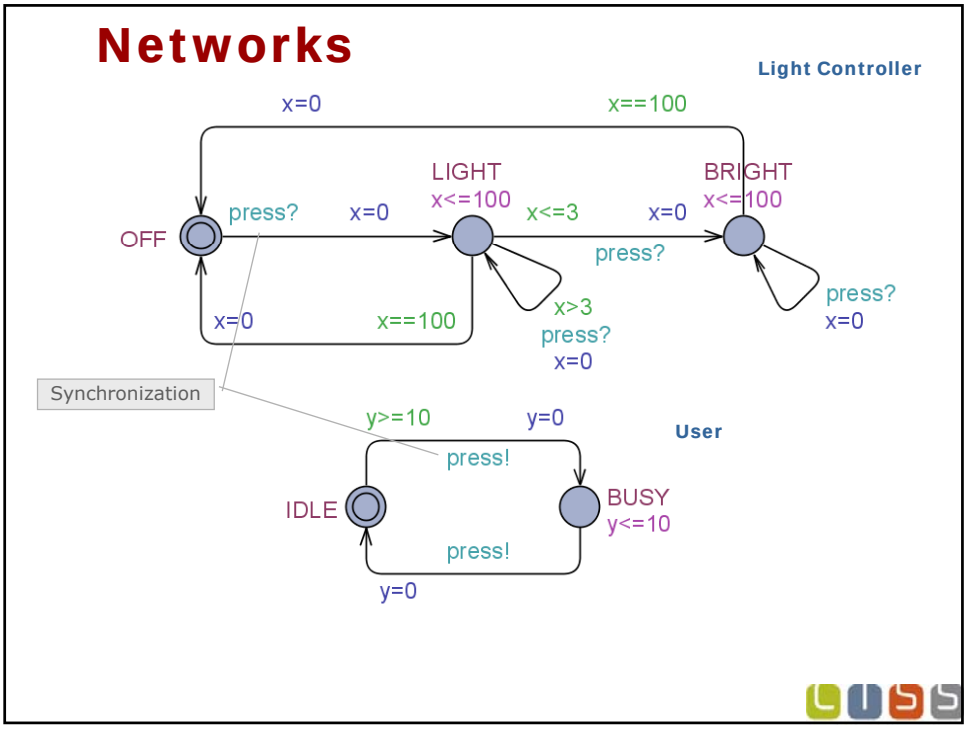


States:
(location , x=v) where v∈**R**

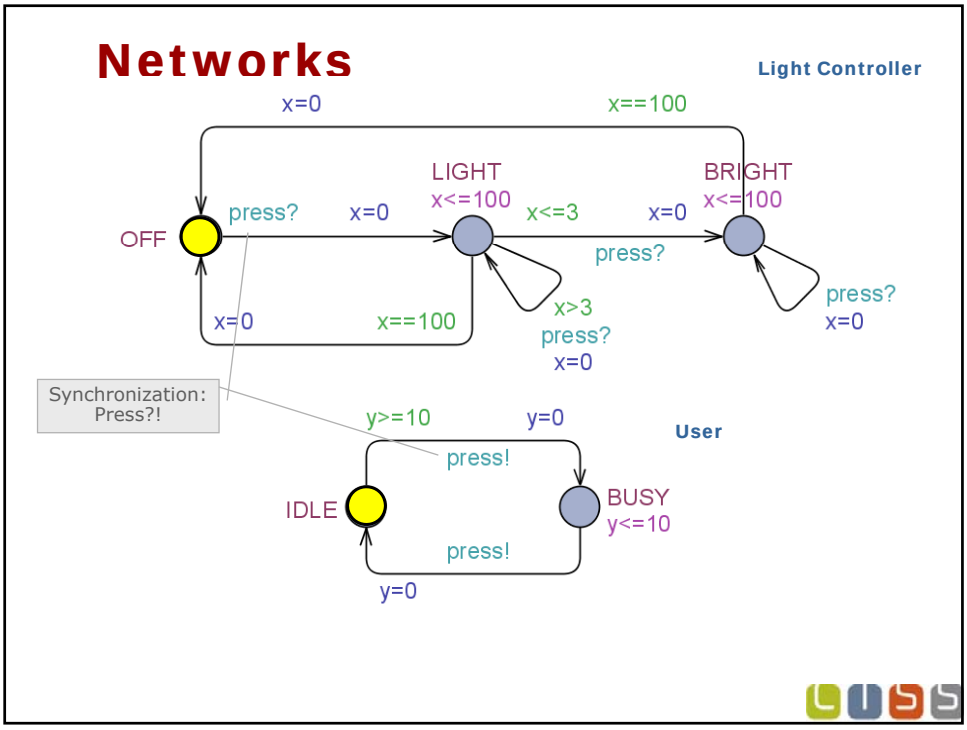
Transitions:

	(Off , x=0)
delay 4.32	→ (Off , x=4.32)
press?	→ (Light , x=0)
delay 4.51	→ (Light , x=4.51)
press?	→ (Light , x=0)
delay 100	→ (Light , x=100)
τ	→ (Off , x=0)

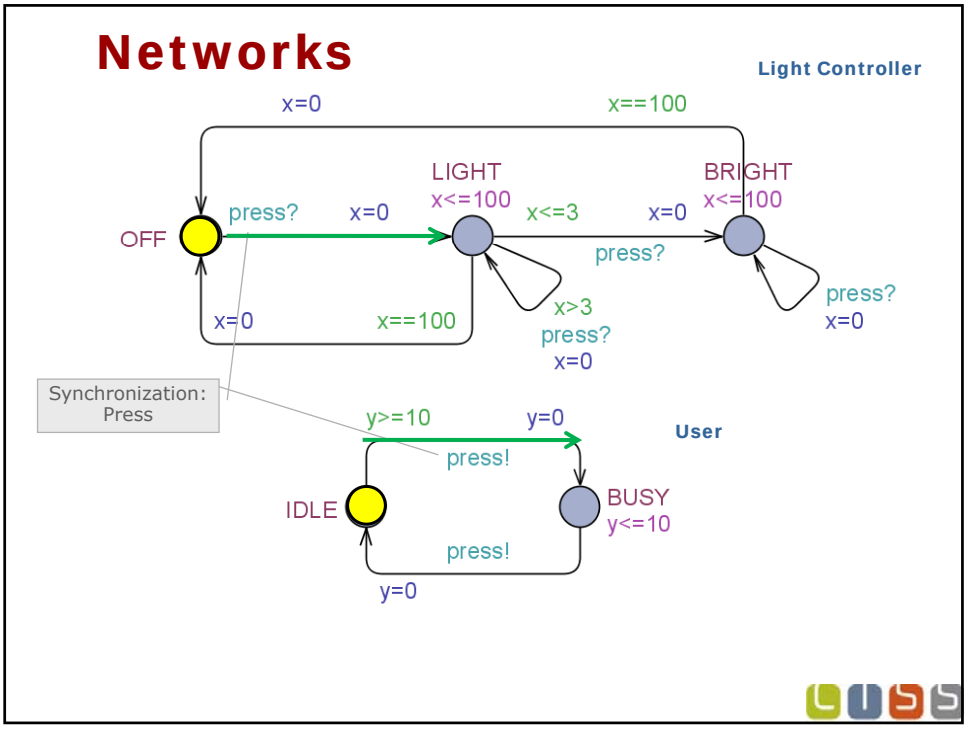
Networks



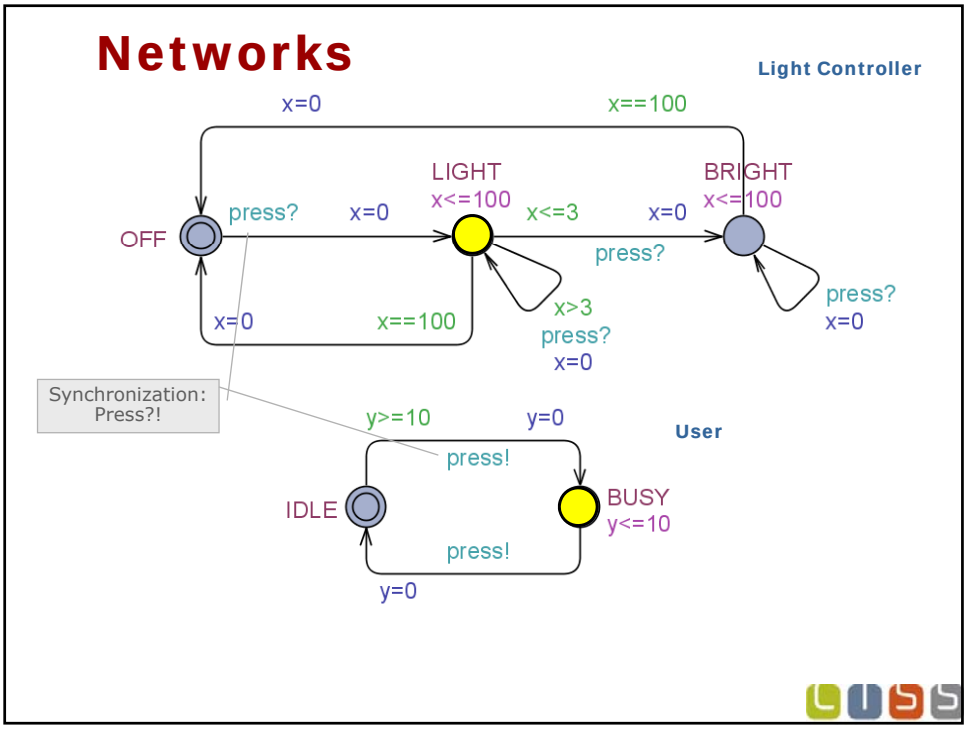
Networks



Networks



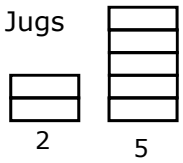
Networks



Modeling Language

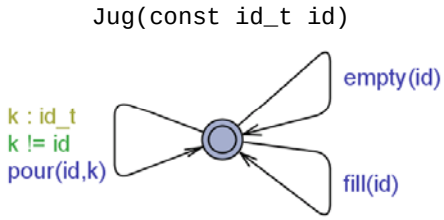
- Network of TA = instances of templates
  - argument *const type expression*
  - argument *type& name*
- Types
  - built-in types: *int*, *int[min,max]*, *bool*, arrays
  - *typedef struct { … } name*
  - *typedef built-in-type name*
- Functions
  - C-style syntax, no pointer but references OK.
- Select
  - *name : type*

Un-timed Example: Jugs



Actions:
- fill
- empty
- pour

Goal: obtain 1 unit.



- Scalable, compact, & readable model.
  - const int N = 2; typedef int[0,N-1] id_t;
  - Jugs have their own id.
  - Actions = functions.
  - Pour: from id to another k *different from id*.

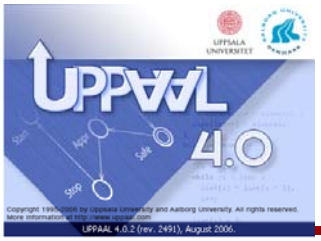
Jugs cont.

- Jug levels & capacities:
  ```
  int level[N];
  const int capa[N] = {2,5};
  ```
- `void empty(id_t i)  { level[i]=0; }`
- `void fill(id_t i)  {  level[i] = capa[i]; }`
- ```
  void pour(id_t i, id_t j)
  {
      int max = capa[j] - level[j];
      int poured = level[i] <? max; //minimum
      level[i] -= poured;
      level[j] += poured;
  }
  ```
- Auto-instantiation: `system Jug;`

Additional features

- Broadcast channels
- Committed
- Stop Watches
- Priorities


  ```



Requirements
Timed Logic

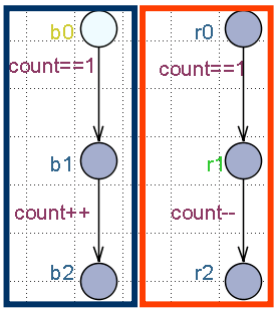
Center for Indlejrede Software Systemer

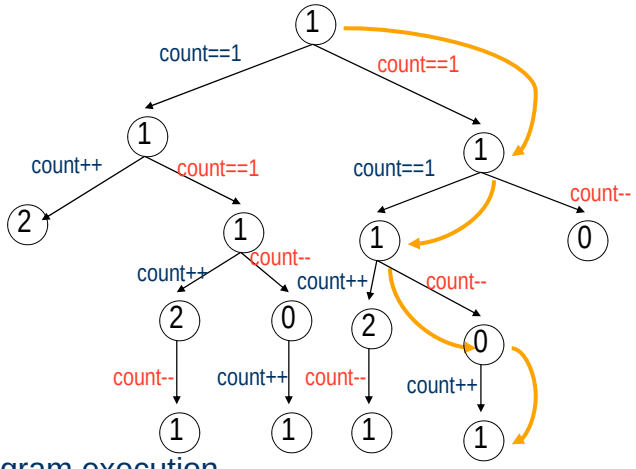
Course Outline

1. Introduction
2. Modeling
   1. Modelling Embedded systems
   2. Introduction to timed automata (TA)
3. Verification using Uppaal
4. Beyond Verification: Synthesis
   1. Optimal Scheduling & Planning
   2. Controller Synthesis
5. Real-Time Conformance
   1. Testing theory
   2. Real-time extensions of the ioco testing theory
6. Real-Time Test Generation
   1. Off-line generation using model checkers
   2. (optimal) quantitative test-sequences (based on Priced TA)
   3. Online real-time testing
   4. Testing strategies using Timed Games
7. Conclusions

State Search

```
Int count:=1
```





Each trace = a program execution
Uppaal checks *all* traces

Is count *possibly* 3 ? `E<> count==3`
Is count *always* 1 ? `A[] count==1`

Logical Specifications

Subset of timed CTL

- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leads-to: *P* → *Q*
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\leq t} Q$

The expressions *P* and *Q* are state-predicates

Logical Specifications

The expressions *P* and *Q* must be type safe, side effect free, and evaluate to a boolean.

Only references to integer variables, constants, clocks, and locations are allowed

process location data guards clock guards

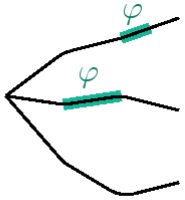
```
p::= a.l | gd | gc | p and p |
     p or p | not p | p imply p |
     ( p ) | deadlock(only for A[],E<>)
```

