Modeling, Verification and Testing of Embedded Systems

Speaker: Brian Nielsen
Centre of Embedded Software Systems
Aalborg University, DK
bnielsen@cs.aau.dk
Course Outline

1. Introduction
2. Modeling
   1. Modelling Embedded systems
   2. Introduction to timed automata (TA)
3. Verification using Upfal
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

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

Plant
Continuous

Controller Program

sensors

actuators

Model of Environment (non-deterministic/User-supplied)

Model of Tasks (user supplied/automatic?)

UPPAAL Model

inputs

outputs

Real-time Model-checking

Plant
Continuous

Controller Program

sensors

actuators

Model of Environment (non-deterministic/User-supplied)

Model of Tasks (user supplied/automatic?)

UPPAAL Model

inputs

outputs

SAT $\phi$ ??
**Real-time Controller Synthesis**

Plant (Continuous) \[\text{sensors}\] \[\text{Controller Program}\] \[\text{actuators}\]

Model of Environment (non-deterministic/User-supplied)

Partial UPPAAL Model

\[
\text{inputs} \quad \text{outputs}\]

\[\text{SAT} \land \phi \land \square !!\]

**Real-time Model-Based Testing**

Plant (Continuous) \[\text{sensors}\] \[\text{Controller Program}\] \[\text{actuators}\]

Test generation (offline or online) wrt. Design Model

UPPAAL Model

\[
\text{inputs} \quad \text{outputs}\]

\[\text{Conforms-to?}\]
Real-time Monitoring

Plant
Continuous

Controller Program

Model of Tasks
(user supplied/automatic?)

Observed trace $\sigma \in M$?

UPPAAL Model

inputs

outputs

Model of Environment
(non-deterministic/User-supplied)

sensors

actuators

UPPAAL

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

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

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

Tool Environment for
modeling, simulation,
verification, optimization &
testing of real-time systems
UPPAAL Tools www.uppaal.com

Upfaal 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

UPPAAL Team

@UPPsala
- Wang Yi
- Paul Pettersson
- John Håkansson
- Anders Hessel
- Pavel Krčal
- Leonid Mokrushin
- Shi Xiaochun

@AALborg
- Kim G Larsen
- Alexandre I
- Marius Mikucionis
- Gerd Behrmann
- Arne Skou
- Brian Nielsen
- Jacob I. Rasmussen
- Thomas Chatain

@Elsewhere
- Emmanuel Fleury, Didier Lime, Johan Bengtsson, Fredrik Larsson, Kåre J Kristoffersen, Tobias Amnelli, 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, François 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 spent on elaborate, ad-hoc testing.
- High potential for improving testing methods and tools.
- Time-to-market may be shortened considerably by verification of early designs.
- Quantitative aspects essential for ES.

Verification and Testing

Model

Code Running System

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

/* Operating system visualTEST process. Simulate a OS process for a */
/* visualTEST system. In their implementation this is the main-process*/
/* containing the visualTEST basic API. */
void OS_VS_Process(void);

/* Define completion code variable. */
// config.c arg cc;
void HandleError(unsigned char ccArg)
{
  printf("Error code %c detected, exiting application.
", ccArg);
  exit(ccArg);
}

/* In d-241 we only use the OS_Wait call. It is used to simulate a */
/* system. Its purpose is to generate events. How this is done is up to */
/* the user. */
void OS_Wait(void);

/* Ignore the parameters; just serves events from the keyboard and */
/* puts them onto the queue. When EVENT_UNDEFINED is read from the */
/* keyboard, this is returned to the calling process. */
/* EDF EVENT_STREAM events; */
/* done */
/* Wait for
void OS_Wait(void);

/* Operating system visualSTATE process. Create a OS process for a
visualSTATE system. In this implementation this is the mainloop.
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. Here this is done in up to
a few.
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. */
    while(event_queue() != -1); /* infinite loop */
}
Test versus Verification

Airbus Control Panel

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 Embedded Systems

Speaker: Brian Nielsen
Centre of Embedded Software Systems
Aalborg University, DK

bnielsen@cs.aau.dk
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?
**Real-time Modeling**

- **Plant (Continuous)**
- **Controller Program**

Model of Environment (non-deterministic/ User-supplied)

- **inputs**
- **outputs**

**UPPAAL Model**

Model of Tasks (user supplied/automatic?)

---

**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.

**Model**

- Simulink
- Scade
- UML
- SDL
- StateFlow
- UPPAAL
- Rhapsody

**Realization**

"Implemented?"
Modelling and Analysis

Software Model \textbf{A} \hspace{1cm} No! \hspace{1cm} Debugging Information

Requirement \textbf{F} \hspace{1cm} Yes, \hspace{0.5cm} Prototypes \hspace{0.5cm} Executable Code \hspace{0.5cm} Test sequences

\textbf{Tools:} UPPAAL, visualSTATE, ESTEREL, SPIN, Statemate, FormalCheck, VeriSoft, Java Pathfinder, ...