```
A[] (mc1.finished and mc2.finished) imply (accountA+accountB==200)
```

Logical Specifications

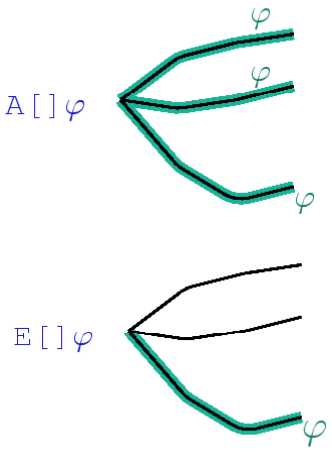
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: *P* → *Q*
- Bounded Liveness
  - Leads to within: $P \rightarrow_{t} Q$

E<>φ



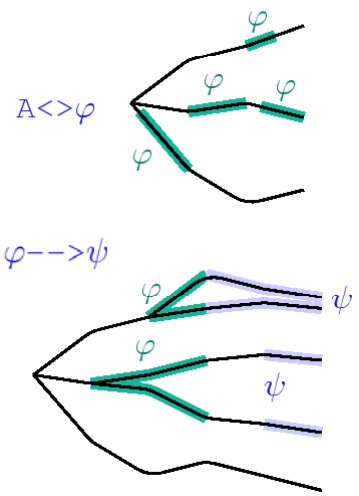
Logical Specifications

- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: *P* → *Q*
- Bounded Liveness
  - Leads to within: *P* →$_{t}$ *Q*



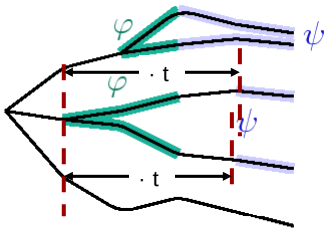
Logical Specifications

- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: *P* → *Q*
- Bounded Liveness
  - Leads to within: *P* →$_{t}$ *Q*



Logical Specifications

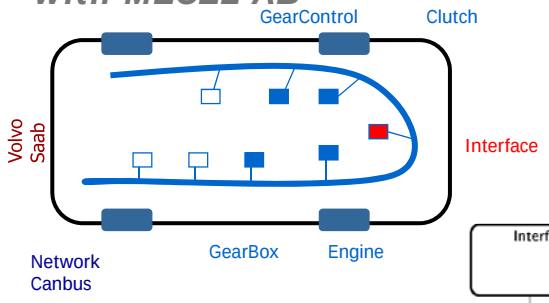
- Validation Properties
  - Possibly: E<> *P*
- Safety Properties
  - Invariant: A[] *P*
  - Pos. Inv.: E[] *P*
- Liveness Properties
  - Eventually: A<> *P*
  - Leadsto: *P* → *Q*
- Bounded Liveness
  - Leads to within: $P \rightarrow_{\cdot t} Q$



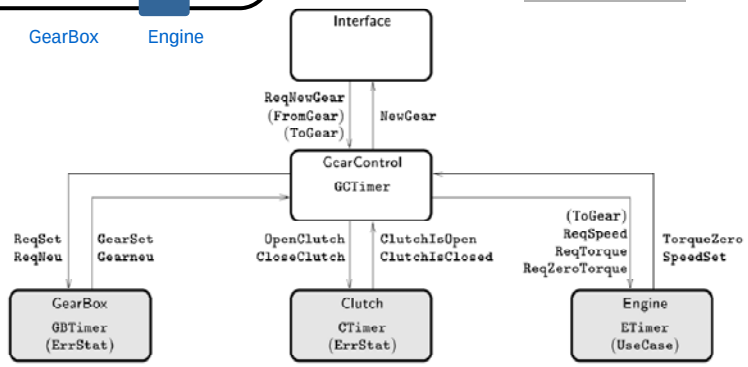
Gear Controller

with MECEL AB

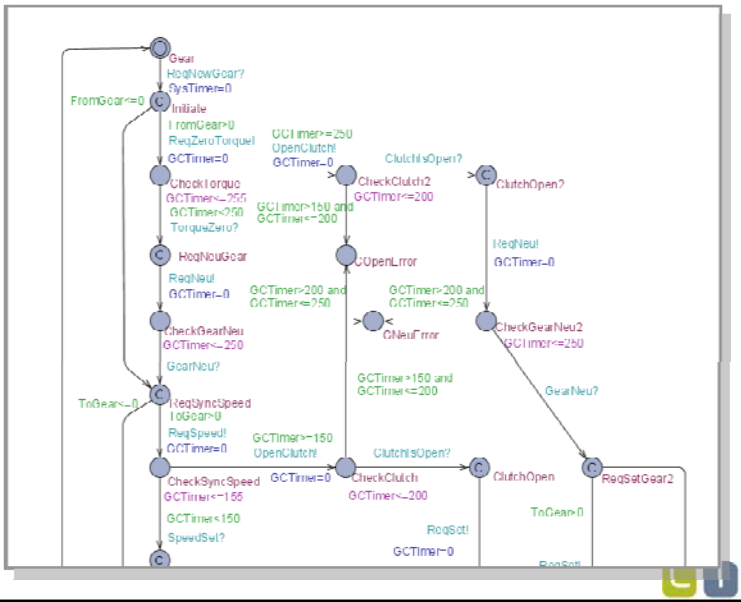
Lindahl, Pettersson, Yi



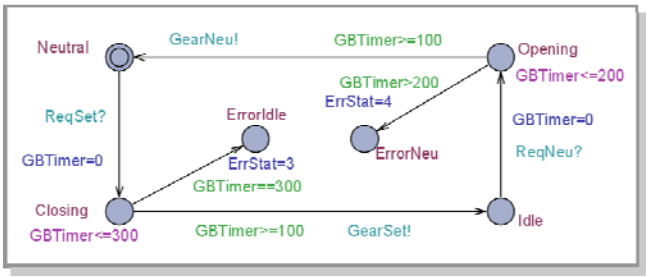
Flowgraph

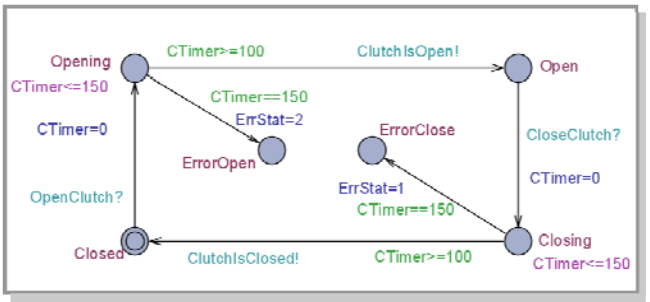


Gear Control (partial)



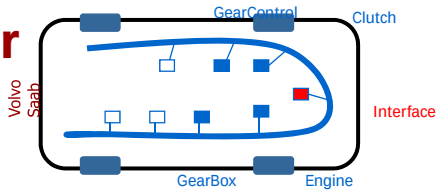
GearBox & Clutch





Gear Controller

with MECEL AB



Requirements

GearControl@Initiate $\rightsquigarrow_{\leq 1500}$ ((ErrStat $= 0$) $\Rightarrow$ GearControl@GearChanged) (1)

GearControl@Initiate $\rightsquigarrow_{\leq 1000}$
((ErrStat $= 0 \wedge$ UseCase $= 0$) $\Rightarrow$ GearControl@GearChanged) (2)

Clutch@ErrorClose $\rightsquigarrow_{\leq 200}$ GearControl@CCloseError (3)

Clutch@ErrorOpen $\rightsquigarrow_{\leq 200}$ GearControl@COpenError (4)

GearBox@ErrorIdle $\rightsquigarrow_{\leq 350}$ GearControl@GSetError (5)

GearBox@ErrorNeu $\rightsquigarrow_{\leq 200}$ GearControl@GNeuError (6)

Inv (GearControl@CCloseError $\Rightarrow$ Clutch@ErrorClose) (7)

Inv (GearControl@COpenError $\Rightarrow$ Clutch@ErrorOpen) (8)

Inv (GearControl@GSetError $\Rightarrow$ GearBox@ErrorIdle) (9)

Inv (GearControl@GNeuError $\Rightarrow$ GearBox@ErrorNeu) (10)

Inv (Engine@ErrorSpeed $\Rightarrow$ ErrStat $\neq 0$) (11)

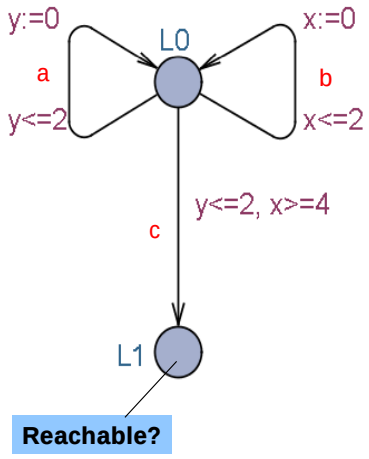
Inv (Engine@Torque $\Rightarrow$ Clutch@Closed) (12)

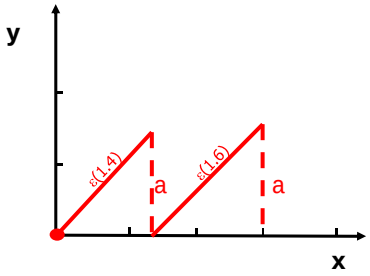
Uppaal Internals

How does it work?

Center for Indlejrede Software Systemer

Example



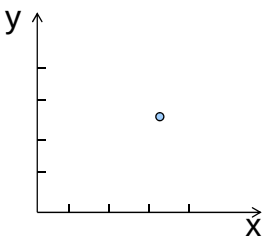


(L0,x=0,y=0)
→ε(1.4)
(L0,x=1.4,y=1.4)
→a
(L0,x=1.4,y=0)
→ε(1.6)
(L0,x=3.0,y=1.6)
→a
(L0,x=3.0,y=0)

Zones

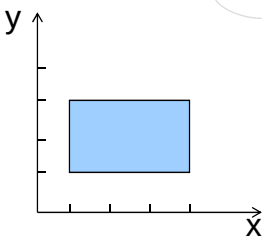
From infinite to finite

State
(n, x=3.2, y=2.5)

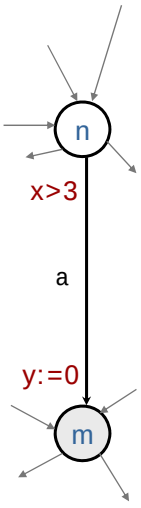


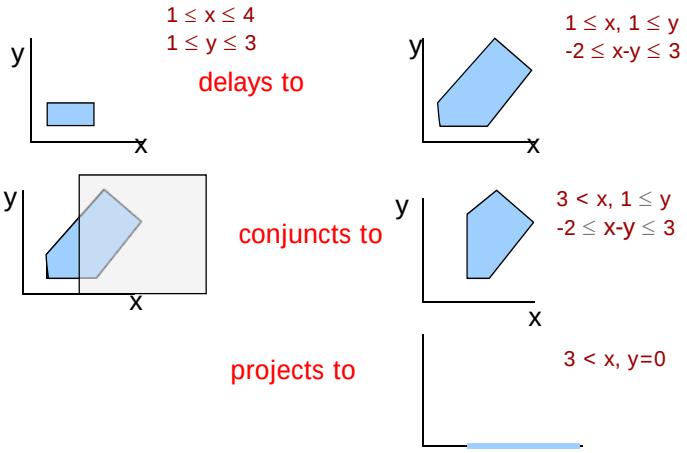
Symbolic state (set)
(n, $1 \le x \le 4$, $1 \le y \le 3$)

Zone:
conjunction of
x-y<=n,
x<=n,
x>=n



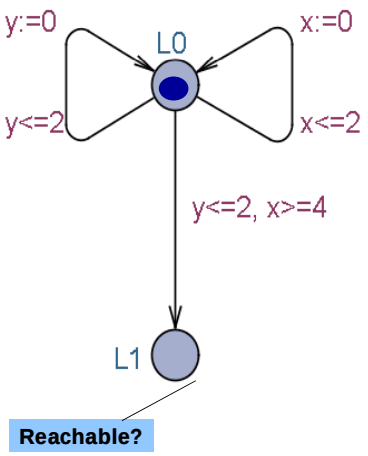
Symbolic Transitions

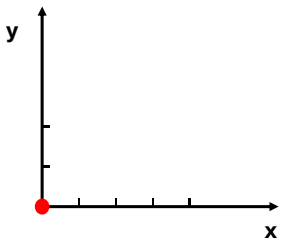




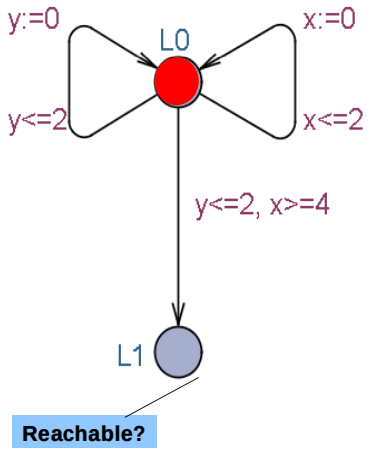
Thus (n, $1 \le x \le 4, 1 \le y \le 3$) $\rightarrow^{a}$ (m, $3 < x, y=0$)

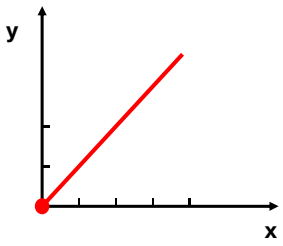
Symbolic Exploration





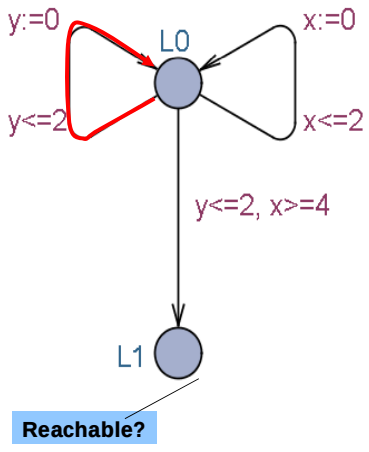
Symbolic Exploration

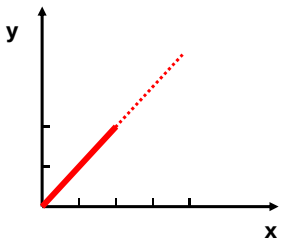




Delay

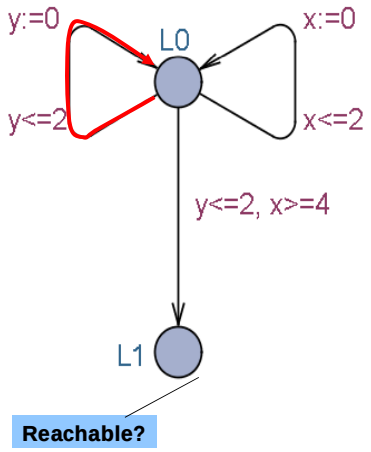
Symbolic Exploration

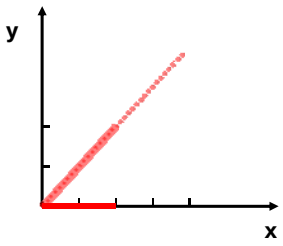




Left

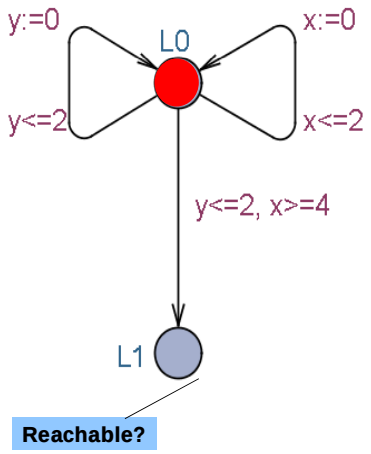
Symbolic Exploration

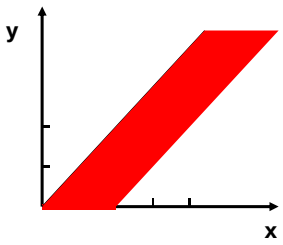




Left

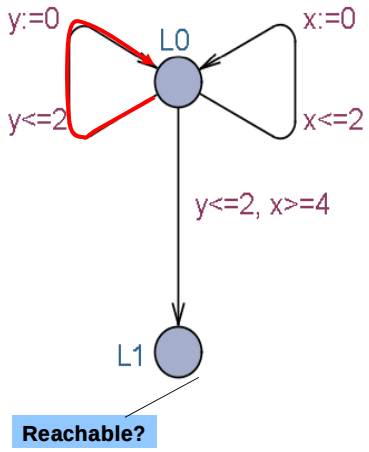
Symbolic Exploration

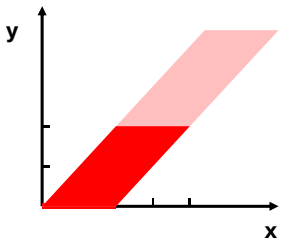




Delay

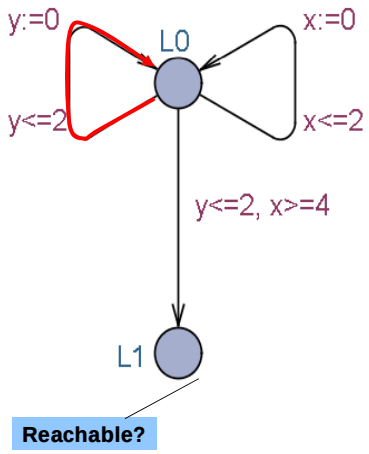
Symbolic Exploration

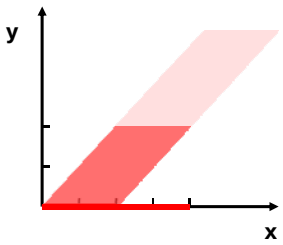




Left

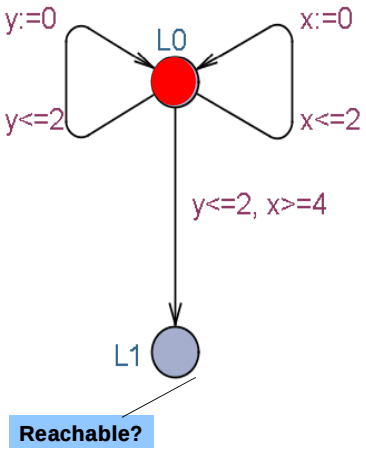
Symbolic Exploration

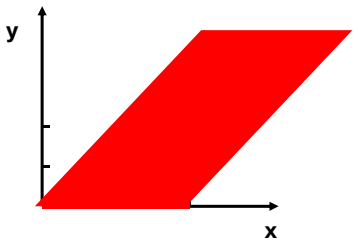




Left

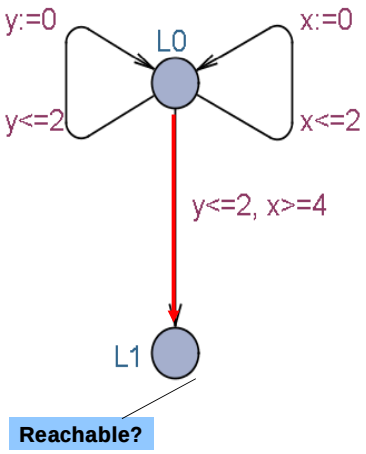
Symbolic Exploration

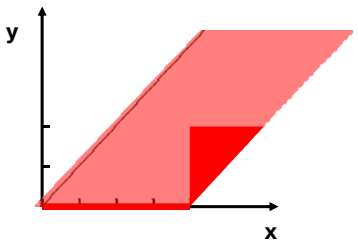




Delay

Symbolic Exploration



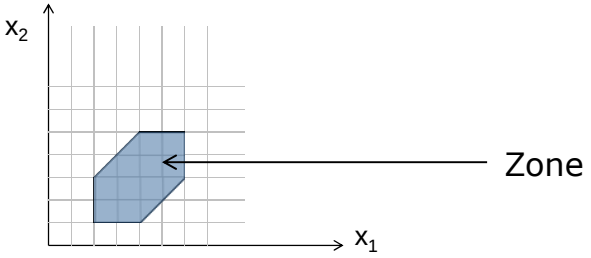


Down

Difference Bound Matrices

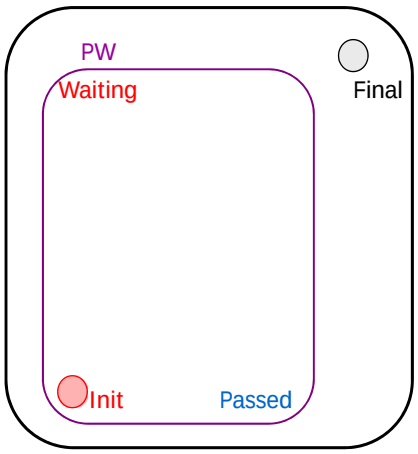
$x_0-x_0<=0$	$x_0-x_1<=-2$	$x_0-x_2<=-1$
$x_1-x_0<=6$	$x_1-x_1<=0$	$x_1-x_2<=3$
$x_2-x_0<=5$	$x_2-x_1<=1$	$x_2-x_2<=0$

$x_i-x_j<=c_{ij}$



Forward Reachability Algorithm

Init -> Final ?



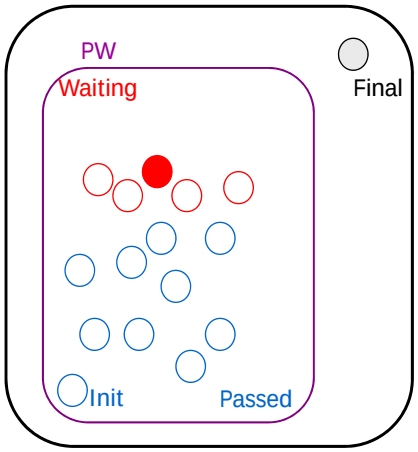
INITIAL Passed := Ø;
Waiting := {(n_0,Z_0)}

REPEAT

UNTIL Waiting = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



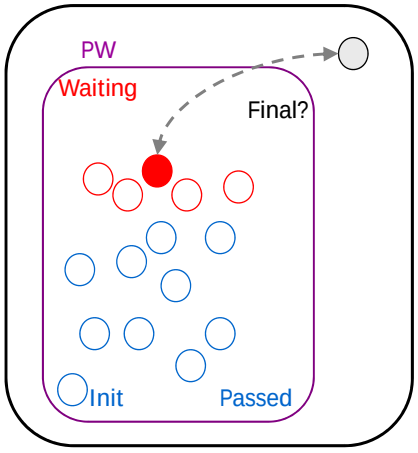
INITIAL Passed := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick (n,Z) in Waiting

UNTIL Waiting = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



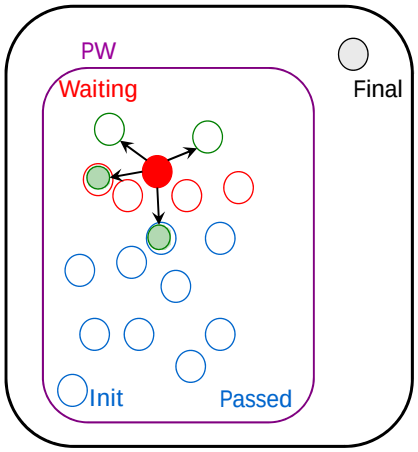
INITIAL Passed := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick (n,Z) in Waiting
if (n,Z) = Final return true

UNTIL Waiting = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



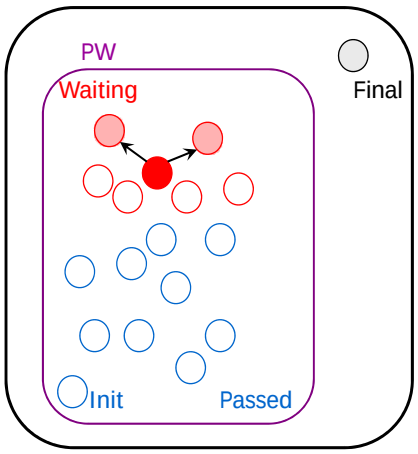
INITIAL Passed := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick (n,Z) in Waiting
if (n,Z) = Final return true
for all (n,Z)→(n',Z'):
if for some (n',Z'') $Z' \subseteq Z''$ continue

UNTIL Waiting = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



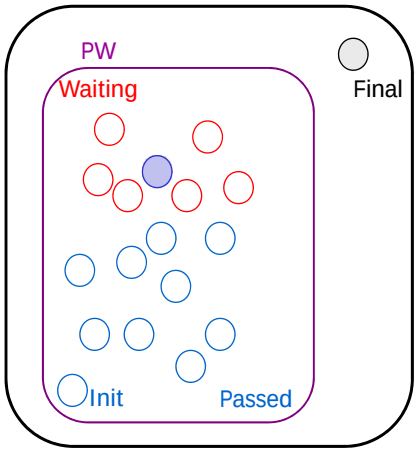
INITIAL Passed := Ø;
Waiting := $\{(n_0,Z_0)\}$

REPEAT
pick (n,Z) in Waiting
if (n,Z) = Final return true
for all (n,Z)→(n',Z'):
if for some (n',Z'') $Z' \subseteq Z''$ continue
else add (n',Z') to Waiting

UNTIL Waiting = Ø
return false

Forward Reachability Algorithm

Init -> Final ?



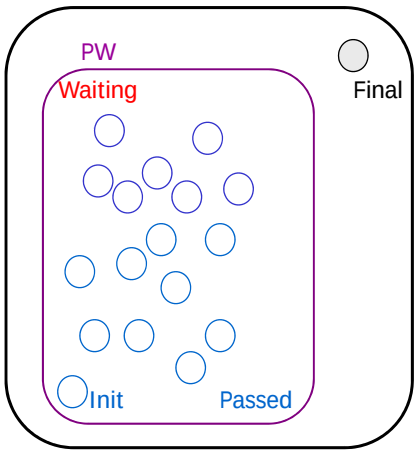
INITIAL Passed := Ø;
Waiting := {(n_0,Z_0)}

REPEAT
pick (n,Z) in Waiting
if (n,Z) = Final return true
for all (n,Z)→(n',Z'):
if for some (n',Z'') Z'⊆ Z'' continue
else add (n',Z') to Waiting
move (n,Z) to Passed

UNTIL Waiting = Ø
return false

Forward Reachability Algorithm

Init -> Final ?

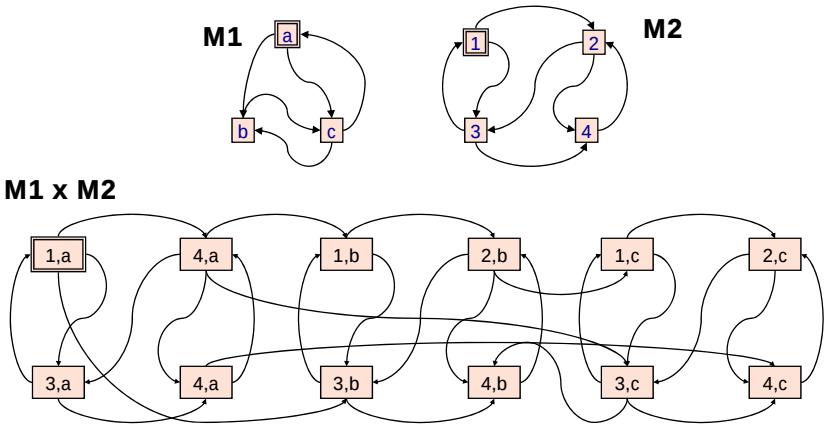


INITIAL Passed := Ø;
Waiting := {(n_0,Z_0)}

REPEAT
pick (n,Z) in Waiting
if (n,Z) = Final return true
for all (n,Z)→(n',Z'):
if for some (n',Z'') Z'⊆ Z'' continue
else add (n',Z') to Waiting
move (n,Z) to Passed

UNTIL Waiting = Ø
return false

State Space Explosion Problem



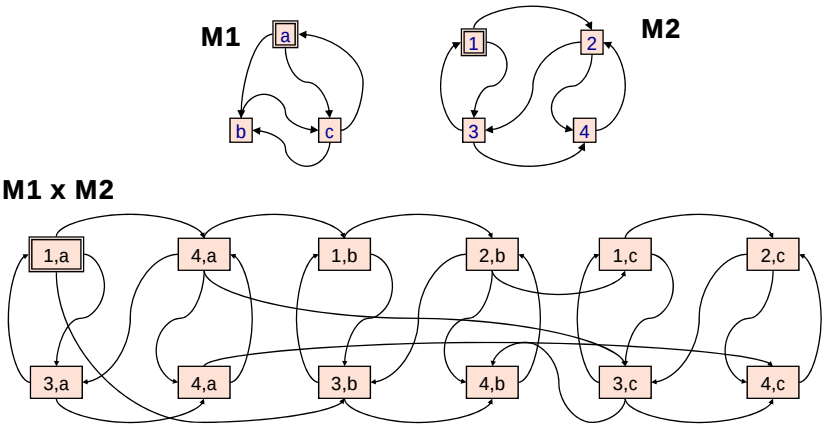
All combinations=
exponential in no of machines

Optimizations

- Compact data structures
  - Shortest path reduction
  - Clock Difference Diagrams
- ”To store or not to store”
- Active clock reduction
- Clock bound optimization

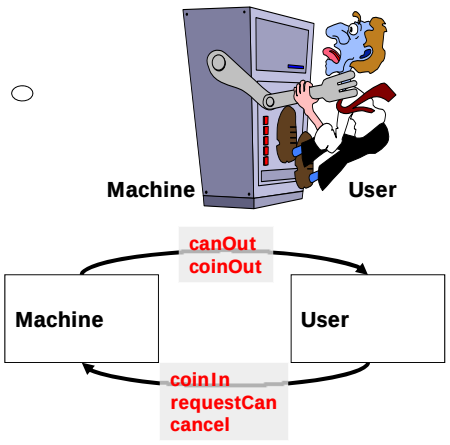
- Over approximations (Convex Hull)
- Under approximations

BUT always
State Space Explosion Problem



All combinations=
exponential in no of machines

Modelling Exercise
The Vending Machine



Simulate model w Random User

Model Fair User

Model Non-Thirsty User

Deadlocks ?

Cans requested will be delivered ?

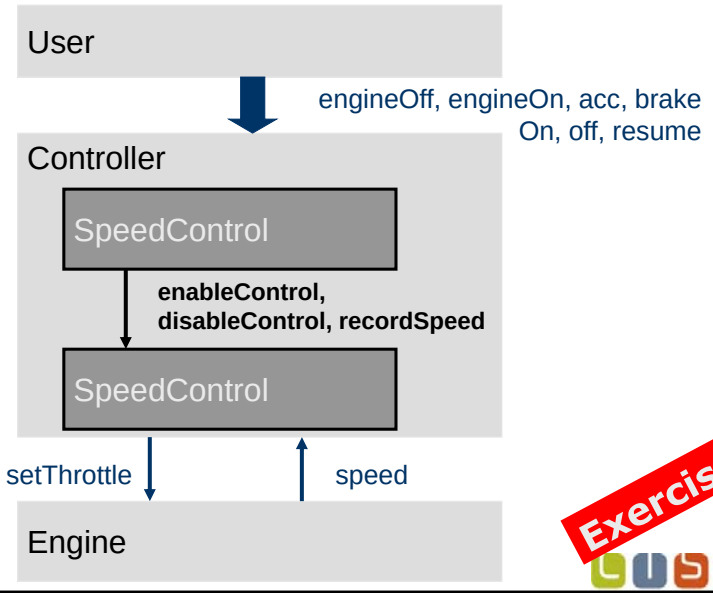
Cancellations are obeyed ?

What happens if multiple users?

Assumption: 1 can = 1 coin!

Exercise

The Cruise Controller



ARTIST Summer School in Morocco
Rabat, July 11-16th, 2010

Modeling, Verification and Testing of of Embedded Systems

Speaker : Brian Nielsen

Centre of Embedded Software Systems
Aalborg University, DK

Modeling, Verification, (Synthesis), and Testing of of Embedded Systems

Brian Nielsen

Centre of
Embedded Software Systems
Aalborg University, DK

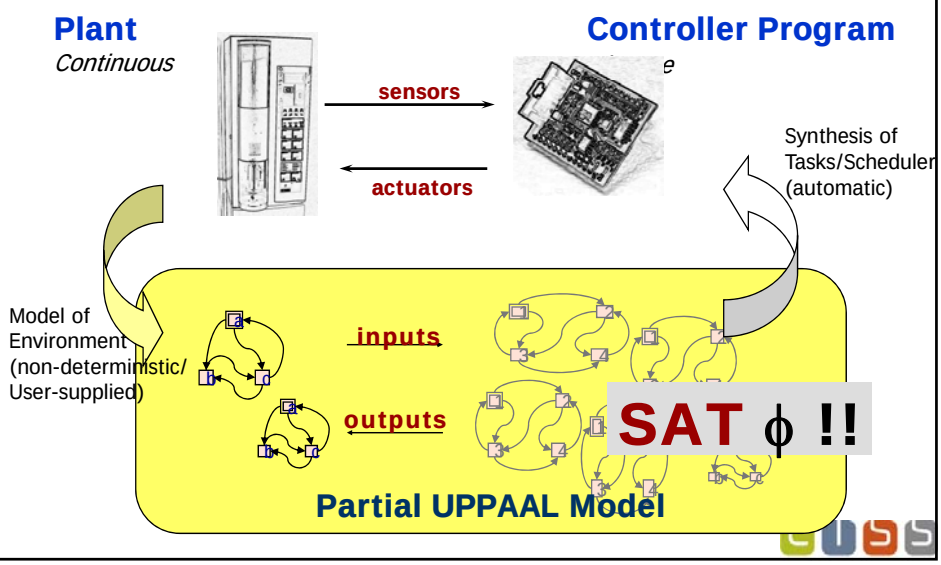
bnielsen@cs.aau.dk

Center for Indlejrede Software Systemer

Course Outline

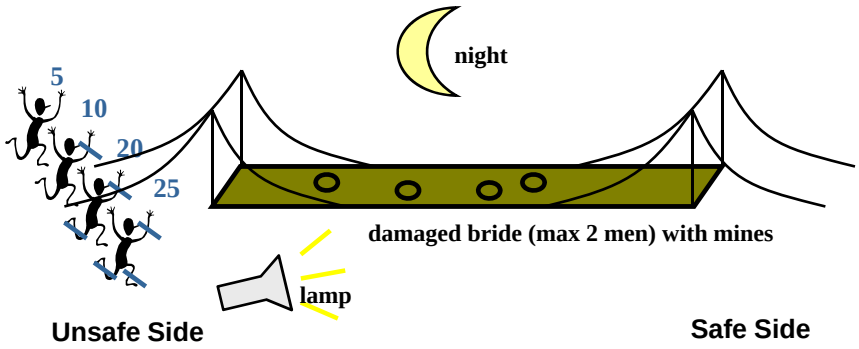
1. Introduction
2. Modeling
   1. Modelling Embedded systems
   2. Introduction to timed automata (TA)
3. Verification using Uppaal
4. Beyond Verification: Synthesis
   1. Optimal Scheduling & Planning
   2. Controller Synthesis
5. Real-Time Conformance
   1. Testing theory
   2. Real-time extensions of the ioco testing theory
6. Real-Time Test Generation
   1. Off-line generation using model checkers
   2. (optimal) quantitative test-sequences (based on Priced TA)
   3. Online real-time testing
   4. Testing strategies using Timed Games
7. Conclusions

Real-time Synthesis



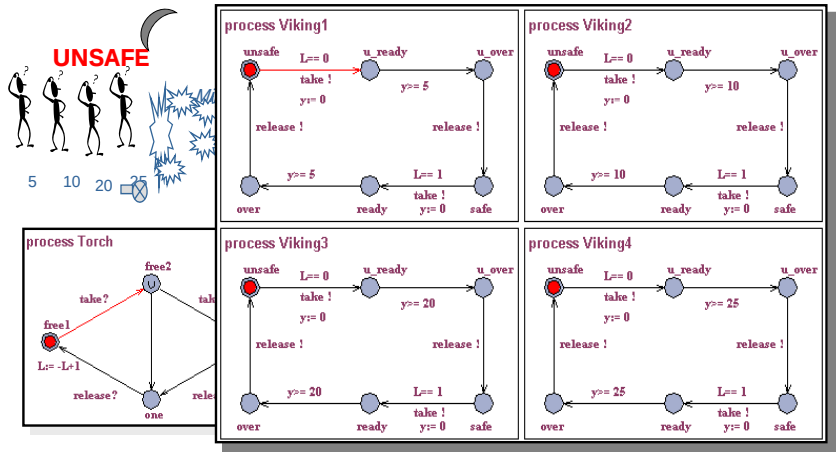
Scheduling and optimization

Example: Bridge Problem



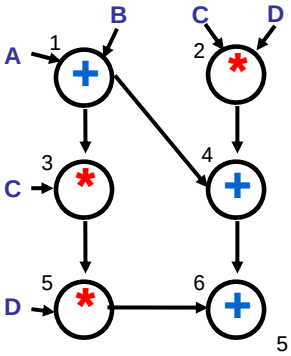
If possible find schedule for all four men to reach safe side in 60 min.

Bridge Problem



- Can be modeled
  and solved with timed automata in UPPAAL.

Optimal Scheduling – Time



Compute :
(D * (C * (A + B)) + ((A + B) + (C * D))

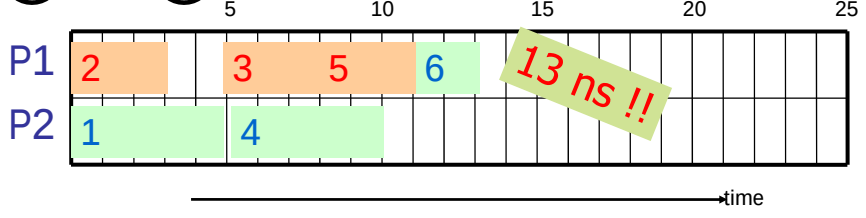
using 2 processors

P1 (fast)

+	2ns
*	3ns

P2 (slow)

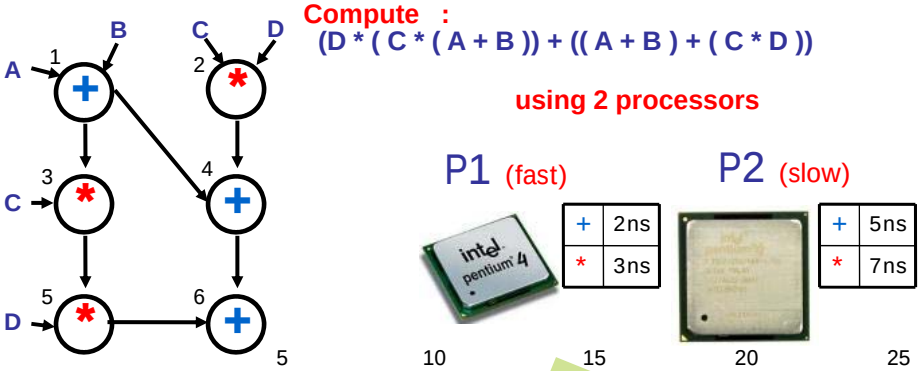
+	5ns
*	7ns



Optimal Scheduling – Time

Compute :
(D * (C * (A + B)) + ((A + B) + (C * D))

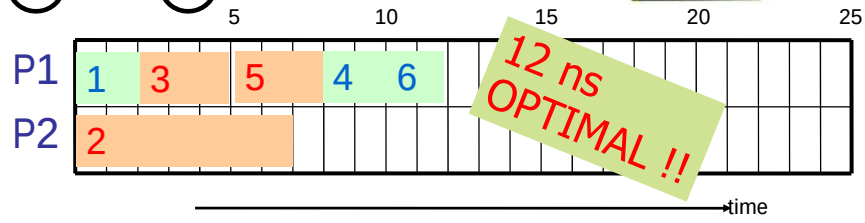
using 2 processors



P1 (fast) P2 (slow)

+	2ns
*	3ns

+	5ns
*	7ns

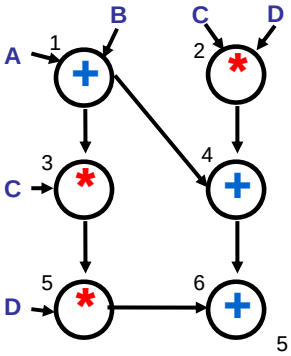


12 ns OPTIMAL !!

Optimal Scheduling – Power

Compute :
(D * (C * (A + B)) + ((A + B) + (C * D))

using 2 processors



P1 (fast) P2 (slow)

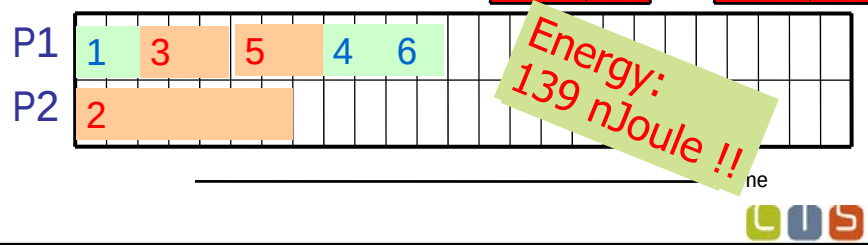
+	2ns
*	3ns

+	5ns
*	7ns

ENERGY:

Idle	1W
In use	9W

Idle	2W
In use	3W

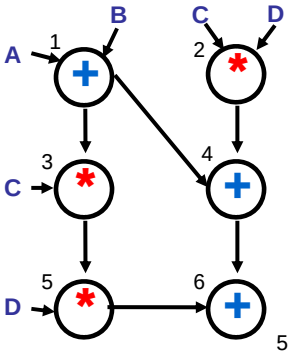


Energy: 139 nJoule !!

Optimal Scheduling – Power

Compute :
(D * (C * (A + B)) + ((A + B) + (C * D))

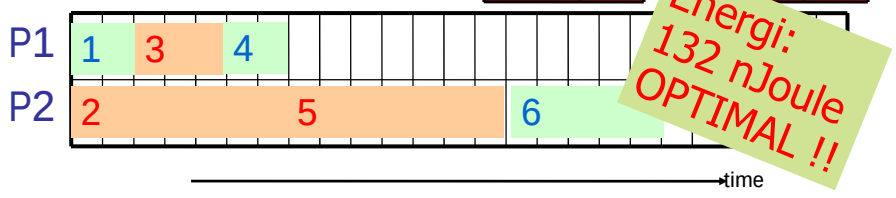
using 2 processors



P1 (fast) P2 (slow)

	P1 (fast)	P2 (slow)
+	2ns	5ns
*	3ns	7ns
Idle	1W	2W
In use	9W	3W

ENERGY:
10

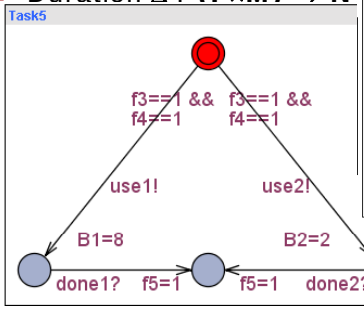


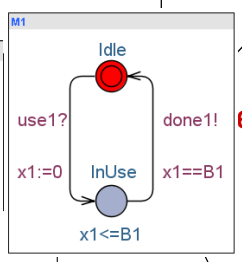
Energi:
132 nJoule
OPTIMAL !!

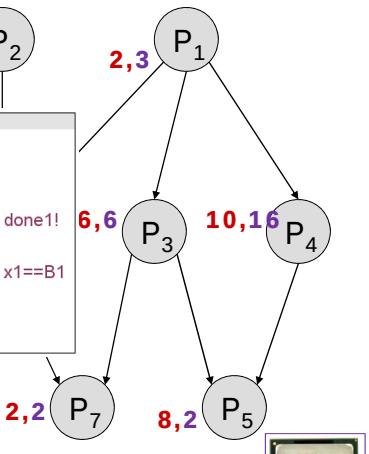
Task Graph Scheduling

Optimal Static Task Scheduling

- Task **P** = {P_1,.., P_m}
- Machines **M** = {M_1,..,M_n}
- Duration Δ : **(P×M) → N**







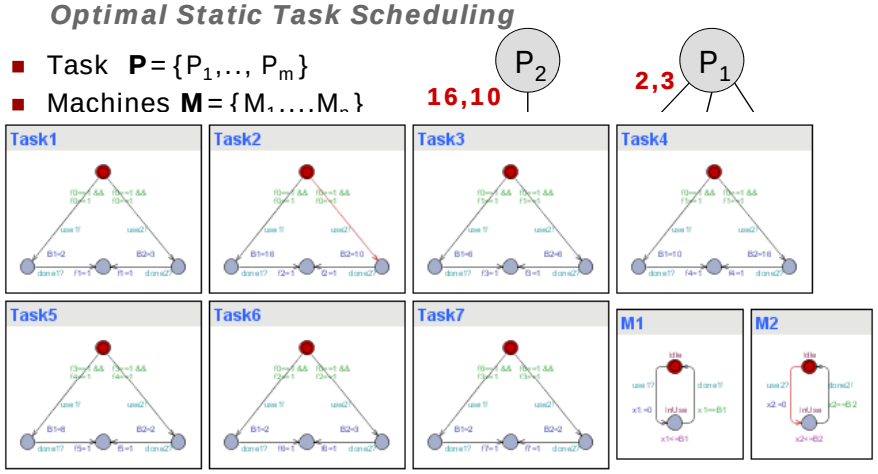
M = {M_1,M_2}

- Compute schedule with minimum completion-time!

Task Graph Scheduling

Optimal Static Task Scheduling

- Task **P** = {P_1,.., P_m}
- Machines **M** = {M_1,...,M_n}



done

E<> (Task1.End and … and Task7.End)

Experimental Results

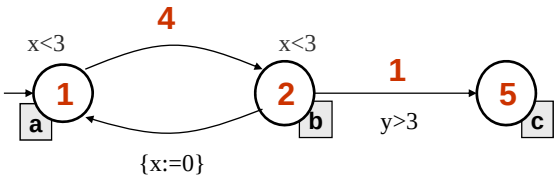
name	#tasks	#chains	# machines	optimal	TA
001	437	125	4	1178	1182
000	452	43	20	537	537
018	730	175	10	700	704
074	1007	66	12	891	894
021	1145	88	20	605	612
228	1187	293	8	1570	1574
071	1193	124	20	629	634
271	1348	127	12	1163	1164
237	1566	152	12	1340	1342
231	1664	101	16	t.o.	1137
235	1782	218	16	t.o.	1150
233	1980	207	19	1118	1121
294	2014	141	17	1257	1261
295	2168	965	18	1318	1322
292	2333	318	3	8009	8009
298	2399	303	10	2471	2473

AMETIST
advanced methods for timed systems

Symbolic A*
Brand-&-Bound
60 sec

Abdeddaïm, Kerbaa, Maler

Linearly Priced Timed Automata



- Timed Automata + costs on transitions and locations
- Cost of performing transition: transition cost
- Cost of performing delay ε : (ε x location cost)
- Trace:

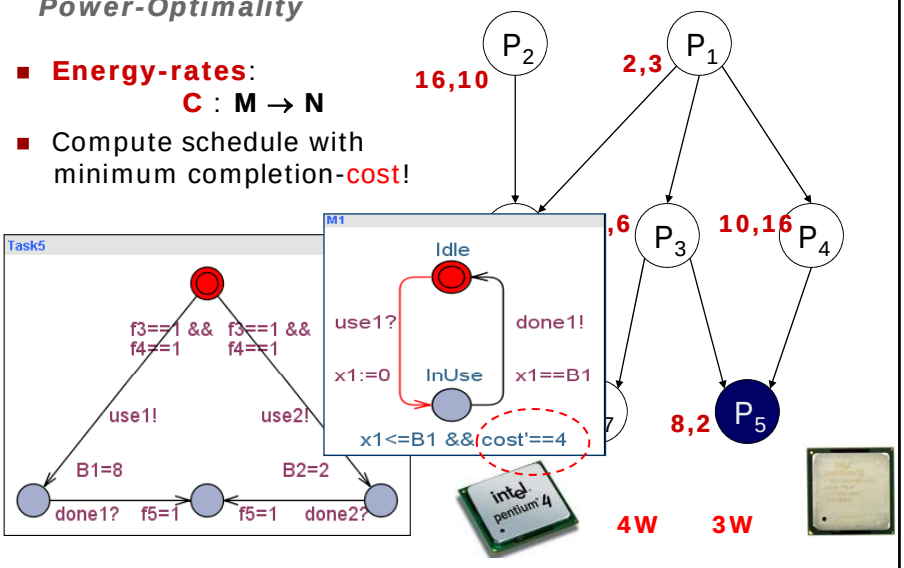
(**a**,x=y=0) —4→ (**b**,x=y=0) —ε(2.5), 2.5 x 2→ (**b**,x=y=2) —0→ (**a**,x=0,y=2)

- Cost of Execution Trace:
  - Sum of costs: **4 + 5 + 0 = 9**

Optimal Task Graph Scheduling

Power-Optimality

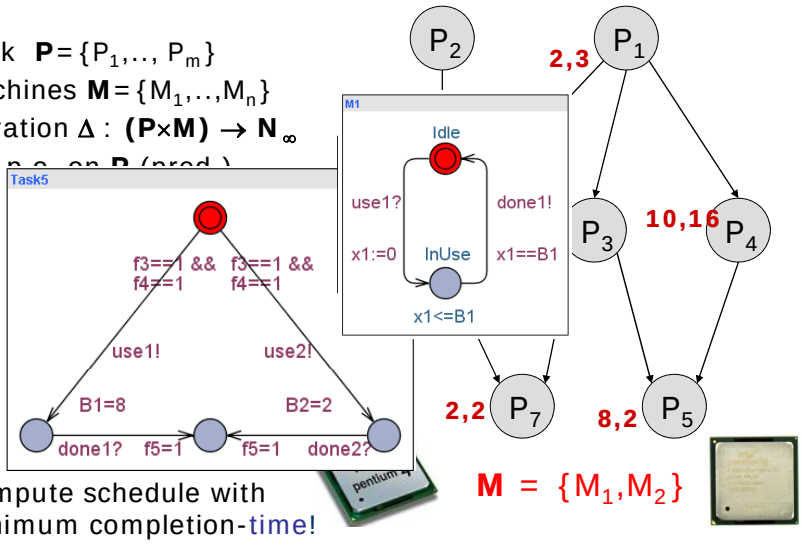
- **Energy-rates**:
  C : **M** → **N**
- Compute schedule with minimum completion-cost!



Task Graph Scheduling

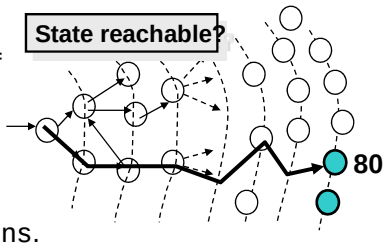
Optimal Static Task Scheduling

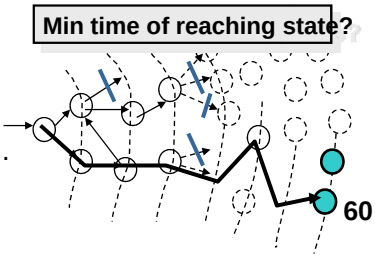
- Task $\mathbf{P} = \{P_1, .., P_m\}$
- Machines $\mathbf{M} = \{M_1, .., M_n\}$
- Duration Δ : $(\mathbf{P} \times \mathbf{M}) \rightarrow \mathbf{N}_\infty$
- < : p.e. on **P** (prod.)
- A ta
  if al
  con
- Eac
  at r
- Tas
- Compute schedule with minimum completion-time!



Verification vs. Optimization

- Verification Algorithms:
  - Checks a logical property of the entire state-space of a model.
  - Efficient Blind search.
- Optimization Algorithms:
  - Finds (near) optimal solutions.
  - Uses techniques to avoid non-optimal parts of the state-space (e.g. Branch and Bound).
- Objective:
  - Bridge gap between the two.
  - New techniques and applications in UPPAAL.

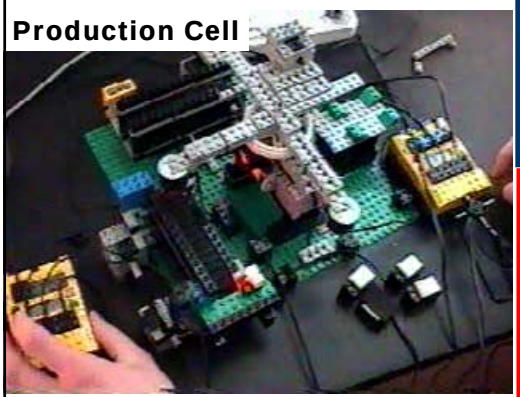




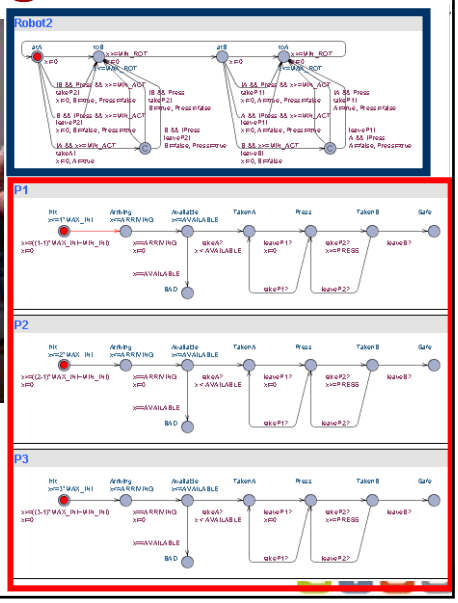
Controller Synthesis

Controller Synthesis and Timed Games

Production Cell

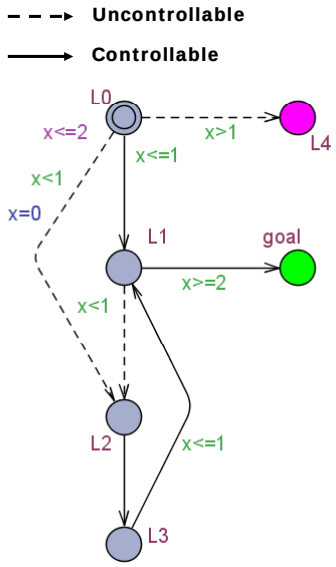


GIVEN System moves **S**,
Controller moves **C**, and property ϕ
FIND strategy s_C such that $s_C || S$ sat ϕ
→
A Two-Player Game



Timed Game Automata

[Maler, Pnueli, Sifakis'95].



The controller continuously observes all delays & moves

Move:

controllable edge: c

delay: λ

Winning strategy: a function that tells the controller how to move in any given state to win the game:

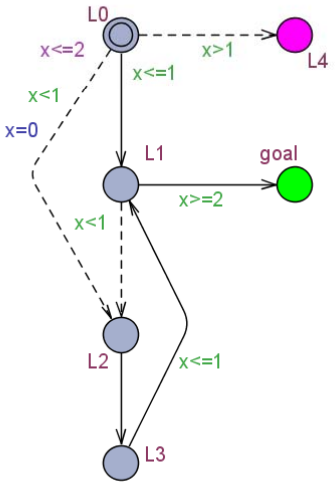
Memoryless strategy:

F : State → $E_c \cup \lambda$

Reachability Games: Reach Goal

Safety Games: Avoid loose

Timed Games



a winning strategy:

L0:

x<1	: λ
x==1	: c

L1:

x<2	: λ
x≥2	: c

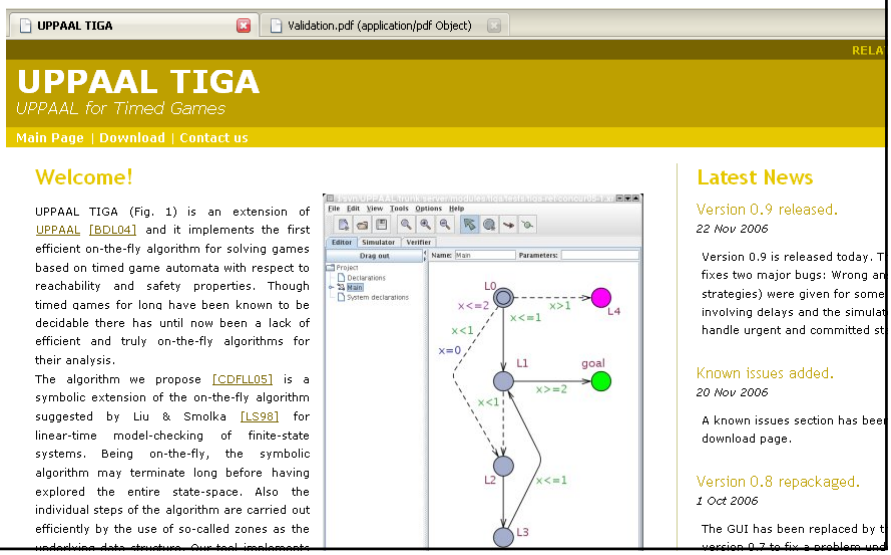
L2:

x≤1	: c

L3:

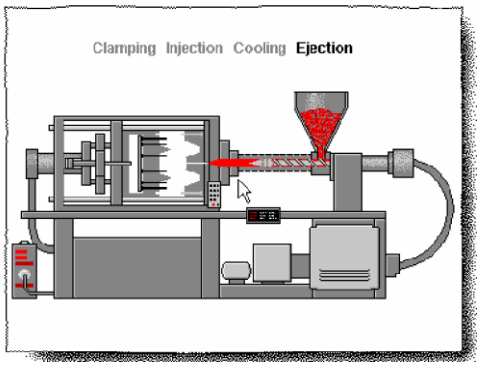
x<1	: λ
x==1	: c

Timed Game Solver



Controller Synthesis: Hydac Case

HYDAC

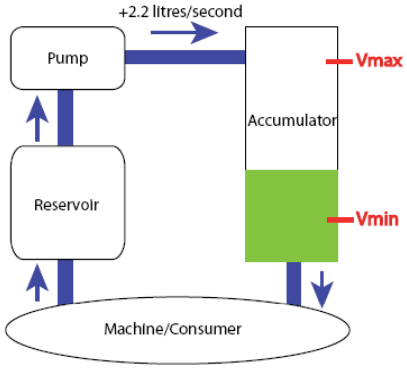


Quasimodo

Plastic Injection Molding Machine

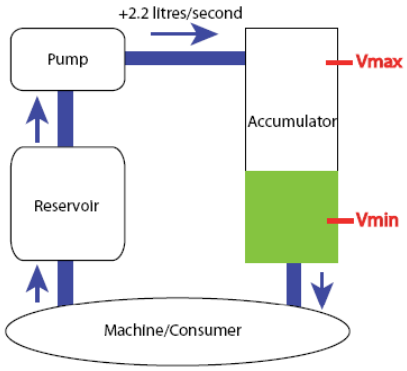
- Robust and optimal control
- Tool Chain
  - Synthesis: **UPPAAL TIGA**
  - Verification: **PHAVer**
  - Performance: **SIMULINK**
- 40% improvement of existing solutions.
- Underlying PTA problem.

The Molding Machine



- The Machine consumes oil from the Accumulator
- The Machine returns oil to the Reservoir
- The total amount of oil in the system is constant.
- The Pump can move oil from Reservoir to the Accumulator.

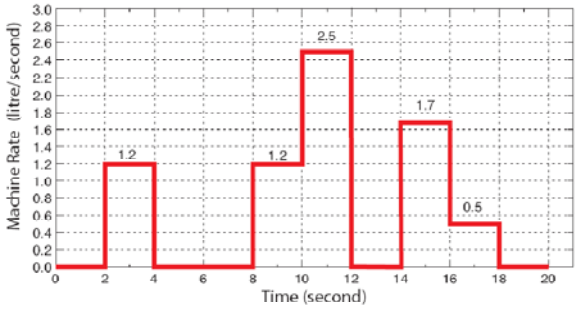
Oil Pump Control Problem



- R1: stay within safe interval [4.9,25.1]
- R2: minimize average/overall oil volume

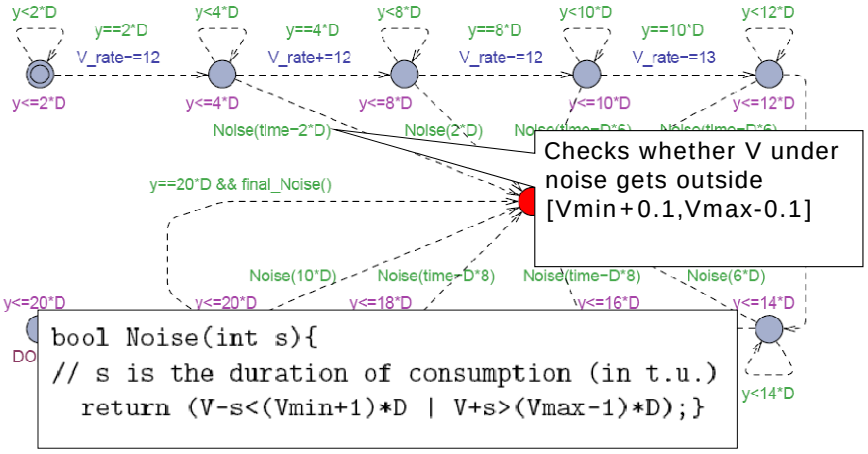
$$\int_{t=0}^{t=T} v(t)dt \,/\, T$$

The Machine (consumption)

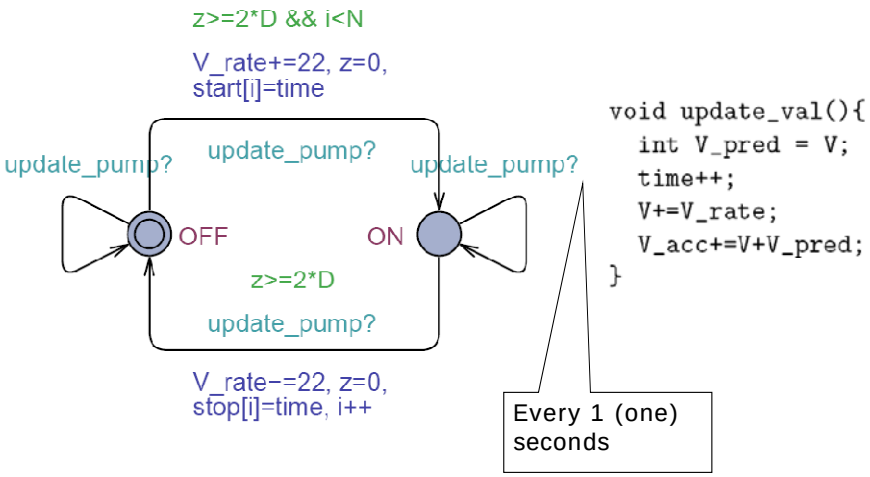


- Infinite cyclic demand to be satisfied by our control strategy.
- P: latency 2 s between state change of pump
- F: noise 0.1 l/s

Machine (uncontrollable)



Pump (controllable)



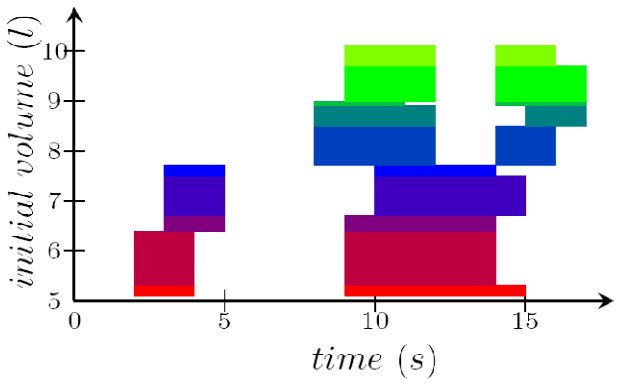
Global Approach

- Find some interval $I_1=[V_1,V_2] \in [4.9,25]$ s.t
  - at the end
  within $I_2=[V_1+m,V_1-m]$
- I_1 is optimal among all m-stable intervals.

Queries (min. K)

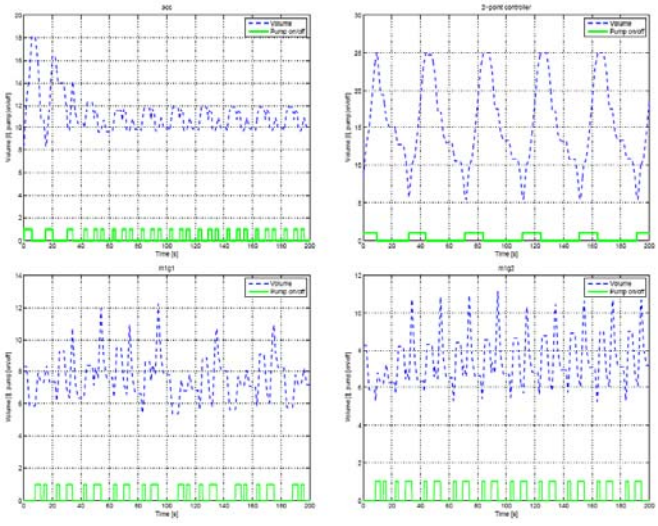
control: A<> (time=20 ∧ not BAD ∧ $V \in I_2$ ∧ V_acc≤ K)

Results



D=1, m=0.4: Optimal stable interval I_1=[5.1,10]

Results

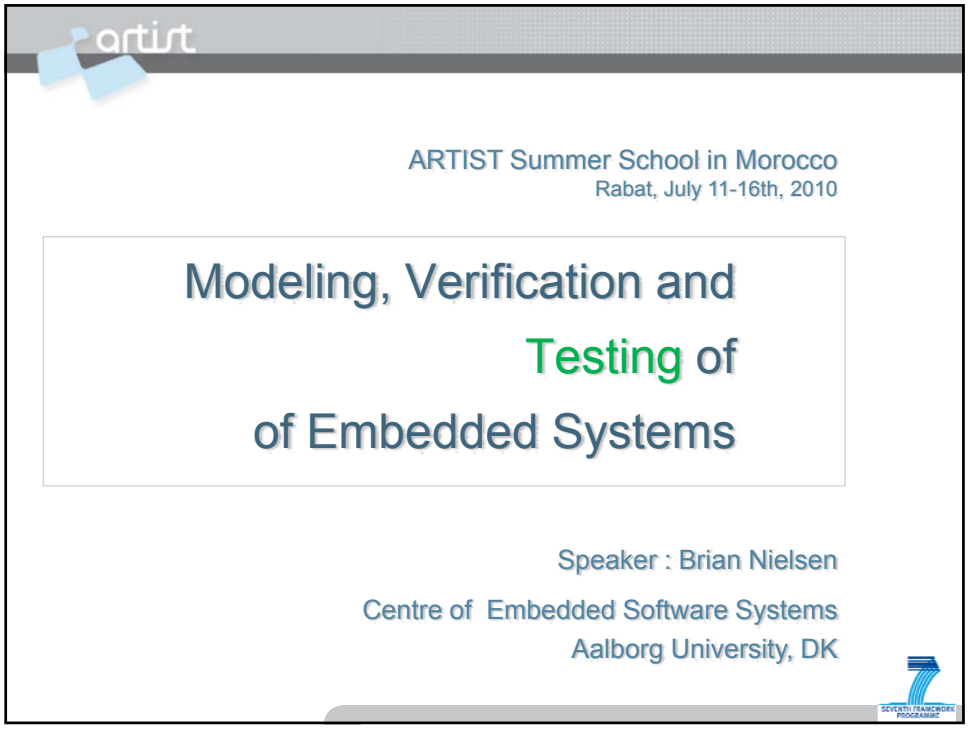


Results

Guaranteed
Correctness
Robustness

with
45% improvement in performance (BangBang)
33% improvement in perform. ("Hydac Smart")

		7.64	8.05
G2M3	1513	7.57	7.95
G2M2	1500	7.5	7.95
G2M1	1489	7.44	7.95



Modeling, Verification, and Testing of of Embedded Systems

Brian Nielsen
Centre of
Embedded Software Systems
Aalborg University, DK

bnielsen@cs.aau.dk

Center for Indlejrede Software Systemer

Course Outline

1. Introduction
2. Modeling
   1. Modelling Embedded systems
   2. Introduction to timed automata (TA)
3. Verification using Uppaal
4. Beyond Verification: Synthesis
   1. Optimal Scheduling & Planning
   2. Controller Synthesis
5. Real-Time Conformance
   1. Testing theory
   2. Real-time extensions of the ioco testing theory
6. Real-Time Test Generation
   1. Off-line generation using model checkers
   2. (optimal) quantitative test-sequences (based on Priced TA)
   3. Online real-time testing
   4. Testing strategies using Timed Games
7. Conclusions

Testing

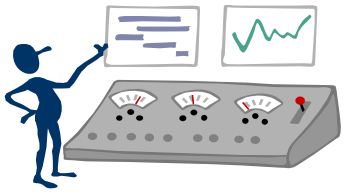
Center for Indlejrede Software Systemer

Testing

Testing:

- to check the **quality** (functionality, reliability, performance, …) of an (software) object

  -by performing experiments
  -in a controlled way

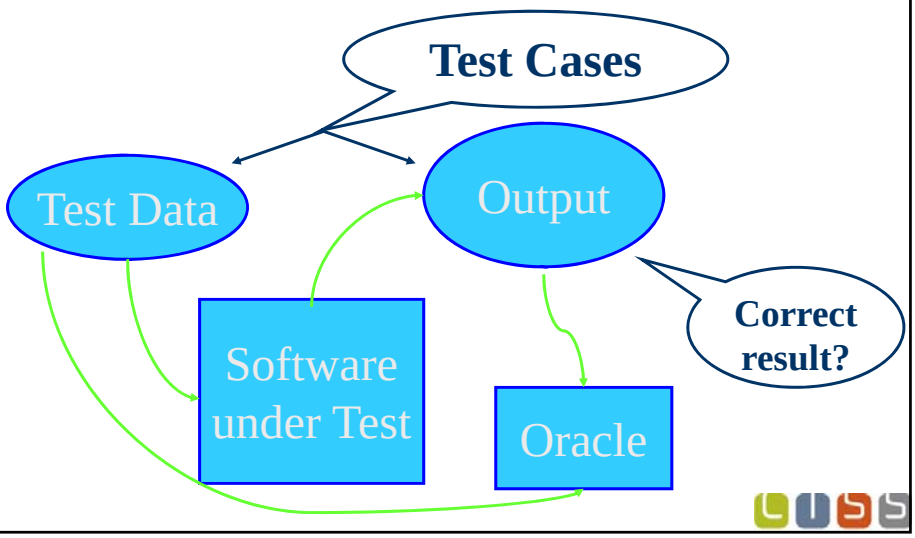


- In avg. 10-20 errors per 1000 LOC
- 30-50 % of development time and cost in embedded software

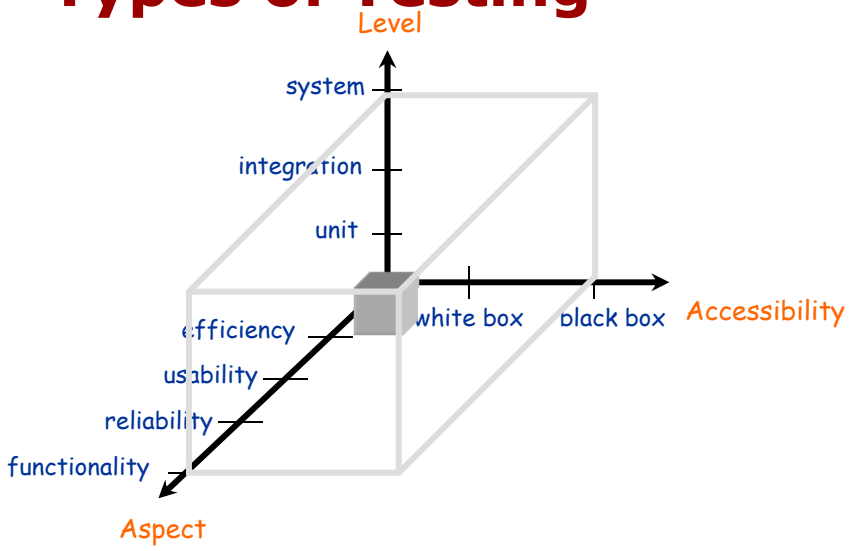
- To find errors
- To determine risk of release

What is testing?

*The **execution** of a system with sample inputs/configurations and evaluating the correctness of outputs*



Types of Testing



Quality-Characteristics (ISO-9126)

- Functionality ⇒ functional testing
  - Suitability, accuracy, security, compliance, interoperability
- Reliability ⇒ reliability testing
  - maturity, fault tolerance, recoverability
- Usability ⇒ usability testing
  - understandability, learnability, operability