UPPAAL Tools \hspace{1cm} \url{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 customers

Thread customer1 () {
    int a,b; //local tmp copy
    a=accountA;
b=accountB;
a=a-10;b=b+10;
    accountA=a;
    accountB=b;
}

Thread customer2 () {
    int a,b;
a=accountA;
b=accountB;
a=a-20; b=b+20;
    accountA=a;
    accountB=b;
}
```

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

Home Banking

A[1] (pc1.finished and pc2.finished) imply (accountA+accountB==200)?
Home Banking

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

Thread customer1 () {
    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 customer2 () {
    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**

- **open**
- **wait**
- **signal**

**Counting Semaphore**

- **counting**
- **c = init_count**
- **c > 0**
- **c = c - 1**
- **wait**
- **signal**
Composition

*IO Automater (2-vejs synkronisering)*

![Diagram](image1)

![Diagram](image2)
Semaphore Solution?

1. A \[\{\text{mc1.finished and mc2.finished}\} \implies (\text{accountA+accountB==200})\] ✓
2. \(E < > \text{mc1.critical_section and mc2.critical_section}\) ✓
3. \(A[] \neg (\text{mc1.finished and mc2.finished}) \implies \neg \text{deadlock}\) +

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

States:
- $(location, x=v)$ where $v \in \mathbb{R}$

Transitions:
- $(off, x=0) \rightarrow (off, x=4.32)$
- $(light, x=0) \rightarrow (light, x=4.51)$
- $(light, x=0) \rightarrow (light, x=100)$
- $\tau \rightarrow (off, x=0)$
- $(off, x=0) \rightarrow (light, x=0)$
- $(light, x=0) \rightarrow (light, x=100)$
- $(light, x=0) \rightarrow (light, x=100)$
- $(light, x=0) \rightarrow (light, x=100)$
Modeling Language

- Network of TA = instances of templates
  - argument \textit{const type expression}
  - argument \textit{type & name}

- Types
  - built-in types: \textit{int}, \textit{int[min,max]}, \textit{bool}, arrays
  - \textit{typedef struct \{ ... \} name}
  - \textit{typedef built-in-type name}

- Functions
  - C-style syntax, no pointer but references OK.

- Select
  - \textit{name : type}

---

Un-timed Example: Jugs

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

Jugs Actions:
- \textit{fill}
- \textit{empty}
- \textit{pour}

Goal: obtain 1 unit.
**Jugs cont.**

- Jug levels & capacities:
  ```c
  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)
  ```c
  {
  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
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

- Validation Properties
  - Possibly: E<> P

- Safety Properties
  - Invariant: A[] P
  - Pos. Inv.: E[] P

- Liveness Properties
  - Eventually: A<> P
  - Leads-to: P \rightarrow Q

- Bounded Liveness
  - Leads to within: P \rightarrow_{st} Q
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.

```plaintext
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: $P \rightarrow Q$

- **Bounded Liveness**
  - Leads to within: $P \rightarrow_t Q$
Logical Specifications

- Validation Properties
  - Possibly: $E<> P$

- Safety Properties
  - Invariant: $A[P]
  - Pos. Inv.: $E[P]

- Liveness Properties
  - Eventually: $A<> P$
  - Leadsto: $P \rightarrow Q$

- Bounded Liveness
  - Leads to within: $P \rightarrow \tau Q$
Logical Specifications

- Validation Properties
  - Possibly: \( E<> P \)

- Safety Properties
  - Invariant: \( A[] P \)
  - Pos. Inv.: \( E[] P \)

- Liveness Properties
  - Eventually: \( A<> P \)
  - Leadsto: \( P \rightarrow Q \)

- Bounded Liveness
  - Leads to within: \( P \rightarrow_{\ell} Q \)

Gear Controller

*with MECEL AB*

Lindahl, Pettersson, Yi

Flowgraph
Gear Control (partial)

GearBox & Clutch

Neutral

Opening

Closing

Idle
Gear Controller  
with MECEL AB

Requirements

\[
\begin{align*}
\text{GearControl@Initiate} & \sim_{\leq 1000} ( ( \text{ErrStat} = 0 ) \Rightarrow \text{GearControl@GearChanged} ) \\
\text{GearControl@Initiate} & \sim_{\leq 1000} ( ( \text{ErrStat} = 0 \land \text{UseCase} = 0 ) \Rightarrow \text{GearControl@GearChanged} ) \\
\text{Clutch@ErrorClose} & \sim_{\leq 300} \text{GearControl@CCloseError} \\
\text{Clutch@ErrorOpen} & \sim_{\leq 300} \text{GearControl@COpenError} \\
\text{GearBox@ErrorIdle} & \sim_{\leq 300} \text{GearControl@GSetError} \\
\text{GearBox@ErrorNew} & \sim_{\leq 300} \text{GearControl@GNewError} \\
\text{Inv} ( \text{GearControl@CCloseError} \Rightarrow \text{Clutch@ErrorClose} ) \\
\text{Inv} ( \text{GearControl@COpenError} \Rightarrow \text{Clutch@ErrorOpen} ) \\
\text{Inv} ( \text{GearControl@GSetError} \Rightarrow \text{GearBox@ErrorIdle} ) \\
\text{Inv} ( \text{GearControl@GNewError} \Rightarrow \text{GearBox@ErrorNew} ) \\
\text{Inv} ( \text{Engine@ErrorSpeed} \Rightarrow \text{ErrStat} \neq 0 ) \\
\text{Inv} ( \text{Engine@Torque} \Rightarrow \text{Clutch@Closed} )
\end{align*}
\]

Uppaal Internals

How does it work?
Example

Example of a system with states L0 and L1, and transitions a, b, and c. The state L0 with conditions x=0, y=0, y<=2, and x<=2 transitions to L1 with condition y<=2, x>=4.

Reachable?

Zones

From infinite to finite

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

Symbolic state (set)
(n, 1 ≤ x ≤ 4, 1 ≤ y ≤ 3)

Zone: conjunction of
x·y<=n,
x<=n,
x>=n
Symbolic Transitions

Thus \((n, 1 \leq x \leq 4, 1 \leq y \leq 3) \rightarrow^a (m, 3 < x, y=0)\)

Symbolic Exploration

Reachable?
Symbolic Exploration

Symbolic Exploration

Reachable?

Reachable?

x:=0
y:=0
y<=2
x<=2
y<=2, x>=4

Delay

Left
Symbolic Exploration

Symbolic Exploration
Symbolic Exploration

Reachable?
Symbolic Exploration

\[ y := 0 \]
\[ y <= 2 \]
\[ x := 0 \]
\[ x <= 2 \]
\[ y <= 2, x >= 4 \]

Reachable?

Symbolic Exploration

\[ y := 0 \]
\[ y <= 2 \]
\[ x := 0 \]
\[ x <= 2 \]
\[ y <= 2, x >= 4 \]

Reachable?
### Difference Bound Matrices

<table>
<thead>
<tr>
<th>$x_0 - x_0 \leq 0$</th>
<th>$x_0 - x_1 \leq -2$</th>
<th>$x_0 - x_2 \leq -1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 - x_0 \leq 6$</td>
<td>$x_1 - x_1 \leq 0$</td>
<td>$x_1 - x_2 \leq 3$</td>
</tr>
<tr>
<td>$x_2 - x_0 \leq 5$</td>
<td>$x_2 - x_1 \leq 1$</td>
<td>$x_2 - x_2 \leq 0$</td>
</tr>
</tbody>
</table>

$x_i - x_j \leq c_{ij}$

### Forward Reachability Algorithm

**Init $\rightarrow$ Final?**

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

**REPEAT**
- pick \((n, Z)\) in **Waiting**
- if \((n, Z) = \text{Final}\) return true
- for all \((n, Z) \rightarrow (n', Z')\):
  - if for some \((n', Z'')\) \(Z' \subseteq Z''\) continue
  - else add \((n', Z')\) to **Waiting**

**UNTIL** \(\text{Waiting} = \phi\)
- return false
**Forward Reachability Algorithm**

Initial Passed := Ø;  
Waiting := \{(n₀,Z₀)\}

**INITIAL**

**REPEAT**

pick \((n,Z)\) in Waiting

**UNTIL** Waiting = Ø

return false
INITIAL \( \text{Passed} := \emptyset; \)
\( \text{Waiting} := \{(n_0,Z_0)\} \)

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

UNTIL \(\text{Waiting} = \emptyset\)
return false
Forward Reachability Algorithm

\[ \text{Init} \rightarrow \text{Final} \]

\[
\begin{align*}
\text{INITIAL} & \quad \text{Passed} := \emptyset; \\
& \quad \text{Waiting} := \{(n_0, Z_0)\}
\end{align*}
\]

REPEAT
pick \((n, Z)\) in Waiting
if \((n, Z) = \text{Final}\) return true
for all \((n, Z) \rightarrow (n', Z')\):
    if for some \((n', Z'')\) \(Z' \subseteq Z''\) continue
else add \((n', Z')\) to Waiting
move \((n, Z)\) to Passed
UNTIL \text{Waiting} = \emptyset
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!
The Cruise Controller

User

Controller

- engineOff, engineOn, acc, brake
- On, off, resume

SpeedControl

- enableControl, disableControl, recordSpeed

Engine

- setThrottle
- speed
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

Plant
Continuous

Controller Program

Model of Environment
(non-deterministic/User-supplied)

 inputs

outputs

Partial UPPAAL Model

sensors

actuators

Synthesis of Tasks/Scheduler (automatic)

SAT ϕ !!
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:
\[(D \times (C \times (A + B))) + ((A + B) + (C \times D))\]

using 2 processors

P1 (fast) P2 (slow)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>+2ns</td>
<td>+5ns</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>+3ns</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>+7ns</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P1: 2 3 5 6 13 ns!!
P2: 1 4
Optimal Scheduling – Time

Compute:
\[(D \cdot (C \cdot (A + B))) + ((A + B) + (C \cdot D))\]

using 2 processors

P1 (fast)  P2 (slow)

\[12 \text{ ns OPTIMAL !!}\]

Optimal Scheduling – Power

Compute:
\[(D \cdot (C \cdot (A + B))) + ((A + B) + (C \cdot D))\]

using 2 processors

\[139 \text{ nJoule !!}\]
Optimal Scheduling – Power

Compute:
\[(D \cdot (C \cdot (A + B)) + ((A + B) + (C \cdot D)))\]
using 2 processors

P1 (fast) P2 (slow)

ENERGY:

In use 1W
Idle 2W

In use 9W
Idle 3W

Energi: 132 nJoule OPTIMAL !!

Task Graph Scheduling

Optimal Static Task Scheduling

- Task \( P = \{P_1, \ldots, P_m\} \)
- Machines \( M = \{M_1, \ldots, M_n\} \)
- Duration \( \Delta : (P \times M) \rightarrow N \)