- Efficiency ⇒ performance testing
  - time behaviour, resource utilization
- Maintainability ⇒ maintainability testing ??
  - Analysability, changeability, stability, testability
- Portability ⇒ portability testing ?
  - Adaptability, installability, conformance, replaceability

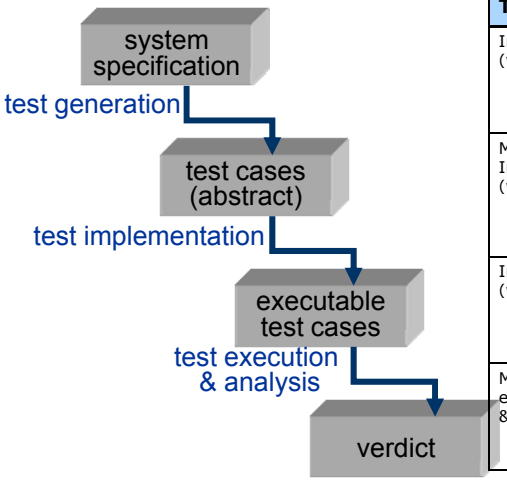
System test

- Eg Mobile Phone Protocol Testing

Test Equipment

- Complete Type Approval Test System (3 M€)

Testing Process

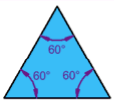


Manual Testing	Automated Testing	Model-based Testing
Informal (word)	Informal word	Formal Test model
Manual Informal (word, excell)	Manual Informal Word/excell	Automated (implicit)
Informal (word, excel)	Manually writtien Scripts	Automated (scripts or implicit)
Manual execution & Inspection	Automated Execution	Automated Execution& Evaluation

A Self-Assessment Test [Myers]

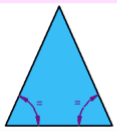
- "A program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral."
- *Write a set of test cases to test this program*

Triangles



Equilateral Triangle

Three equal sides
Three equal angles, always 60°

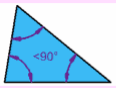


Isosceles Triangle

Two equal sides
Two equal angles

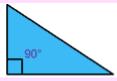
Scalene Triangle

No equal sides
No equal angles



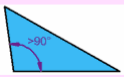
Acute Triangle

All angles are less than 90°



Right Triangle

Has a right angle (90°)



Obtuse Triangle

Has an angle more than 90°

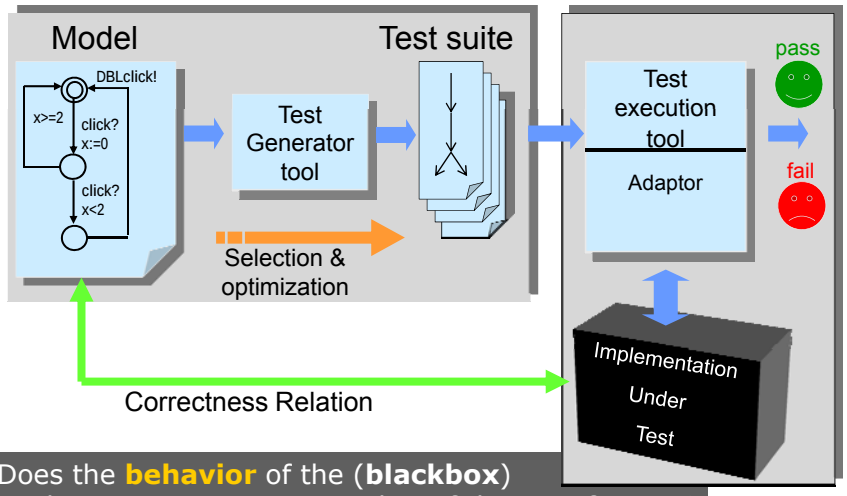
A Self-Assessment Test [Myers]

Test cases for: …

Model-based Testing

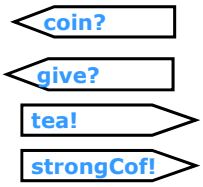
Conformance

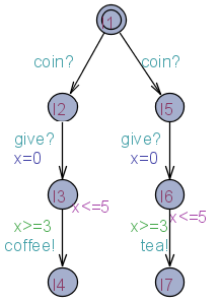
Automated Model Based Conformance Testing



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

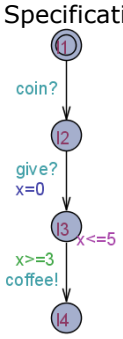
Timed Coffee Machine



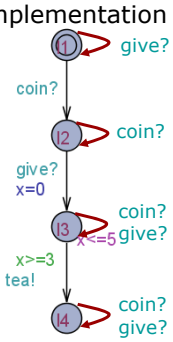


Conformance Relation

Specification

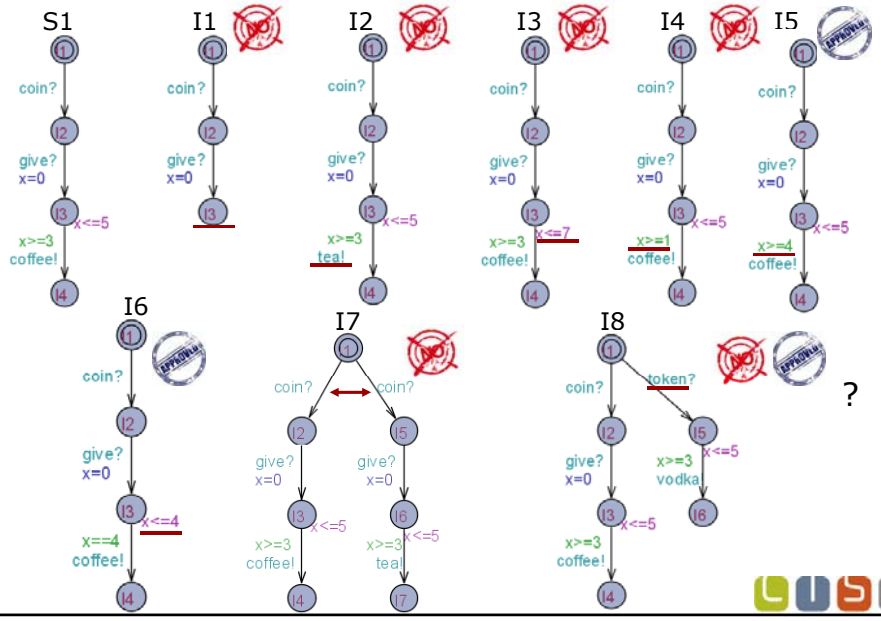


Implementation

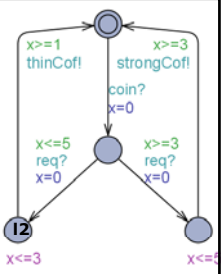


- Timed Automata with Timed-LTS semantics
- **Input** actions (?) are controlled by the environment
- **Output** actions (!) are controlled by the implementation
- Implementations are *input enabled*
- ***Testing hypothesis:*** IUT can be modeled by some (unknown) TA

Does I_n conform-to S_1 ?



Timed Conformanc



Derived from Tretman's IOCO

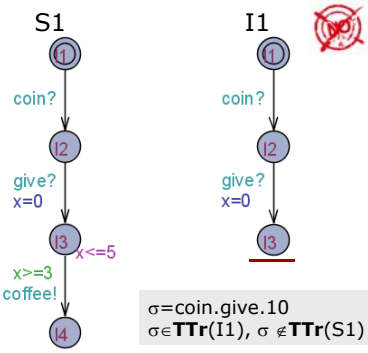
Let **I, S** be timed I/O LTS, P a set of states
TTr(P): the set of *timed traces* from P
eg.: σ = coin?.**5**.req?.**2**.thinCoffee!.**9**.coin?
Out(P **after** σ) = possible ***outputs*** and ***delays*** after σ
eg. out ({l2,x=1}): {thinCoffee, **0…2**}

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]

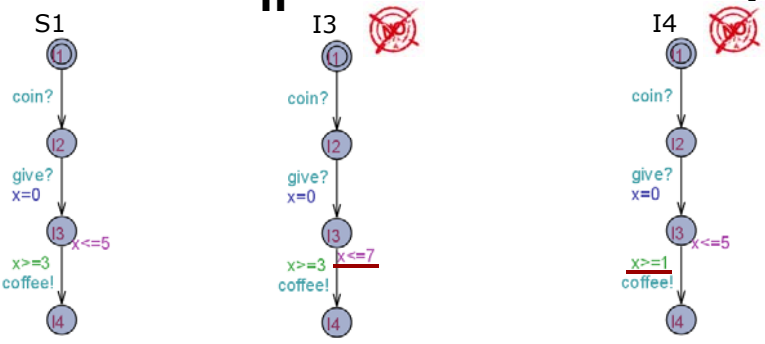
Does I_n conform-to S_1 ?



σ=coin.give.10
σ∈**TTr**(I1), σ ∉**TTr**(S1)

out(I1 **after** coin.give.3)={0... ∞}
⊄
out(S1 **after** coin.give.3)={coffee,0…2}

Does I_n conform-to S_1 ?



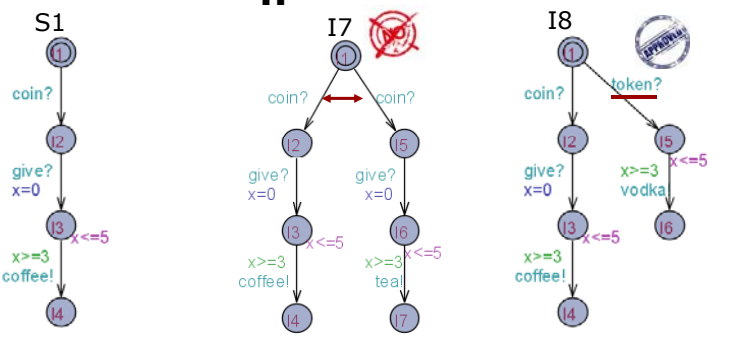
σ=coin.give.7.coffee
σ∈**TTr**(I3), σ ∉TTr(S1)

out(I3 after coin.give.7)={coffee,0}
⊄
out(S1 after coin.give.7)={}

σ=coin.give.1.coffee
σ∈**TTr**(I4), σ ∉TTr(S1)

out(I4 after coin.give.1)={coffee,0...4}
⊄
out(S1 after coin.give.1)={0...4}

Does I_n conform-to S_1 ?

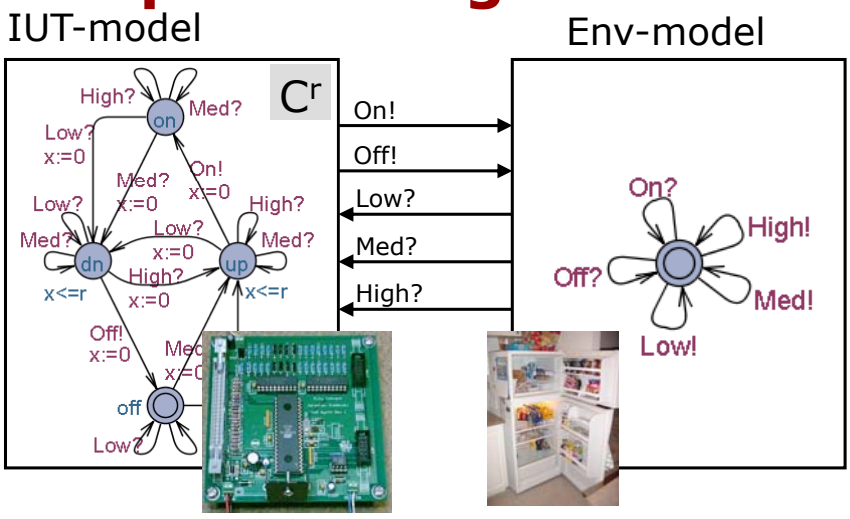


σ=coin.give.5.tea
σ∈**TTr**(I7), σ ∉**TTr**(S1)

σ=token.5.vodka
σ∈**TTr**(I8), σ ∉**TTr**(S1)
But σ was not specified

out(I7 **after** coin.give.5)={tea, coffee,0}
⊄
out(S1 **after** coin.give.5)={coffee,0}

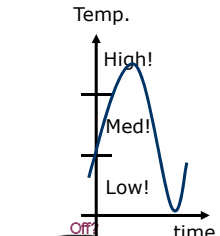
Sample Cooling Controller

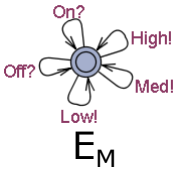


- When T is high (low) switch on (off) cooling within r secs.
- When T is medium cooling may be either on or off (impl freedom)

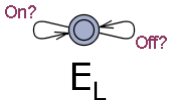
Environment Modeling

- 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)

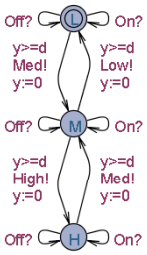




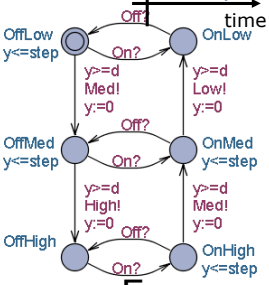
E_M



E_L



E_1

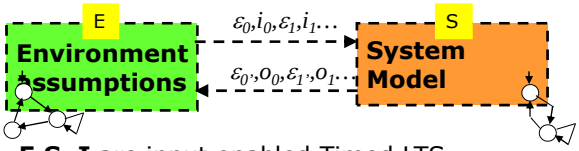


E_2

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

Conformance 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**

Re-use Testing Effort

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

1. Given new (weaker) system specification S'

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

2. Given new (stronger) environment specification E'

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

Advantages of Explicit Environments

- Realism and guiding
- Separation of concerns
- Modularity
- Creative tool uses
- Theoretical properties

Tretman's IOCO

- "*The*" conformance relation used for blackbox testing of (untimed) reactive systems
  - **Quiescence**: a state is *quiescent* iff it never produces an output (without further inputs)
  - Quiescent is an ***observable output action*** δ

i ioco s $=_{def}$ $\forall \sigma \in$ ***Straces*** **(s)** : ***out*** **(i after** σ**)** $\subseteq$ ***out*** **(s after** σ**)**

p after σ $= \{ p' \mid p \stackrel{\sigma}{\Longrightarrow} p' \}$

$p \stackrel{\delta}{\longrightarrow} p$ iff $\forall o! \in L_U \cup \{\tau\} : p \stackrel{o!}{\not\longrightarrow}$

out **(P)** $= \{ o! \in L_U \mid p \stackrel{!o}{\longrightarrow} p \in P \}$

$\cup \{ \delta \mid p \stackrel{\delta}{\longrightarrow} p, \ p \in P \}$

Straces **(s)** $= \{ \sigma \in (L \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Longrightarrow} \}$

[Jan Tretmans].

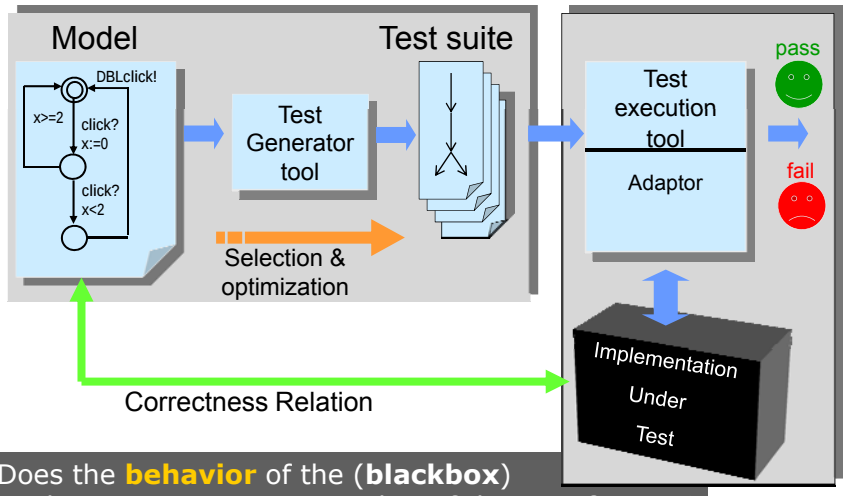
Course Outline

1. Introduction
2. Modeling
   1. Modelling Embedded systems
   2. Introduction to timed automata (TA)
3. Verification using Uppaal
4. Beyond Verification: Synthesis
   1. Optimal Scheduling & Planning
   2. Controller Synthesis
5. Real-Time Conformance
   1. Testing theory
   2. Real-time extensions of the ioco testing theory
6. Real-Time Test Generation
   1. Off-line generation using model checkers
   2. (optimal) quantitative test-sequences (based on Priced TA)
   3. Online real-time testing
   4. Testing strategies using Timed Games
7. Conclusions

Model-based Testing

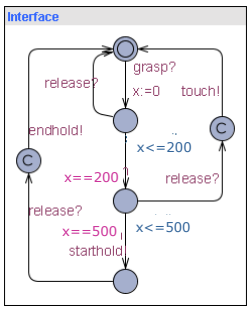
Offline

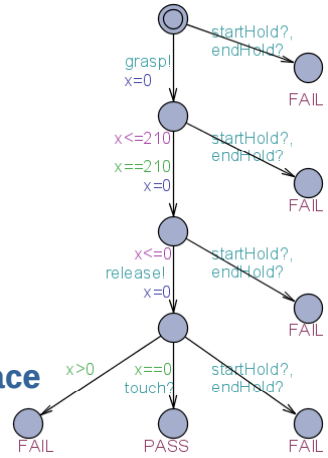
Automated Model Based Conformance Testing



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

Timed Tests





EXAMPLE test cases for **Interface**

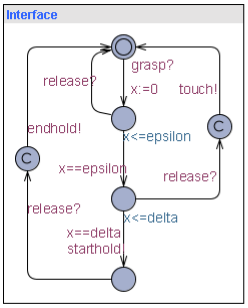
`0·grasp!·210·release!·touch?.`**PASS**

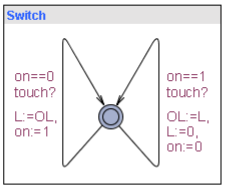
`0·grasp!·317·release!·touch?·2½·grasp!·220·release!·touch?·`**PASS**

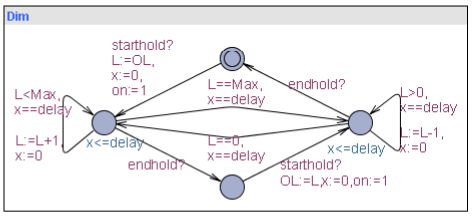
`1000·grasp!·517·starthold?·100·release!·endhold?·`**PASS**

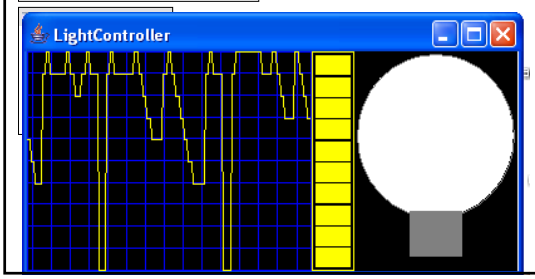
INFINITELY MANY SEQUENCES!!!!!!

DEMO: Touch-sensitive Light-Controller









Overview of Techniques

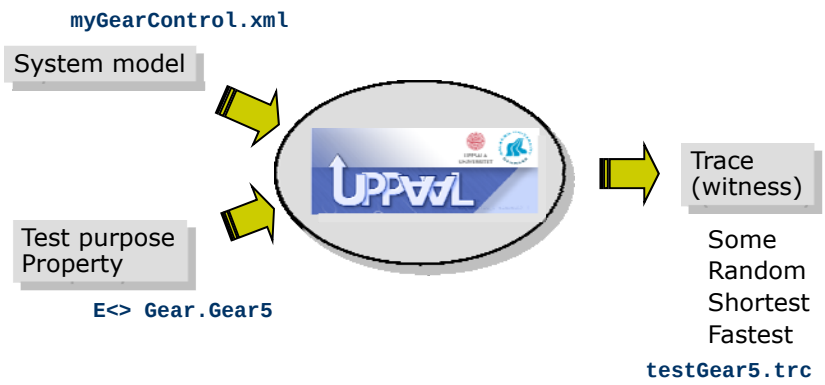
More Restricted

More Liberal

Model	Restrictions	Technique	When
”Douta”	Completely controllable	Counter Examples (Guarantees coverage/cost)	Offline
Observable TA	Timeing uncertainty+ Multiple Outputs	Game (definitely and possibly winning)	Offline (+ online)
Partially Observable TA	Observation Predicates	Game	Offline
Timed Automata	Unrestricted non-determinism	Counter Examples (Preset-input sequences only)	Offline
Timed Automata	Unrestricted non-determinism	Stat-set tracking	Online

Test Generation

using Verification



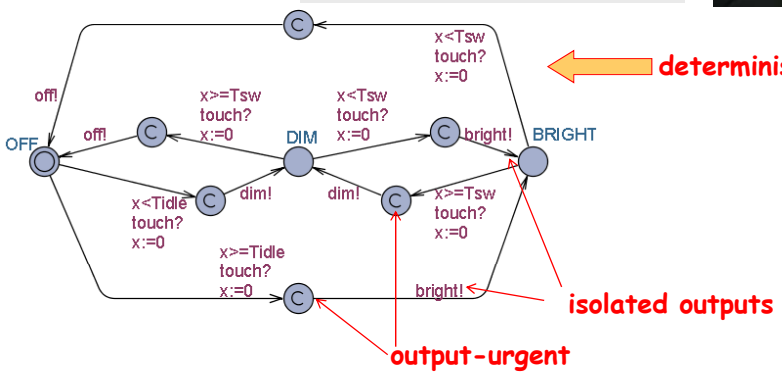
Use trace scenario as test case??!!

Controllable Timed Automata

- "DOUTA"-Model
  - **Determinism**: for any state, two transitions with same input/output leads to the same next state
  - **Output Urgent**: enabled outputs will occur immediately
  - **Isolated Outputs**: if an output is enabled, no other output is enabled
  - **Input Enabled**: all inputs can always be accepted

"Controllable" Timed I/O Automata

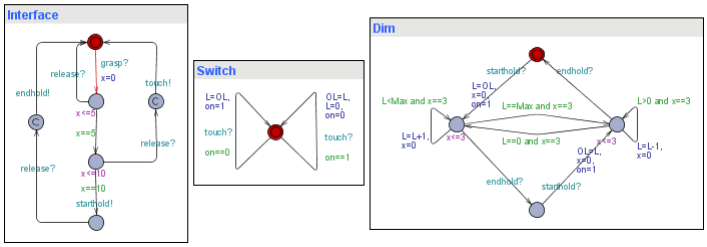
Inputs (?) are controllable
Outputs (!) are uncontrollable



- Test case is a **preset sequence** of timed I/O actions
- Time and resource **optimal** tests can be generated

Test Purposes

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



TP: Check that the light can become bright:

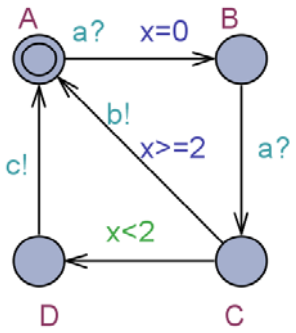
E<> L==10