- Compute schedule with minimum completion-time!
**Task Graph Scheduling**

**Optimal Static Task Scheduling**

- Task $P = \{P_1, \ldots, P_m\}$
- Machines $M = \{M_1, \ldots, M_n\}$

### Task Information

- Task 1
- Task 2
- Task 3
- Task 4
- Task 5
- Task 6
- Task 7

### Machine Information

- $M_1$
- $M_2$

---

**Experimental Results**

<table>
<thead>
<tr>
<th>name</th>
<th>#tasks</th>
<th>#chains</th>
<th># machines</th>
<th>optimal</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>437</td>
<td>125</td>
<td>4</td>
<td>1178</td>
<td>1182</td>
</tr>
<tr>
<td>000</td>
<td>452</td>
<td>43</td>
<td>20</td>
<td>537</td>
<td>537</td>
</tr>
<tr>
<td>018</td>
<td>730</td>
<td>175</td>
<td>10</td>
<td>700</td>
<td>704</td>
</tr>
<tr>
<td>074</td>
<td>1007</td>
<td>66</td>
<td>12</td>
<td>891</td>
<td>894</td>
</tr>
<tr>
<td>021</td>
<td>1145</td>
<td>88</td>
<td>20</td>
<td>605</td>
<td>612</td>
</tr>
<tr>
<td>228</td>
<td>1187</td>
<td>293</td>
<td>8</td>
<td>1570</td>
<td>1574</td>
</tr>
<tr>
<td>071</td>
<td>1193</td>
<td>124</td>
<td>20</td>
<td>629</td>
<td>634</td>
</tr>
<tr>
<td>271</td>
<td>1348</td>
<td>127</td>
<td>12</td>
<td>1163</td>
<td>1164</td>
</tr>
<tr>
<td>237</td>
<td>1566</td>
<td>152</td>
<td>12</td>
<td>1340</td>
<td>1342</td>
</tr>
<tr>
<td>231</td>
<td>1664</td>
<td>101</td>
<td>16</td>
<td>t.o.</td>
<td>1137</td>
</tr>
<tr>
<td>235</td>
<td>1782</td>
<td>218</td>
<td>16</td>
<td>t.o.</td>
<td>1150</td>
</tr>
<tr>
<td>252</td>
<td>1980</td>
<td>207</td>
<td>19</td>
<td>1118</td>
<td>1121</td>
</tr>
<tr>
<td>294</td>
<td>2044</td>
<td>141</td>
<td>17</td>
<td>1257</td>
<td>1261</td>
</tr>
<tr>
<td>295</td>
<td>2168</td>
<td>965</td>
<td>18</td>
<td>1318</td>
<td>1322</td>
</tr>
<tr>
<td>292</td>
<td>2333</td>
<td>318</td>
<td>3</td>
<td>8009</td>
<td>8009</td>
</tr>
<tr>
<td>298</td>
<td>2599</td>
<td>303</td>
<td>10</td>
<td>2471</td>
<td>2473</td>
</tr>
</tbody>
</table>

**Symbolic A**\*\, Brand-&-Bound
60 sec

Abbeddaïm, Kerbaa, Maler
**Linearly Priced Timed Automata**

- Timed Automata + costs on transitions and locations
- Cost of performing transition: transition cost
- Cost of performing delay $\varepsilon : (\varepsilon \times$ location cost)
- Trace:
  $\varepsilon = (a,x=y=0) \rightarrow (b,x=y=0) \rightarrow (b,x=y=2) \rightarrow (a,x=0,y=2)$
- Cost of Execution Trace:
  - Sum of costs: $4 + 5 + 0 = 9$

---

**Optimal Task Graph Scheduling**

*Power-Optimality*

- **Energy-rates:**
  $C : M \rightarrow N$
- Compute schedule with minimum completion-cost!
Task Graph Scheduling
Optimal Static Task Scheduling

- Task \( P = \{P_1, \ldots, P_m\} \)
- Machines \( M = \{M_1, \ldots, M_n\} \)
- Duration \( \Delta : (P \times M) \rightarrow N \)
- \(< : p.o. on P (pred.) \)

- A task can be executed only if all predecessors have completed.
- Each task takes a specific duration.
- Task cannot be preempted.

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 $\phi$

**FIND** strategy $s_C$ such that $s_C || S$ sat $\phi$

A Two-Player Game
**Timed Game Automata**

[Maler, Pnueli, Sifakis'95].

The controller continuously observes all delays & moves

**Move:**
- uncontrollable edge: $c$
- delay: $\lambda$

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

**Memoryless strategy:**

$F : \text{State} \rightarrow E_c \cup \lambda$

**Reachability Games:** Reach Goal

**Safety Games:** Avoid loose

---

**Timed Games**

*a winning strategy:*

- **L0:**
  - $x<1 : \lambda$
  - $x=1 : c$

- **L1:**
  - $x<2 : \lambda$
  - $x\geq2 : c$

- **L2:**
  - $x\leq1 : c$

- **L3:**
  - $x<1 : \lambda$
  - $x=1 : c$
Timed Game Solver

UPPAAL TIGA
UPPAAL for Timed Games
Main Page | Download | Contact us

Welcome!

UPPAAL TIGA (Fig. 1) is an extension of UPPAAL (Fig. 2) and it implements the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and Safety properties. Though timed games for long have been known to be undecidable there has until now been a lack of efficient and tried on-the-fly algorithms for their analysis.

The algorithm we propose is a symbolic extension of the on-the-fly algorithms supported by UPPAAL (2000) for linear-time model-checking of finite-state systems. Being on-the-fly, this symbolic algorithm may terminate long before having explored the entire state-space. Also the individual steps of the algorithm are carried out efficiently by the use of specialized solvers as the algorithm runs.

Controller Synthesis:
Hydac Case

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.

Latest News
Version 0.9 released.
22 Nov 2001

Version 0.9 is released today. It fixed few major bugs. Wrong error messages were given for some involving axis and the simulation handle was prevented from being created in the GUI.

Known issues added.
20 Nov 2001

A known issue section has been finalized today.

Version 0.8 repackaged.
20 Nov 2000

The GUI has been replaced by a new GUI providing more features.

Quasimodo
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
  \[ \int_{t=0}^{T} v(t) dt / T \]
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)

bool Noise(int s){
    // s is the duration of consumption (in t.u.)
    return (V-s<(Vmin+1)*D | V+s>(Vmax-1)*D);}

Checks whether V under noise gets outside [Vmin+0.1,Vmax-0.1]
**Pump** (controllable)

```c
void update_val()
{
    int V_curr = V,
    time++;
    V += V_rate;
    V_acc += V + V_preds;
}
```

**Global Approach**

- Find some interval $I_1 = [V_1, V_2] \in [4.9, 25.1]$ s.t.
- $I_1$ is $m$-stable i.e. from any $V_0$ in $I_1$ there is strategy st.
- whatever fluctuation
- volume is always within $[5, 25]$ and at the end within $[10, 15]$.
- $I_1$ is optimal among all $m$-stable intervals.