```
out(IGrasp);silence(500);in(OSetLevel,0);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);
out(IRelease);
```

Shortest - and fastest –test !

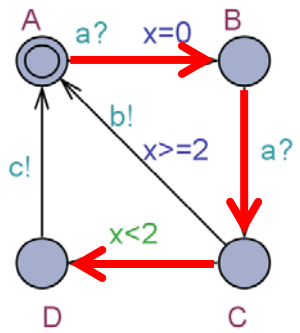
Coverage Based Test Generation

- 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

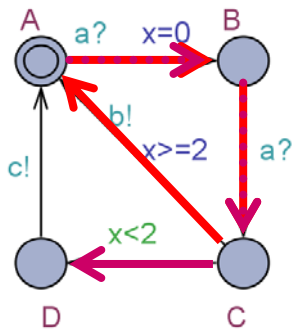


Coverage Based Test Generation

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

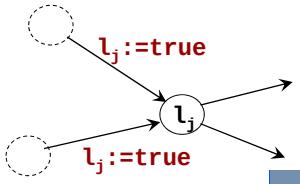
Coverage Based Test Generation

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



Location Coverage

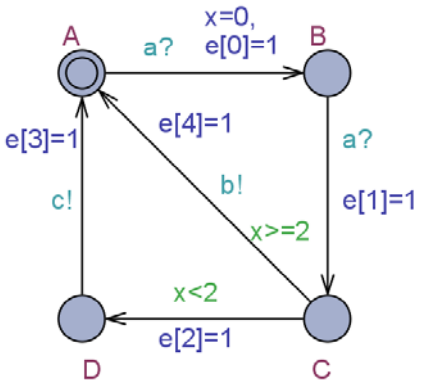
- Test sequence traversing all locations
- Encoding:
  - Enumerate locations $l_0, \ldots, l_n$
  - Add an auxiliary variable l_i for each location
  - Label each ingoing edge to location i l_i:=true
  - Mark initial visited l_0:=true
- Check: E<>(l_0=true $\wedge$ … $\wedge$ l_n=true)



UPPAAL COVER

Edge Coverage

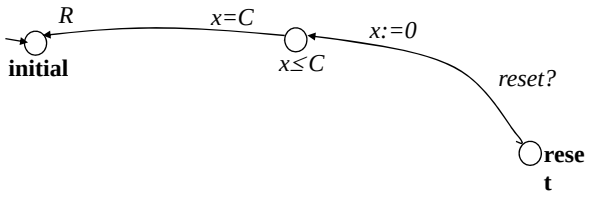
- Test sequence traversing all edges
- Encoding:
  - Enumerate edges
    $e_0, \ldots, e_n$
  - Add auxiliary variable **e[i]** for each edge
  - Label each edge **e[i]:=1**



- Check:
  E<>(e[0]=1 ∧ … ∧ e[n]=1)

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)



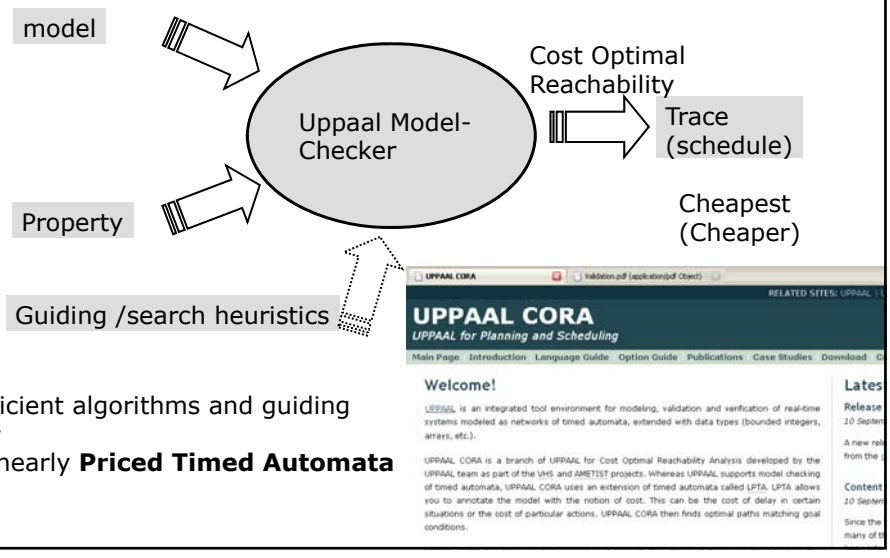
- Same encodings and min-cost reachability
- Test sequence $\sigma = \mathcal{E}_0, i_0, \ldots, \mathcal{E}_1, i_1$, **reset** $\underbrace{\mathcal{E}_2, i_2, \ldots, \mathcal{E}_0, i_0}_{\sigma_i}$, **reset**, $\mathcal{E}_1, i_1, \mathcal{E}_2, i_2, \ldots$
- Test suite $T = \{\sigma_1, \ldots, \sigma_n\}$ with minimum cost

Time-optimal test suites

- Product instance testing
- Test more behavior in less time
- Some operations (e.g SUT **reset**) are very time-consuming
- Stressful for SUT??

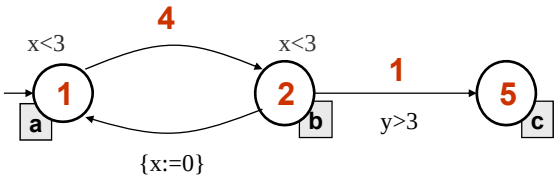
- Other resources
  - Power
  - Mechanical wear
  - Manual operations

Test generation using Optimal Scheduling



Efficient algorithms and guiding for
Linearly **Priced Timed Automata**

Linearly Priced Timed Automata



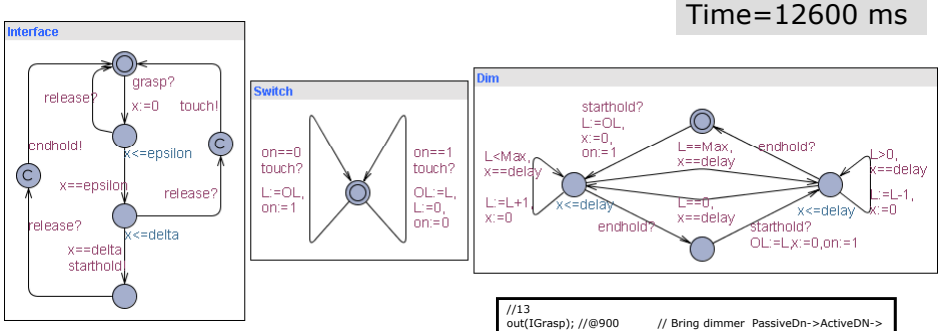
- Timed Automata + costs on transitions and locations
- Cost of performing transition: transition cost
- Cost of performing delay ε : (ε x location cost)
- Trace:

$$(\mathbf{a},x=y=0) \xrightarrow[4]{} (\mathbf{b},x=y=0) \xrightarrow[2.5 \times 2]{\varepsilon(2.5)} (\mathbf{b},x=y=2) \xrightarrow[0]{} (\mathbf{a},x=0,y=2)$$

- Cost of Execution Trace:
  - Sum of costs: **4 + 5 + 0 = 9**

Fastest Edge Coverage

Time=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);

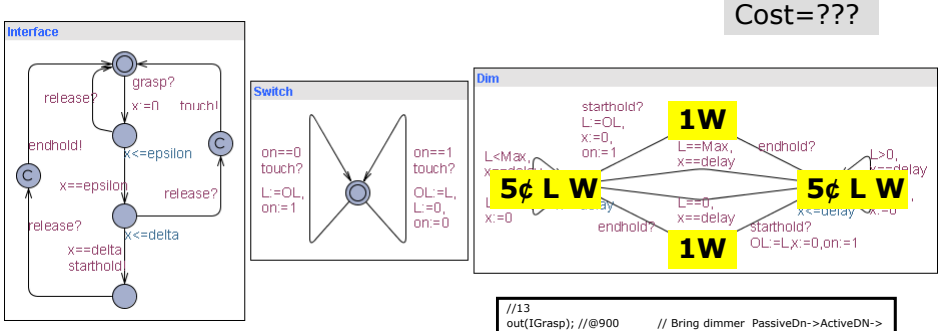
//9
out(IGrasp); //@400       //Bring dimmer from ActiveUp
silence(500); //hold      //To Passive DN (level=0)
in(OSetLevel,0);
out(IRelease);
```

```
//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);
```