**Queries** (min. $K$)
- control: $A <\neg (time = 20 \land not BAD 
  \land V \in I_2 \land V_{acc} \leq K)$
- $I_1$ is optimal among all $m$-stable intervals.
Results

\[ I_1 = [5.1, 10] \]

\[ D=1, \ m=0.4: \text{ Optimal stable interval } I_1 = [5.1, 10] \]
## Results

**Guaranteed Correctness Robustness**

with **45% improvement** in performance (BangBang) in **33% improvement** (“Hydac Smart”)

<table>
<thead>
<tr>
<th></th>
<th>1513</th>
<th>7.64</th>
<th>8.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>G2M3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td>7.5</td>
<td>7.95</td>
</tr>
<tr>
<td>G2M2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1489</td>
<td></td>
<td>7.44</td>
<td>7.95</td>
</tr>
<tr>
<td>G2M1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Modeling, Verification and Testing of Embedded Systems

Speaker: Brian Nielsen
Centre of Embedded Software Systems
Aalborg University, DK

bnielsen@cs.aau.dk
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
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

Test Cases

Test Data

Output

Correct result?

Software under Test

Oracle
Types of Testing

**Level**
- system
- integration
- unit

**Accessibility**
- white box
- black box

**Aspect**
- functionality
- reliability
- usability
- efficiency

**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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal (word)</td>
<td>Informal word</td>
<td>Formal Test model</td>
</tr>
<tr>
<td>Manual Informal (word, excell)</td>
<td>Manual Informal Word/excell</td>
<td>Automated (implicit)</td>
</tr>
<tr>
<td>Informal (word, excell)</td>
<td>Manually written Scripts</td>
<td>Automated (scripts or implicit)</td>
</tr>
<tr>
<td>Manual execution &amp; Inspection</td>
<td>Automated Execution</td>
<td>Automated Execution &amp; Evaluation</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equilateral Triangle</td>
<td>Three equal sides. Three equal angles, always $60^\circ$</td>
</tr>
<tr>
<td>Isosceles Triangle</td>
<td>Two equal sides. Two equal angles</td>
</tr>
<tr>
<td>Scalene Triangle</td>
<td>No equal sides. No equal angles</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acute Triangle</td>
<td>All angles are less than $90^\circ$</td>
</tr>
<tr>
<td>Right Triangle</td>
<td>Has a right angle ($90^\circ$)</td>
</tr>
<tr>
<td>Obtuse Triangle</td>
<td>Has an angle more than $90^\circ$</td>
</tr>
</tbody>
</table>

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

![Timed Coffee Machine diagram](image)

Conformance Relation

- Timed Automata with Timed-LTS semantics
- **Input** actions (?) are controlled by the environment
- **Output** actions (!) are controlled by the implementation
- Implementations are *input enabled*

**Testing hypothesis:** IUT can be modeled by some (unknown) TA
Does $I_n$ conform-to $S_1$?

Timed Conformance

Derived from Tretman’s IOCO

Let $I, S$ be timed I/O LTS, $P$ a set of states

$TTr(P)$: the set of timed traces from $P$

eg.: $\sigma = \text{coin?} .5 .\text{req?} .2 .\text{thinCoffee}! .9 .\text{coin}$

$\text{Out}(P \text{ after } \sigma)$ = possible outputs and delays after $\sigma$

eg. out ($\{12,x=1\}$): $\{\text{thinCoffee}, 0...2\}$

$I \rt-ioco S = \text{def}$

$\forall \sigma \in TTr(S): \text{Out}(I \text{ after } \sigma) \subseteq \text{Out}(S \text{ after } \sigma)$

$TTr(I) \subseteq TTr(S)$ 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 $S_1$?

$S_1$

- coin?
- give? $x=0$
- $x=3$
- coffee

$I_1$

- coin?
- give? $x=0$
- $x=3$
- coffee

$\sigma = \text{coin.give.10}$
$\sigma \in \text{TTr}(I_1)$, $\sigma \notin \text{TTr}(S_1)$

$out(I_1 \text{ after coin.give.3}) = \{0...\}$
$\subset$
$out(S_1 \text{ after coin.give.3}) = \{\text{coffee,0...2}\}$

Does $I_n$ conform-to $S_1$?

$S_1$

- coin?
- give? $x=0$
- $x=3$
- coffee

$I_3$

- coin?
- give? $x=0$
- $x=3$
- coffee

$\sigma = \text{coin.give.7.coffee}$
$\sigma \in \text{TTr}(I_3)$, $\sigma \notin \text{TTr}(S_1)$

$out(I_3 \text{ after coin.give.7}) = \{\text{coffee,0}\}$
$\subset$
$out(S_1 \text{ after coin.give.7}) = \{\}$

$I_4$

- coin?
- give? $x=0$
- $x=3$
- coffee

$\sigma = \text{coin.give.1.coffee}$
$\sigma \in \text{TTr}(I_4)$, $\sigma \notin \text{TTr}(S_1)$

$out(I_4 \text{ after coin.give.1}) = \{\text{coffee,0...4}\}$
$\subset$
$out(S_1 \text{ after coin.give.1}) = \{0...4\}$
**Does \( I_n \) conform-to \( S_1 \)?**

\[ \sigma = \text{coin.give.5.tea} \]
\[ \sigma \in \text{Tr}(17), \sigma \notin \text{Tr}(S1) \]

\[ \text{out}(17 \text{ after coin.give.5}) = \{\text{tea, coffee, 0}\} \]
\[ \sigma = \text{token.5.vodka} \]
\[ \sigma \in \text{Tr}(I8), \sigma \notin \text{Tr}(S1) \]

But \( \sigma \) was not specified

**Sample Cooling Controller**

IUT-model

- 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)

**Conformance relation**

Relativized real-time io-conformance

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

\[
\text{I rt-ioco}_{E} S \text{ iff } \forall \sigma \in \text{TTr}(E): \text{Out}((E,I) \text{ after } \sigma) \subseteq \text{Out}((E,S) \text{ after } \sigma)
\]

\[
\text{I rt-ioco}_{E} S \text{ iff } \text{TTr}(I) \cap \text{TTr}(E) \subseteq \text{TTr}(S) \cap \text{TTr}(E) \text{ // input enabled}
\]

- 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 \text{rt}-ioco_E S

1. Given new (weaker) system specification S’
   \[
   \text{If } S \sqsubseteq S' \text{ then } I \text{ rt}-ioco_E S'
   \]

2. Given new (stronger) environment specification E’
   \[
   \text{If } E' \sqsubseteq E \text{ then } I \text{ rt}-ioco_{E'} S
   \]

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 $\delta$

$$\text{ioco } s \overset{\text{def}}{=} \forall \sigma \in \text{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$$

$$\text{p after } \sigma = \{ p' \mid p \xrightarrow{\sigma} p' \}$$

$$\text{p } \xrightarrow{\delta} \text{p} \text{ iff } \forall o! \in L_U \cup \{\tau\} : p \xrightarrow{o!} p \in P$$

$$\text{out}(P) = \{ o! \in L_U \mid p \xrightarrow{o!} p \in P \}$$

$$\cup \{ \delta \mid p \xrightarrow{\delta} p, p \in P \}$$

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

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? · 2h · grasp! · 220 · release! · touch? · PASS`
- `1000 · grasp! · 517 · starthold? · 100 · release! · endhold? · PASS`

INFINITELY MANY SEQUENCES!!!!!!

DEMO: Touch-sensitive Light-Controller
Overview of Techniques

<table>
<thead>
<tr>
<th>Model</th>
<th>Restrictions</th>
<th>Technique</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Douta&quot;</td>
<td>Completely controllable</td>
<td>Counter Examples (Guarantees coverage/cost)</td>
<td>Offline</td>
</tr>
<tr>
<td>Observable TA</td>
<td>Timeing uncertainty+ Multiple Outputs</td>
<td>Game (definitely and possibly winning)</td>
<td>Offline (+ online)</td>
</tr>
<tr>
<td>Partially Observable TA</td>
<td>Observation Predicates</td>
<td>Game</td>
<td>Offline</td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted nondeterminism</td>
<td>Counter Examples (Preset-input sequences only)</td>
<td>Offline</td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted nondeterminism</td>
<td>Stat-set tracking</td>
<td>Online</td>
</tr>
</tbody>
</table>

Test Generation using Verification

myGearControl.xml

System model

Test purpose Property

\( E<> Gear.Gear5 \)

Trace (witness)

Some Random Shortest Fastest

testGear5.trc

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:

\[ E<> L>=10 \]

---

**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 $l_0, ..., l_n$
  - Add an auxiliary variable $l_i$ for each location
  - Label each ingoing edge to location $i$ $l_i := \text{true}$
  - Mark initial visited $l_0 := \text{true}$
- Check: $E<> ( l_0 = \text{true} \land ... \land l_n = \text{true} )$
**Edge Coverage**

- Test sequence traversing all edges
- Encoding:
  - Enumerate edges $e_0, \ldots, e_n$
  - Add auxiliary variable $e[i]$ for each edge
  - Label each edge $e[i] := 1$
- Check:
  $$E<> ( e[0]=1 \land \ldots \land e[n]=1 )$$

**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 $\sigma = e_{i_0}i_0, \ldots, e_{i_1}i_1, \text{reset } e_{i_2}i_2, \ldots, e_{i_n}i_n, \text{reset } e_{i_2}$
- 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 $\varepsilon$ : ( $\varepsilon$ x location cost )
- Trace:
  $\langle a,x=0 \rangle \rightarrow (b,x=0) \rightarrow (b,x=0,y=0) \rightarrow (a,x=0,y=2)$
- Cost of Execution Trace:
  - Sum of costs: $4 + 5 + 0 = 9$

Fastest Edge Coverage

Time = 12600 ms
Power-Optimal Edge Coverage

Cost = ???

Offline Testing of Non-Deterministic TA

1. **Compute “preset” timed input-sequence** \( \sigma_i \)
2. **Blindly Execute input sequence and log i/o sequence** \( \sigma_{io} \)
3. **Post mortem verdict evaluation by model-checking**
   
   **trace inclusion** \( \sigma_{io} \subseteq T(M) \)

**FAIL:** \( \sigma_{io} \not\subseteq T(M) \)

**PASS:** \( \text{INCONC } \sigma_{io} \subseteq T(M) \) and goal-state possible reached

**INCONC:** \( \sigma_{io} \subseteq T(M) \) but goal state not reachable

Can be answered using Uppaal reachability analysis of \( \sigma_{io} \parallel M \)
Offline Testing of Non-Deterministic TA

1. Compute “preset” timed input-sequence $\sigma_i$
2. Blindly Execute input sequence and log i/o sequence $\sigma_{io}$
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 possibly reached
INCONC: $\sigma_{io} \in TTr(M)$ but goal state not reachable
Can be answered using Uppaal reachability analysis of $\sigma_{io} \mid \mid M$