Power-Optimal Edge Coverage

Cost=???




```
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);

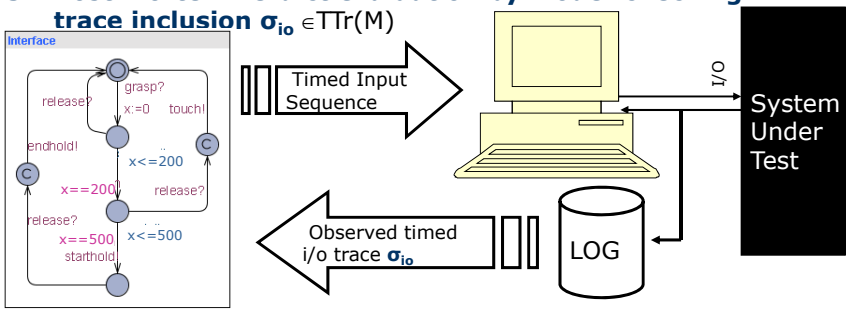
//9
out(IGrasp); //@400       //Bring dimmer from ActiveUp
silence(500); //hold      //To Passive DN (level=0)
in(OSetLevel,0);
out(IRelease);
```

```
//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);
```

Offline Testing of Non-Deterministic TA

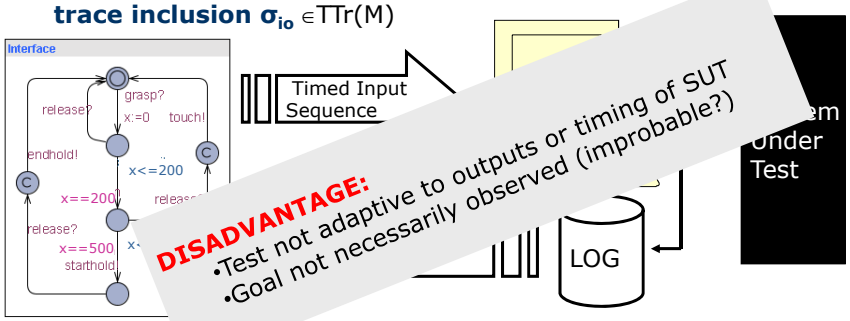
1. **Compute "preset" timed input-sequence σ_i**
2. **Blindly Execute input sequence and log i/o sequence σ_{io}**
3. **Post mortem verdict evaluation by model-checking trace inclusion $\sigma_{io} \in TTr(M)$**



FAIL: $\sigma_{io} \notin TTr(M)$
PASS: INCONC $\sigma_{io} \in TTr(M)$ and goal-state possible reached
INCONC: $\sigma_{io} \in TTr(M)$ but goal state not reachable
Can be anserwed using Uppaal reachability analysis of σ_{io} || M

Offline Testing of Non-Deterministic TA

1. **Compute "preset" timed input-sequence σ_i**
2. **Blindly Execute input sequence and log i/o sequence σ_{io}**
3. **Post mortem verdict evaluation by model-checkin**
   trace inclusion $\sigma_{io} \in$ TTr(M)



FAIL: $\sigma_{io} \notin$ TT
PASS: INCONC **$\sigma_{io} \in$** TTr(M) and goal-state possibly reached
INCONC: $\sigma_{io} \in$ TTr(M) but goal state not reachable
Can be anserwed using Uppaal reachability analysis of **σ_{io}** || M

Testing

On-Line

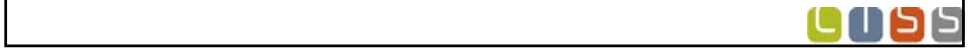
Center for Indlejrede Software Systemer

Overview of Techniques

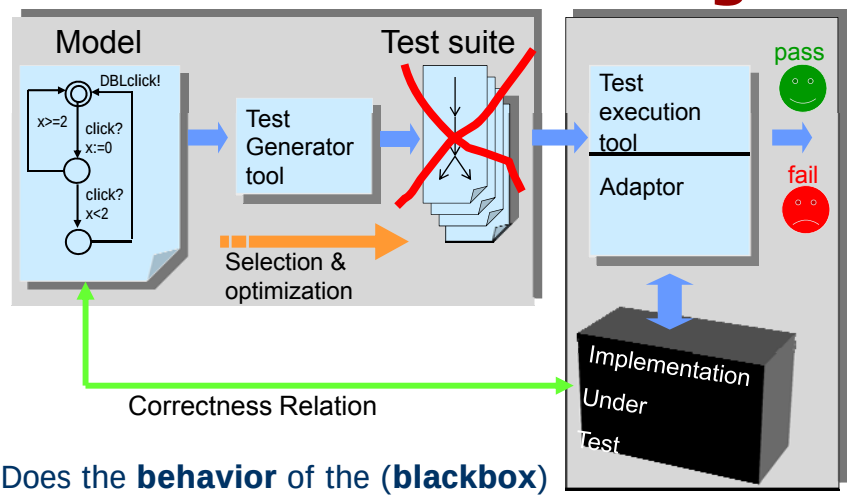
More Restricted

More Liberal

Model	Restrictions	Technique	When
”Douta”	Completely controllable	Counter Examples (Guarantees coverage/cost)	Offline
Observable TA	Timeing uncertainty+ Multiple Outputs	Game (definitely and possibly winning)	Offline (+ online)
Partially Observable TA	Observation Predicates	Game	Offline
Timed Automata	Unrestricted non-determinism	Counter Examples (Preset-input sequences only)	Offline
Timed Automata	Unrestricted non-determinism	Stat-set tracking	Online

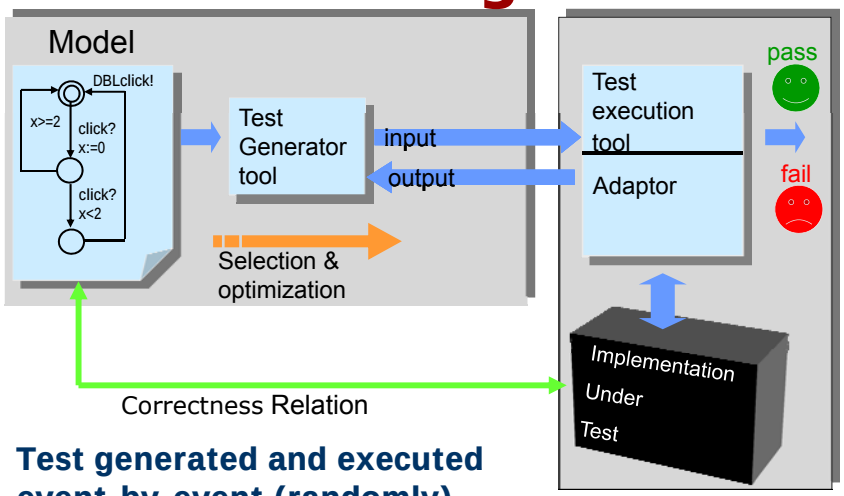


Automated Model Based Conformance Testing



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

Online Testing



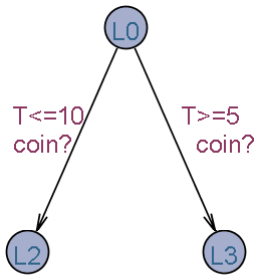
Test generated and executed event-by-event (randomly)

A.K.A on-the-fly testing

Non-Determinism

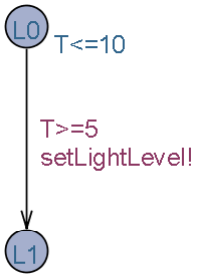
Initially T=0

Transitions / Locations



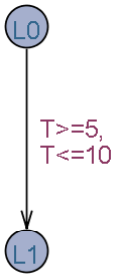
Between 5 and 10 coin leads to L2 or L3

Timing Uncertainty



LightLevel must be adjusted between 5 and 10

Internal actions (+ timing)

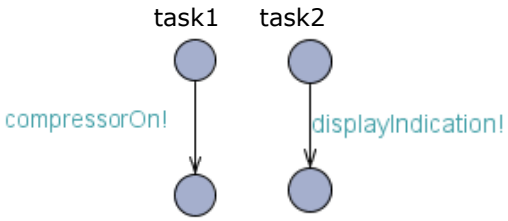


Internal transition may be taken between 5 and 10

Non-Determinism

Modeling Action uncertainty

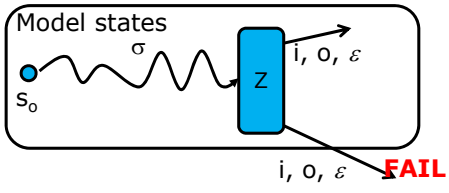
Event output ordering of two concurrent tasks in the IUT may be unknown



CompressorOn then displayIndication, or displayIndication then compressorOn???

Algorithm Idea:
State-set tracking

- Dynamically compute all potential states that the model M can reach after the timed trace
  $\sigma = \varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2, \ldots$ [Tripakis] Failure Diagnosis
- $Z = M$ ***after*** $(\varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2)$
- *If $Z = \emptyset$ the IUT has made a computation not in model:* ***FAIL***
- *i is a relevant input in Env iff $i \in EnvOutput(Z)$*

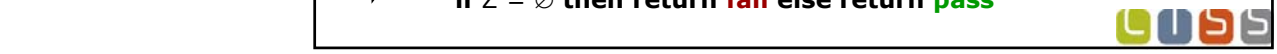


(Abstract) Online Algorithm

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* i
2. // 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**



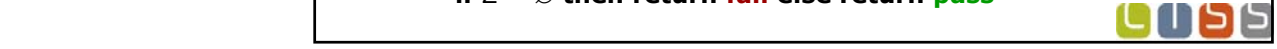
(Abstract) Online Algorithm

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
      Z := Z
2. // wait d f
   randomly
   wait (for
   if o occurr
      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**

Sound
Complete (as T → ∞)
(Under some technical assumptions)

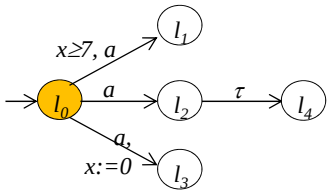


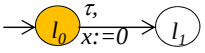
State-set computation

- Compute all potential states the model can occupy after the timed trace $\varepsilon_0, i_0, \varepsilon_1, o_1, \varepsilon_2, i_2, o_2, \ldots$
- *Let Z be a set of states*

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

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





{ ⟨l_0,x=3⟩ } **after** *a* =
{ ⟨l_2,x=3⟩, ⟨l_4, x=3⟩, ⟨l_3, x=0⟩ }

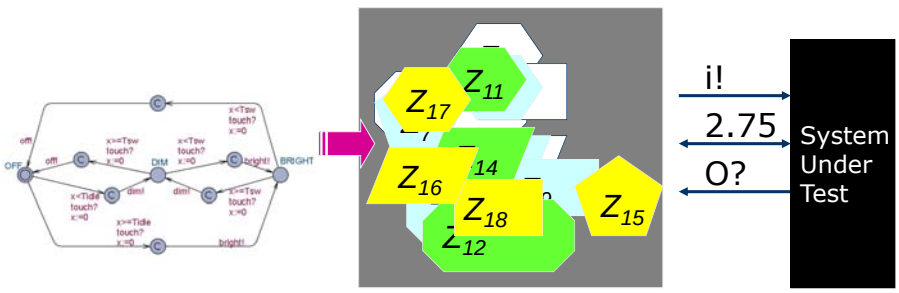
{ ⟨l_0,x=0⟩} **after** 4 =
{ ⟨l_0,x=4⟩, ⟨l_1, 0 ≤ x ≤ 4⟩ }

$\langle l_0, x=0\rangle \xrightarrow{1} \langle l_0, x=1\rangle \xrightarrow{\tau} \langle l_1, x=0\rangle \xrightarrow{3} \langle l_1, x=3\rangle$

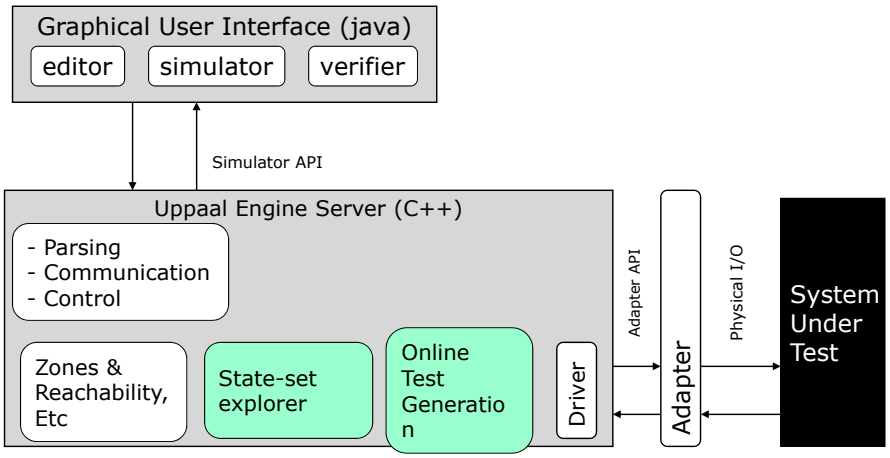
Real-time Online

Specification
TA-network

State-set explorer:
maintain and analyse a *set* of *symbolic* states (zones) in real time!

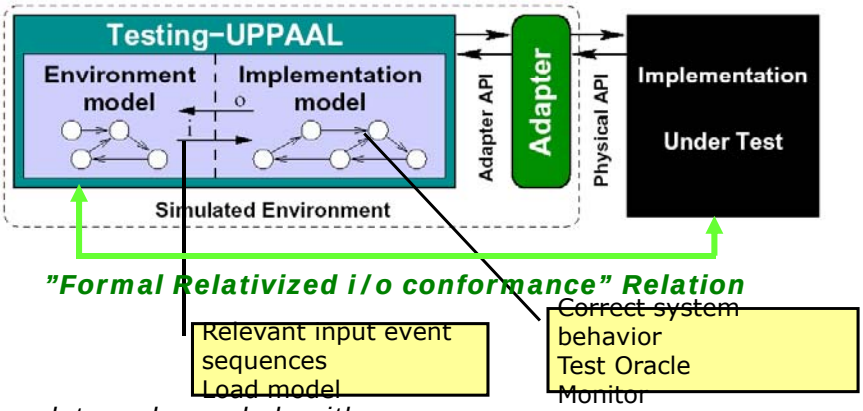


Tron: implementation



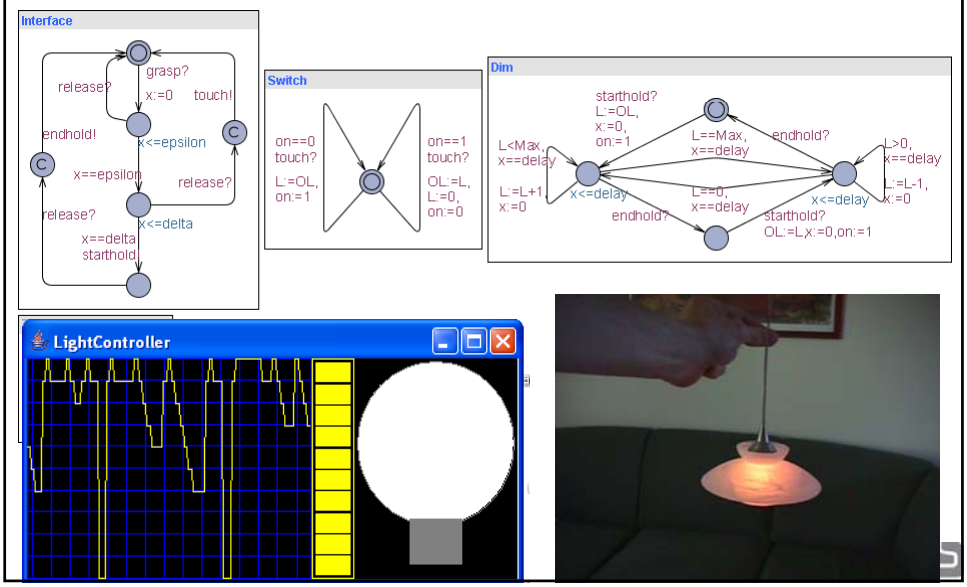
Our Framework

UppAal Timed Automata *Network: Env || IUT*



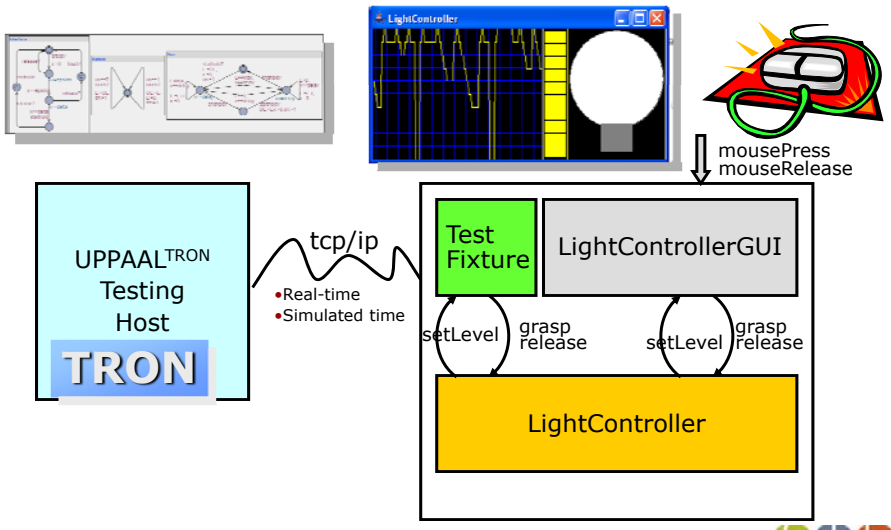
Complete and sound algorithm
Efficient symbolic reachability algorithms
UppAal-TRON: Testing Real-Time Systems Online
Release 1.4 **http://www.cs.aau.dk/~marius/tron/**

DEMO:
Touch-sensitive Light-Controller



On-line Testing

Light Controller



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 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

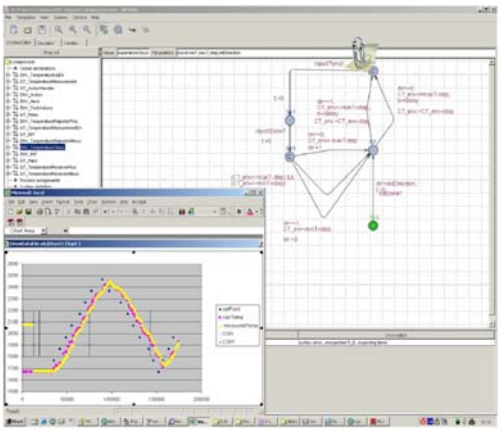
- Optional real-time clock or LON network module

Industrial Cooling Pla

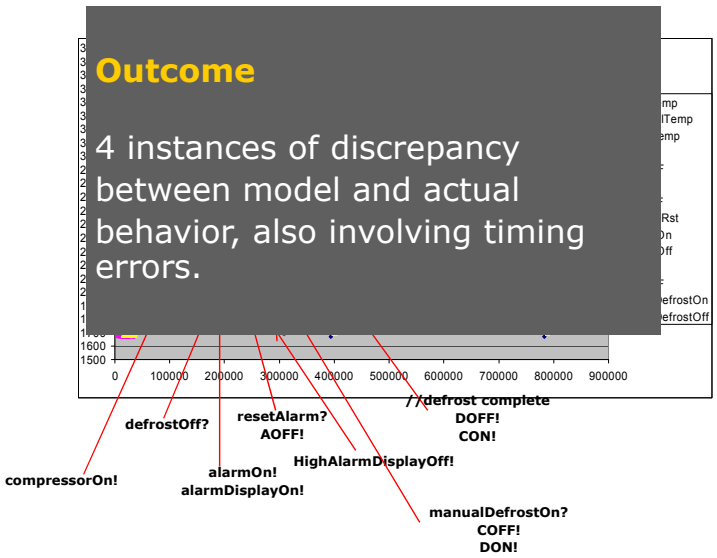
Industrial Applicat

Danfoss Electronic Cooling Controller

18 timed automata components
14 clocks, 14 integers



Example Test Run



Offline Testing of Uncontrollable Timed Systems

Center for Indlejrede Software Systemer

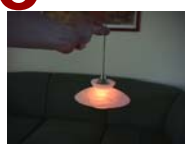
Overview of Techniques

More Restricted

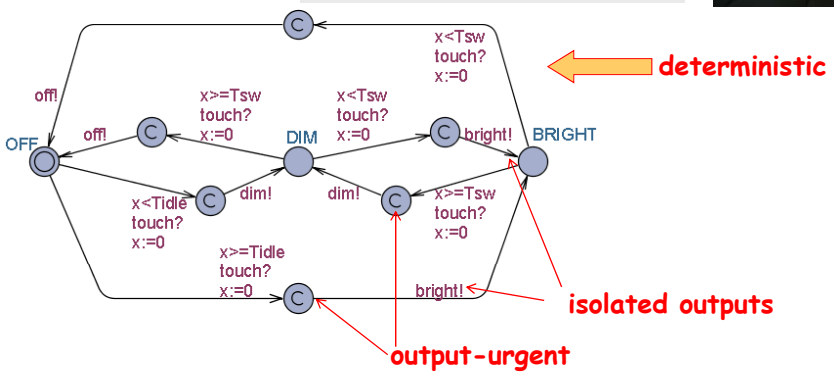
More Liberal

Model	Restrictions	Technique	When
”Douta”	Completely controllable	Counter Examples (Guarantees coverage/cost)	Offline
Observable TA	Timeing uncertainty+ Multiple Outputs	Game (definitely and possibly winning)	Offline (+ online)
Partially Observable TA	Observation Predicates	Game	Offline
Timed Automata	Unrestricted non-determinism	Counter Examples (Preset-input sequences only)	Offline
Timed Automata	Unrestricted non-determinism	Stat-set tracking	Online

“Controllable” Timed I/O Automata



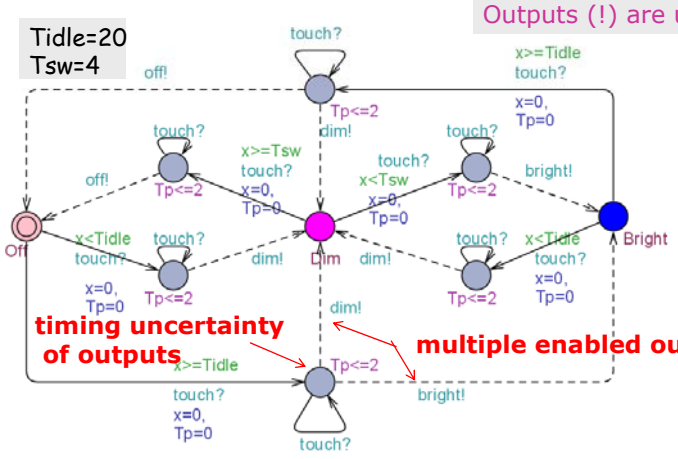
Inputs (?) are controllable
Outputs (!) are uncontrollable



- Test case is a **preset sequence** of timed I/O actions
- Time and resource **optimal** tests can be generated

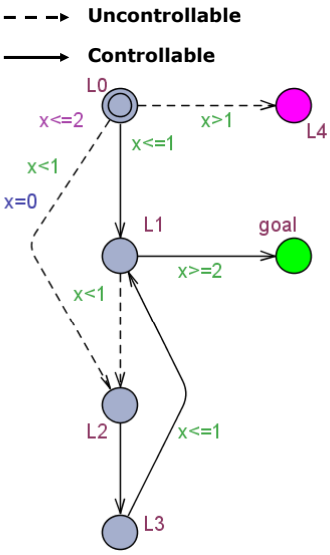
TA with Uncertainty

Inputs (?)are controllable
Outputs (!) are uncontrollable



Timed Game Automata

[Maler, Pnueli, Sifakis'95].



The controller continuously observes all delays & moves

Move:

controllable edge: c

delay: λ

Winning strategy: a function that tells the controller how to move in any given state to win the game:

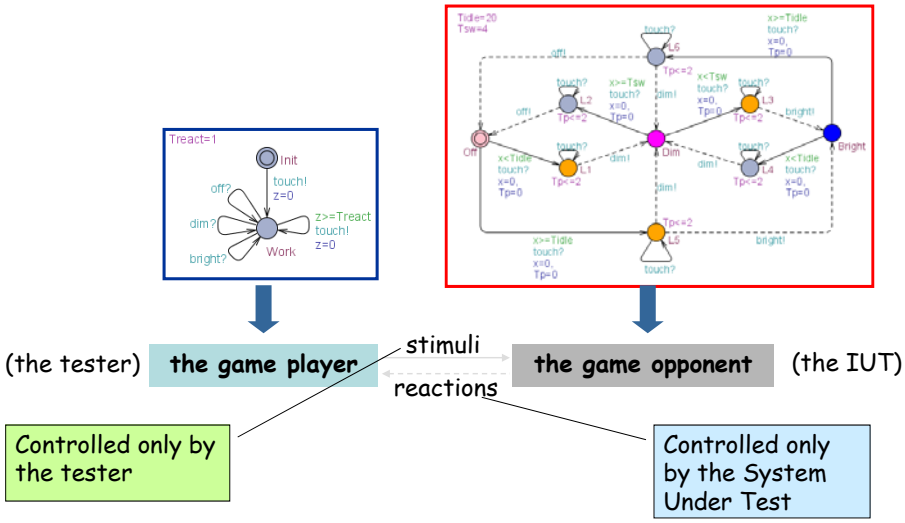
Memoryless strategy:

F : State → $E_c \cup \lambda$

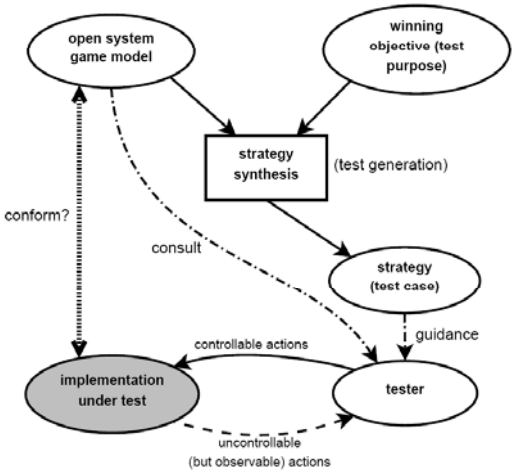
Reachability Games: Reach Goal

Safety Games: Avoid loose

Testing as Playing Games



Game Strategy as Test Case

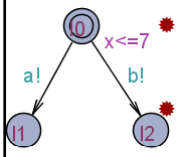


Given an ACTL reachability test purpose φ :
Generate a winning strategy for φ as the test case.