**DISADVANTAGE:**
- Test not adaptive to outputs or timing of SUT
- Goal not necessarily observed (improbable?)

---

Testing
On-Line
Overview of Techniques

<table>
<thead>
<tr>
<th>Model</th>
<th>Restrictions</th>
<th>Technique</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Douta&quot;</td>
<td>Completely controllable</td>
<td>Counter Examples</td>
<td>Offline</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Guarantees coverage/cost)</td>
<td></td>
</tr>
<tr>
<td>Observable TA</td>
<td>Timeing uncertainty+</td>
<td>Game</td>
<td>Offline (+ online)</td>
</tr>
<tr>
<td></td>
<td>Multiple Outputs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partially Observable TA</td>
<td>Observation Predicates</td>
<td>Game</td>
<td>Offline</td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted non-determinism</td>
<td>Counter Examples</td>
<td>Offline</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Preset-input sequences only)</td>
<td></td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted non-determinism</td>
<td>Stat-set tracking</td>
<td>Online</td>
</tr>
</tbody>
</table>

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

- $L_0$
  - $T \leq 10$
  - Coin?
  - $T \geq 5$
  - Coin?
  - $L_2$
  - $L_3$

Between 5 and 10 coin leads to $L_2$ or $L_3$

Timing Uncertainty

- $L_0$
  - $T \leq 10$
  - $L_1$

LightLevel must be adjusted between 5 and 10

Internal actions (+ timing)

- $L_0$
  - $T \geq 5$
  - $T \leq 10$
  - $L_1$

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

```
  task1
  \rightarrow
  \text{compressorOn}
  \rightarrow
  \text{displayIndication}
```

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 $\sigma = \epsilon_0, i_0, \epsilon_1, o_1, \epsilon_2, i_2, o_2, \ldots$ [Tripakis] Failure Diagnosis

- $Z = M \text{ after } (\epsilon_0, i_0, \epsilon_1, o_1, \epsilon_2, i_2, o_2)$

- If $Z = \emptyset$ the IUT has made a computation not in model: **FAIL**
- $i$ is a relevant input in Env iff $i \in EnvOutput(Z)$

Model states

$Z \rightarrow \sigma \rightarrow S_0 \rightarrow i, o, \epsilon$
Algorithm TestGenExe \((S, E, IUT, T)\) returns \{pass, fail\}

\[ Z := \{(s_0, e_0)\}. \]

while \(Z \neq \emptyset \) \& \#iterations \leq T do either randomly:

1. // offer an input
   
   \[ \text{if } \text{EnvOutput}(Z) \neq \emptyset \]
   
   \[ \text{randomly choose } i \in \text{EnvOutput}(Z) \]
   
   \[ \text{send } i \text{ to IUT} \]
   
   \[ Z := Z \text{ After } i \]

2. // wait \(d\) for an output
   
   \[ \text{randomly choose } d \in \text{Delays}(Z) \]
   
   \[ \text{wait } \text{for } d \text{ time units or output } o \text{ at } d' \leq d \]
   
   \[ \text{if } o \text{ occurred } \text{then} \]
   
   \[ Z := Z \text{ After } o' \]
   
   \[ Z := Z \text{ After } o \] \( \Rightarrow \text{may become } \emptyset \)
   
   \[ \text{else} \]
   
   \[ Z := Z \text{ After } d \] \( \Rightarrow \text{no output within } d \text{ delay} \)

3. // restart:
   
   \[ Z := \{(s_0, e_0)\}, \text{reset } IUT \text{ //reset and restart} \]
   
   if \(Z = \emptyset\) then return fail else return pass

(Sound Complete \((as \; T \rightarrow \infty)\))

(Under some technical assumptions)
State-set computation

- Compute all potential states the model can occupy after the timed trace $\varepsilon_{0, i_0, o_0}, \varepsilon_{1, i_1, o_1}, \varepsilon_{2, i_2, o_2}, \ldots$
- Let $Z$ be a set of states

$Z$ after $a$:
- possible states after $a$ (and $\tau^*$)

$Z$ after $\varepsilon$:
- possible states after $\tau^*$ and $\varepsilon_i$, totaling a delay of $\varepsilon$

Real-time Online

Specification TA-network

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

Graphical User Interface (Java)
- editor
- simulator
- verifier

Simulator API

Uppaal Engine Server (C++)
- Parsing
- Communication
- Control

Zones & Reachability, Etc

State-set explorer
Online Test Generation

System Under Test

**Our Framework**

*UppAal Timed Automata* Network: Env || IUT

Complete and sound algorithm
Efficient symbolic reachability algorithms

**UppAal-TRON:** Testing Real-Time Systems Online
**DEMO:**
**Touch-sensitive Light-Controller**

![Diagram of light controller](image)

**On-line Testing**

*Light Controller*

![Diagram of on-line testing setup](image)
Mutants

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

Sensor Input
- air temperature sensor

Output Relays
- compressor relay
- defrost relay
- alarm relay
- defrost temperature sensor
- (door open sensor)

Keypad Input
- 2 buttons (~40 user settable parameters)

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

Industrial Application
Danfoss Electronic Cooling Controller

18 timed automata components
14 clocks, 14 integers
Example Test Run

Outcome

4 instances of discrepancy between model and actual behavior, also involving timing errors.