In a game-theoretic context (i.e., no matter what the uncontrollable actions the SUT executes ...)

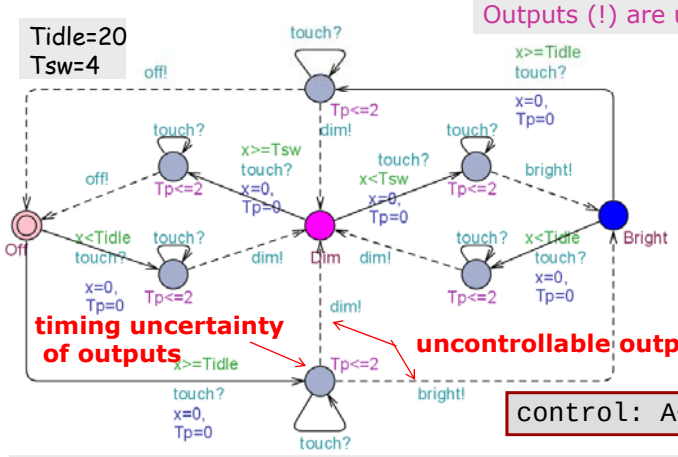
Timed Games and Test Generation

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



Observable Timed Automata

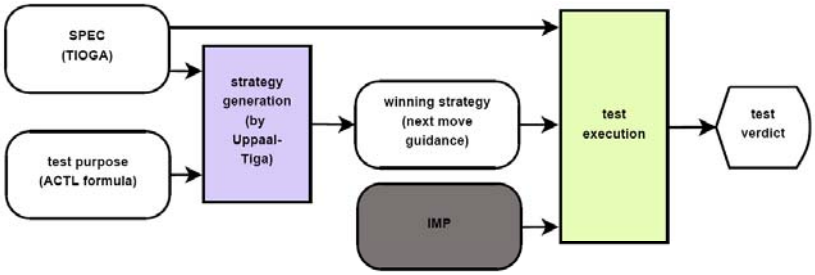
Inputs (?) are controllable
Outputs (!) are uncontrollable




```
control: A<> Bright
```

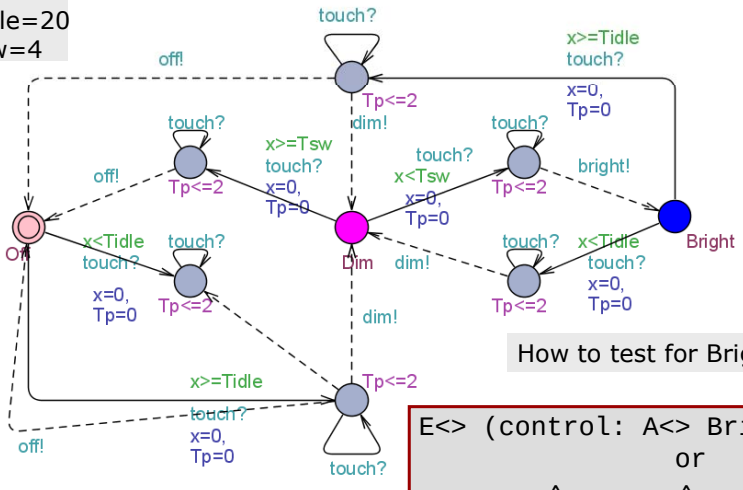
Off-line test-case generation =
Compute winning strategy for reaching **Bright**
Assign verdicts st. lost game means IUT not conforming

Timed Games for Testing



A trick light control

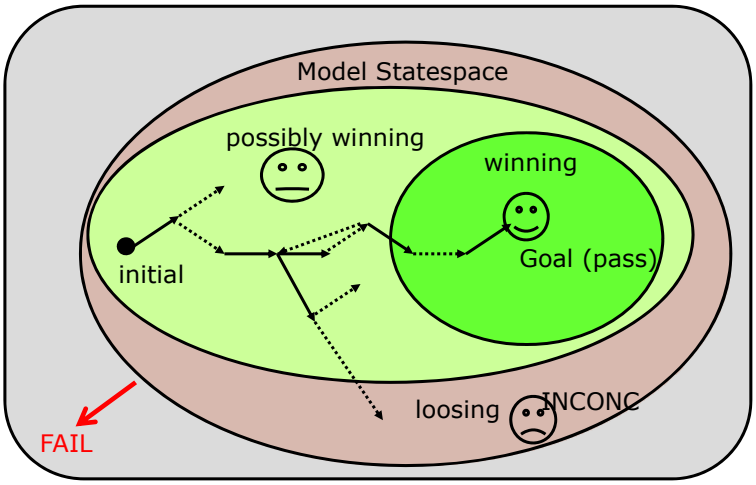
Tidle=20
Tsw=4



How to test for Bright ?

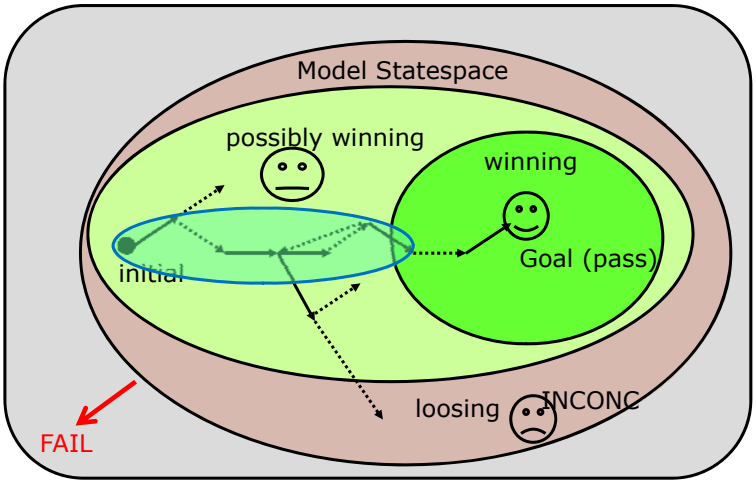
E<> (control: A<> Bright)
or
<<c,u>> ◊(<<c>> ◊ Bright)

Cooperative Strategies



Uppaal-Tiga extended to compute this partitioning motivated by testing applications

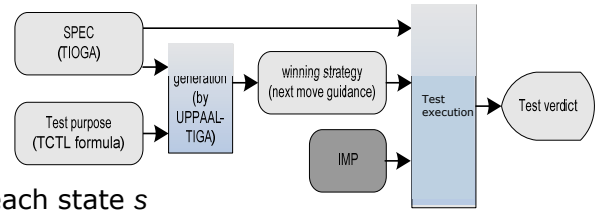
Generate test case



- Choose & prune sub-tree of cooperative states
- Convert to suitable test notation
- with verdicts according to RT-IOCO.

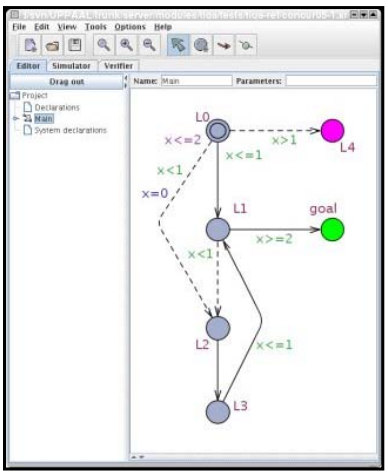
Executing Test Strategies

winning strategy ≠ test case (lacks test verdicts)

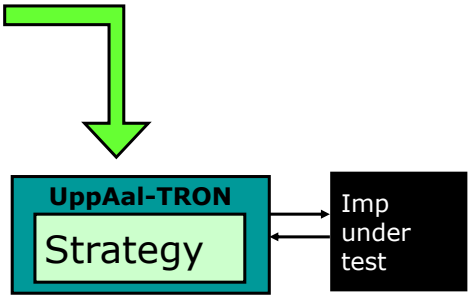


- At each state *s*
  - The tester monitors outputs and delays
  - If a disallowed output or delay occurs (RT-IOCO), declare "FAIL";
  1. If *s* is cooperative, then according to $\mathbf{F_c(s)}$ either
     - offer a random enabled inputs to IUT or
     - delay random
  2. If *s* is winning, then deterministically according to $\mathbf{F_w(s)}$
     - offer input to IUT or
     - delay
  3. If *s* is a goal-state, declare "PASS".
  4. If *s* is loosing, declare "INCONC"
- Untill verdict, or max test duration elapses

Online execution of Testing Games



Cooperative or Definitely (Winning) Strategy



Overview of Techniques

More Restricted

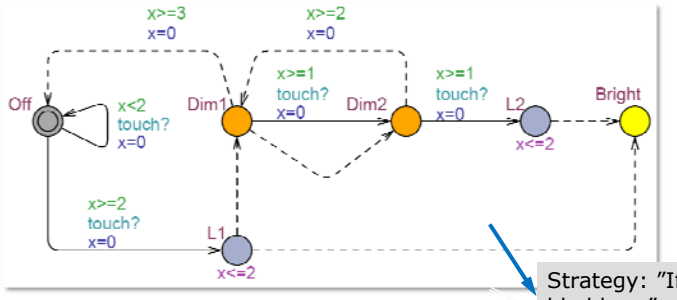
More Liberal

Model	Restrictions	Technique	When
”Douta”	Completely controllable	Counter Examples (Guarantees coverage/cost)	Offline
Observable TA	Timeing uncertainty+ Multiple Outputs	Game (definitely and possibly winning)	Offline (+ online)
Partially Observable TA	Observation Predicates	Game	Offline
Timed Automata	Unrestricted non-determinism	Counter Examples (Preset-input sequences only)	Offline
Timed Automata	Unrestricted non-determinism	Stat-set tracking	Online

However,

- Full Observability is not always feasible
- We may have:
  - Inaccurate meassurements of SUT
    - limited number of IUT probings or instrur
    - limited precision sensors, e.g., ”x∈[0, 2)” rather than ” x∈[0, 1)”
  - Components interactions inside SUT
    - coupling I/O actions between two SUT components leads to silent transitions (internal state changes)
  - Tester cannot report (infer) the exact SUT state
  - Cannot use state-based strategy

Partially Observable Systems



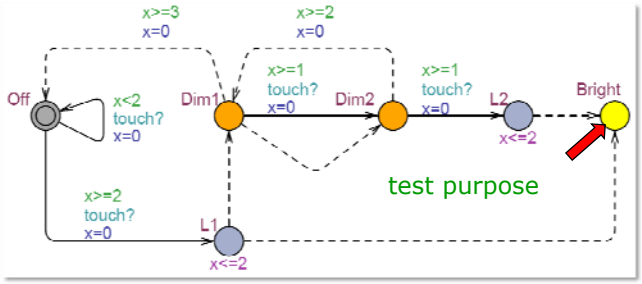
Strategy: "If in **Dim1** then bla bla bla ..."
- We can tell it is in "Dim1 or Dim2" ☺ ; but not exactly which one ☹

Need new style strategy!

What if :
- Locations **Off** and **Bright** can be sensed;
- **Dim1** and **Dim2** **are indistinguishable**
- Other locations (L1, L2): don't care;
- Clock **y** can only be checked if y ∈ [0, 1).

Specifying Observations

Smart Light Controller

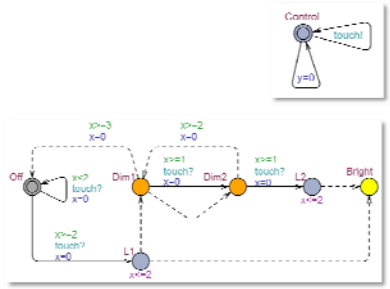


Using a set of **observable predicates**:
(In some location?, clocks satisfy some constraints?)

```
e.g.,
{ ({Off}, true),
  ({Dim1,Dim2}, true),
  ({Bright}, true),
  ( L, 0=<y<1) }
```

Test Generation for Partially Observable Systems

PO-TGA models



observable predicates

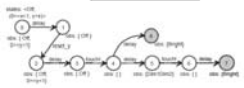
```
{ ({Off}, true),
  ({Dim1,Dim2}, true),
  ({Bright}, true),
  ( L, 0=<y<1) }
```

test purpose

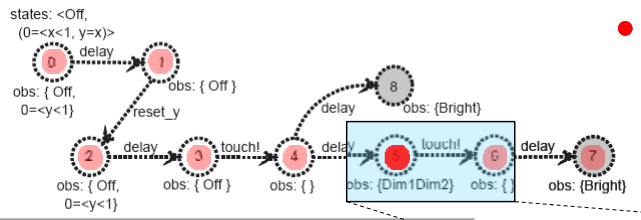
```
control: A<> Bright
```

Uppaal-TIGA

winning game strategy

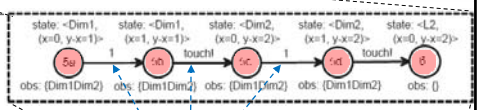


Playing OBSI Strategy

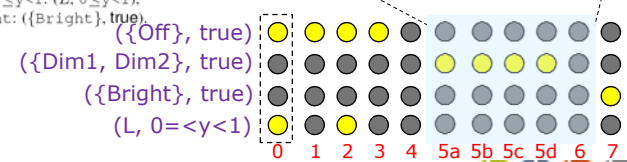


Node	TA symbolic states	Observations
0	{<Off, (0≤x<1, y-x=0)>}	{Off, 0≤y<1}
1	{<Off, (x=1, y-x=0)>}	{Off}
2	{<Off, (1≤x<2, y-x=-1)>}	{Off, 0≤y<1}
3	{<Off, (x=2, y-x=-1)>}	{Off}
4	{<L1, (0≤x≤2, y-x=1)>}	{ }
5	{<Dim1, (0≤x<3, y-x=1)>, <Dim2, (0≤x<3, 2≤y-x<4)>}	{Dim1Dim2}
6	{<L2, (0≤x≤2, 2≤y-x<5)>}	{ }
7	{<Bright, (0≤x≤2, 2≤y-x<5)>}	{Bright}
8	{<Bright, (0≤x≤2, y-x=1)>}	{Bright}

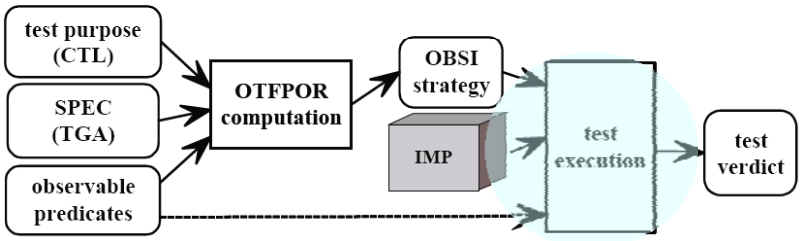
Observable predicates: Off: ({Off}, true), 0≤y<1: (L, 0≤y<1), Dim1Dim2: ({Dim1, Dim2}, true), Bright: ({Bright}, true).



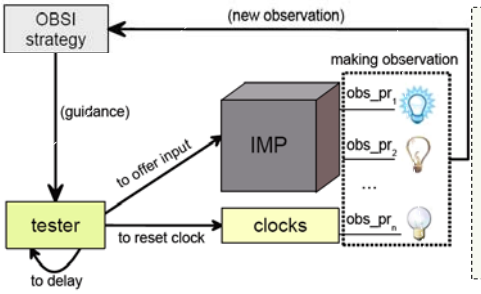
stuttering steps



Testing Partially Observable Timed Systems



Test Execution for Partially Observable Timed Systems



Sketch of Test Execution Algorithm:

1. If goal observation is reached, then "**pass**"; else continue;
2. Offer input or do a delay as instructed by the strategy, until the observation changes;
3. If the new observation is allowed, then continue on, otherwise "**fail**"

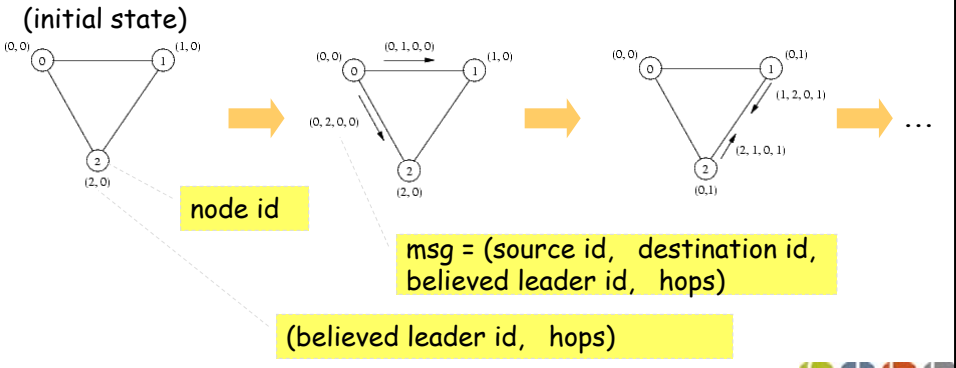
Soundness

Completeness (w.r.t. test purpose)

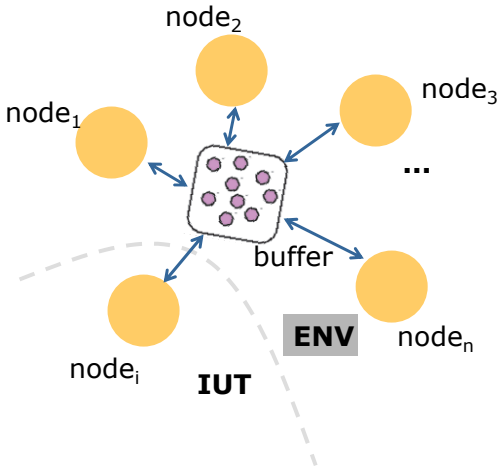
NB: partial observation-based conformance (**poco**): Whatever observations made on the IMP, these observations should be allowed by the SPEC model.

Case Study

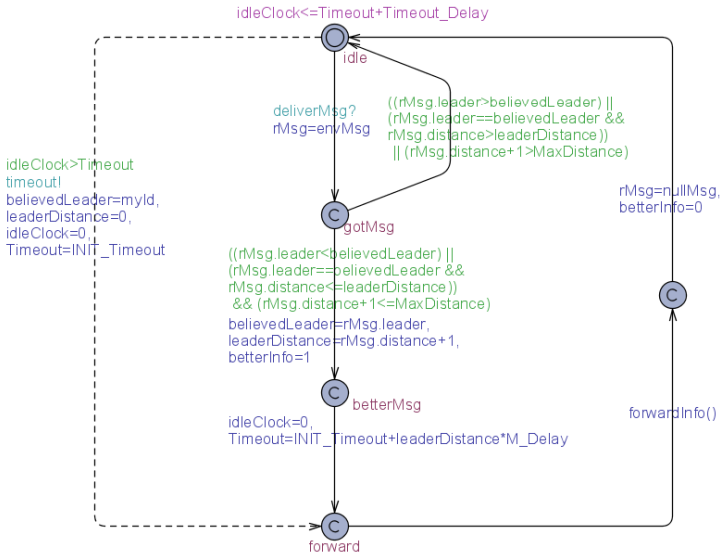
- The Leader Election Protocol [lamport05]
- To elect the node with the lowest id
- Time sensitive:
  timeout=INIT_TO+leaderDist*PropagationDelay



System Architecture



Model of a Node



Results

	TIOGA						PO-TIOGA					
	1 node as SUT			1 node + all bufs as SUT			1 node as SUT			1 node + all bufs as SUT		
(nodes, bufs)	size	time(s)	mem (KB)	size	time	mem (KB)	size	time(s)	mem (KB)	size	time	mem (KB)
(3, 3)	1739	0.16	5140	2102	0.30	6640	67	1.11	33924	31	4.96	95060
(4, 4)	72300	25.15	183252	439269	703.08	3464788	56	18.33	172224	51	216.28	805864
(5, 5)		out of mem			out of mem		81	137.34	779124		out of mem	

- Promising (but may be costly)
- Surprisingly P.O test generation scales better
  - Different algorithms for game solving
  - Finer (fully observable) vs. Coarser (partially observable) state space partitioning

Conclusions

Model-driven development

- Modelling, verification and testing are important activities
- Early design exploration & synthesis
- Testing can be formal too
- Testing verification and synthesis have much in common
- Research remains test generation for real-time, hybrid, probabilistic systems

www.uppaal.com

- **Much research for prospective students**

References

- Check online version http://www.cs.aau.dk/~bnielsen/rabat2010.pdf