Offline Testing of Uncontrollable Timed Systems
Overview of Techniques

<table>
<thead>
<tr>
<th>Model</th>
<th>Restrictions</th>
<th>Technique</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Douta&quot;</td>
<td>Completely controllable</td>
<td>Counter Examples (Guarantees coverage/cost)</td>
<td>Offline</td>
</tr>
<tr>
<td>Observable TA</td>
<td>Timing uncertainty+ Multiple Outputs</td>
<td>Game (definitely and possibly winning)</td>
<td>Offline (+ online)</td>
</tr>
<tr>
<td>Partially Observable TA</td>
<td>Observation Predicates</td>
<td>Game</td>
<td>Offline</td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted non-determinism</td>
<td>Counter Examples (Preset-input sequences only)</td>
<td>Offline</td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted non-determinism</td>
<td>Stat-set tracking</td>
<td>Online</td>
</tr>
</tbody>
</table>

"Controllable" Timed I/O Automata

- Inputs (?) are controllable
- Outputs (!) are uncontrollable
- Deterministic
- Isolated outputs
- Output-urgent

- 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

Timing uncertainty of outputs
multiple enabled outputs

Timed Game Automata

[Malen, Pnueli, Sifakis’95].

The controller continuously observes all delays & moves

**Move:**
- controllable edge: $c$
- delay: $\lambda$

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

**Memoryless strategy:**
- $F : \text{State} \rightarrow E \cup \lambda$

**Reachability Games:** Reach Goal
**Safety Games:** Avoid loose
**Testing as Playing Games**

- **the game player** (the tester)
- **the game opponent** (the IUT)
- Stimuli reactions

- Controlled only by the tester
- Controlled only by the System Under Test

**Game Strategy as Test Case**

*Given an ACTL reachability test purpose $\varphi$:*

Generate a winning strategy for $\varphi$ 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

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

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?

- $E<> (control: A<> Bright)$
- $<<c,u>> (<<c>> \Diamond Bright)$
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

- 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 $F_c(s)$ either
       - offer a random enabled inputs to IUT or
       - delay random
    2. If $s$ is winning, then deterministically according to $F_w(s)$
       - offer input to IUT or
       - delay
    3. If $s$ is a goal-state, declare “PASS”.
    4. If $s$ is loosing, declare “INCONC”
- Until verdict, or max test duration elapses

Online execution of Testing Games

Cooperative or Definitely (Winning) Strategy
### Overview of Techniques

<table>
<thead>
<tr>
<th>Model</th>
<th>Restrictions</th>
<th>Technique</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Douta&quot;</td>
<td>Completely controllable</td>
<td>Counter Examples</td>
<td>Offline</td>
</tr>
<tr>
<td></td>
<td>(Guarantees coverage/cost)</td>
<td>Game</td>
<td>Online (+ online)</td>
</tr>
<tr>
<td>Observable TA</td>
<td>Timeing uncertainty+</td>
<td>Game</td>
<td>Offline (+ online)</td>
</tr>
<tr>
<td></td>
<td>Multiple Outputs</td>
<td>(definitely and possibly winning)</td>
<td></td>
</tr>
<tr>
<td>Partially Observable TA</td>
<td>Observation Predicates</td>
<td>Game</td>
<td>Offline</td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted non-determinism</td>
<td>Counter Examples</td>
<td>Offline</td>
</tr>
<tr>
<td></td>
<td>(Preset-input sequences only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timed Automata</td>
<td>Unrestricted non-determinism</td>
<td>Stat-set tracking</td>
<td>Online</td>
</tr>
</tbody>
</table>

**However,**

- Full Observability is not always feasible
- We may have:
  - **Inaccurate** measurements of SUT
    - limited number of IUT probings or instrumentations
    - 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**

**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 $y \in [0, 1)$.

**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

\[
\{ \langle \text{Off}, \text{true} \rangle, \\
\langle \text{Dim1}, \text{Dim2}, \text{true} \rangle, \\
\langle \text{Bright}, \text{true} \rangle, \\
\langle L, 0 <= y < 1 \rangle \}
\]

test purpose

control: A<> Bright

Playing OBSI Strategy

<table>
<thead>
<tr>
<th>Node</th>
<th>TA symbolic states</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{off, (0, x&lt;1, y-x&lt;0)}</td>
<td>{off, y-y1}</td>
</tr>
<tr>
<td></td>
<td>{off, (x=1, y-x&lt;0)}</td>
<td>{off}</td>
</tr>
<tr>
<td></td>
<td>{off, (1, x&lt;y&lt;1)}</td>
<td>{off, y-y1}</td>
</tr>
<tr>
<td></td>
<td>{off, (x=2, y-x&lt;1)}</td>
<td>{off}</td>
</tr>
<tr>
<td></td>
<td>{(1), (0, x&lt;2, y-x&lt;1)}</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>{(1), (0, x&lt;3, y-x&lt;1)}</td>
<td>{Dim1, Dim2}</td>
</tr>
<tr>
<td></td>
<td>{(2), (0&lt;y&lt;2, x&lt;y&lt;3)}</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>{(2), (1&lt;y&lt;2, 2&lt;y&lt;3)}</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>{}, (0&lt;y&lt;2, x&lt;3) }</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>{(0&lt;y&lt;2, x&lt;y&lt;3), y-y1}</td>
<td>{}</td>
</tr>
<tr>
<td></td>
<td>{(1&lt;y&lt;2, 2&lt;y&lt;3), y-y1}</td>
<td>{}</td>
</tr>
</tbody>
</table>

Observables

- \{Off\}(Off), \{On\}(On), \{y<y1\}(y<y1), Dim1Dim2(Dim1, Dim2), \{Bright\}(Bright)

stuttering steps

\[
\{(\text{Off}, \text{true})\}, \\
\{(\text{Dim1}, \text{Dim2}), \text{true}\}, \\
\{(\text{Bright}), \text{true}\}, \\
\{(L, 0 <= y < 1)\}
\]
Testing 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”.

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

Infor

Testing 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”.

**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:
  \[ \text{timeout} = \text{INIT}_\text{TO} + \text{leaderDist} \times \text{PropagationDelay} \]

### System Architecture

![System Architecture Diagram]
Results

- 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

- Much research for prospective students

References

- Check online version