Introduction to UPPAAL

Gerd Behrmann

Aalborg University

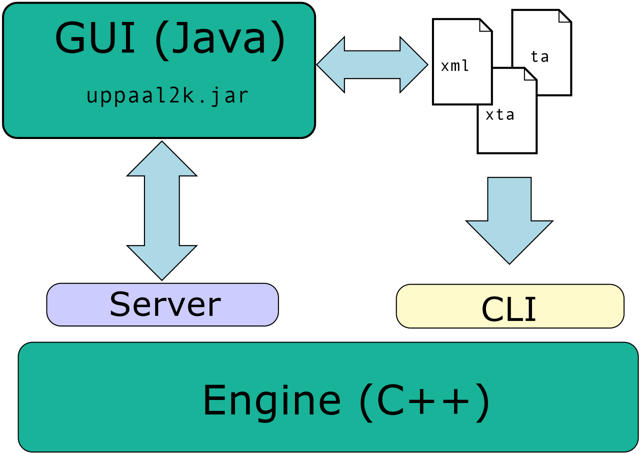
April 13, 2005

Outline

1. A short look at UPPAAL
   - Demo
   - Architecture
2. Syntax of UPPAAL
   - Declarations
   - Expressions
   - Locations and synchronisation
   - Properties
3. Train Gate Example
4. Verification Options
   - How UPPAAL works
   - State space reduction techniques
   - Reusing the state space
   - State space representation techniques.

Demo

UPPAAL's Architecture


GUI (Java)

uppaal2k.jar

xml

ta

xta

Server

CLI

Engine (C++)

Declarations

Clocks

```
clock x1, x2,...,xn;
```

Bounded Integer Variables

```
int[0,5] i1, i2,...  in;
```

Default range is -32767;32768.

Constants

```
const int delay = 5, a = 0;
```

Arrays

```
int x[4] = { 1, 4, 7, 2 };
```

Declarations

New in version 3.5

Booleans

```
bool b;
```

Records

```
struct { int a; int b; } a = { 1, 2 };
```

Type declarations

```
typedef struct { int a; int b; } A;
```

Expressions

```
Expression
  ::= ID
  |   NAT | 'true' | 'false'
  |   Expression '[' Expression ']'
  |   '(' Expression ')'
  |   Expression '++' | '++' Expression
  |   Expression '--' | '--' Expression
  |   Expression AssignOp Expression
  |   UnaryOp Expression
  |   Expression BinOp Expression
  |   Expression '?' Expression ':' Expression
  |   ID '.' ID
  |   ID '(' [ Expression ( ',' Expression )* ] ')'
```

Operators

Unary

```
’-’ | ’+’ | ’!’ | ’not’
```

Binary

```
’<’    | ’<=’ | ’==’ | ’!=’ | ’>=’ | ’>’
’+’    | ’-’  | ’*’  | ’/’  | ’%’  | ’&’
’|’    | ’^’  | ’<<’ | ’>>’ | ’&&’ | ’||’
’and’  | ’or’ | ’imply’
```

Assignment

```
 ’=’ | ’+=’ | ’-=’ | ’*=’ | ’/=’ | ’%=’
’|=’ | ’&=’ | ’^=’ | ’<<=’ | ’>>=’
```

Guards

Any expression satisfying the following conditions is a guard:

- It is side effect free, type correct and evaluates to a boolean.
- Only clock variables, integer variables and constants are referenced (or arrays of these types).
- Clocks and differences between clocks are only compared to integer expressions (no inequality).
- Guards over clocks are essentially conjunctions (*i.e.* disjunctions are only allowed over integer conditions).

Assignments

Any expression satisfying the following conditions is an assignment:

- It has a side effect and is type correct.
- Only clock variables, integer variables and constants are referenced (or arrays of these types).
- Only integers are assigned to clocks.

Invariants

Any expression satisfying the following conditions is an invariant:

- It is side effect free and is type correct.
- Only clock variables, integer variables and constants are referenced (or arrays of these types).
- It forms a conjunction of conditions on the form x `<` e or x `<=` e, where x is a clock reference and e evaluates to an integer.

Functions

New in version 3.5

- User defined functions can be declared globally or locally.
- An extended subset of C.
- Supports `while`, `for`, `do while`, `if`, `return`.
- Tests on clock variables are not allowed.
- Reset of clocks are allowed.
- Always evaluated atomically:
  - No interleaving with other processes.
  - If your function does not return, neither does UPPAAL.
- Still experimental.

```
int sum(int a, int b)
{
    return a + b;
}
```

Binary Synchronisation

Channels can be declared like:

```
chan a, b, c[3];
```

If a is channel, then:

- a! is an emission
- a? is a reception

Two edges in different processes can synchronise if one is emitting and the other is receiving on the same channel.

Broadcast Synchronization

Broadcast channels can be de declared like:

```
broadcast chan a, b, c[2];
```

If a is a broadcast channel, then:

- a! is an emission of a broadcast
- a? is a reception of a broadcast

A set of edges in different processes can synchronise if one is emitting and the others are receiving on the same broadcast channel. A process can always emit on a broadcast channel.

Urgency

Definition (Urgent State)

A state is urgent if either

- a process is in an urgent location, or
- an action transition on an urgent channel can be taken.

Definition (Semantics)

An urgent state has no delay transitions.

Urgency

Urgent channels

```
urgent chan a,b,c[3];
```

Urgent locations

- Right click location and mark it urgent.
- Equivalent to having an invariant $x \leq 0$ and reseting x before entering the location.

Committed Locations

Definition (Committed Process)

A process is committed if it is in a committed location.

Definition (Committed State)

A state is committed if any of the processes is committed.

Definition (Semantics)

- A committed state cannot delay.
- A committed state only has action transitions involving at least one committed processes.

Main purpose of committed locations is to create atomic sequences of transitions. Committed locations reduce the state space considerably by eliminating interleaving.

Templates

- Templates can be instantiated to form processes.
- Templates are parameterised.
- Call-by-value is the default (except for arrays).
- Call-by-reference is used if identifier is prefixed with &.

Example of parameter declaration of a template A:

```
process A(int &v, const int min, const int max)
```

Example of instantiation:

```
P = A(i, 1, 5);
Q = A(j, 0, 4);
```

Example of system declaration:

```
system P, Q;
```

Syntax of Properties

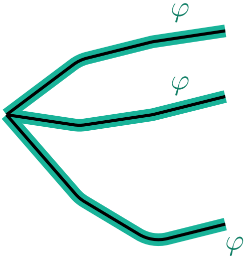
```
A[] Expression
E<> Expression
A<> Expression
E[] Expression
Expression --> Expression
A[] not deadlock
```

The expressions must be type safe, side effect free, and evaluate to a boolean. Only references to integers variables, constants, clocks, and locations are allowed (and arrays of these).

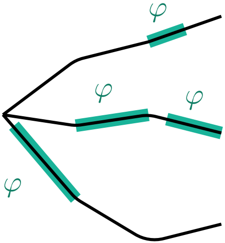
Operators `A[]` and `A<>`

For all paths

`A[]` φ



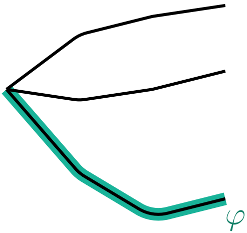
`A<>` φ



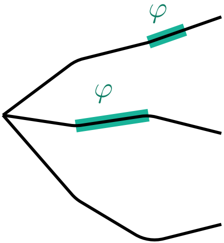
Operators E[] and E<>

There is a path

E[]φ



E<>φ



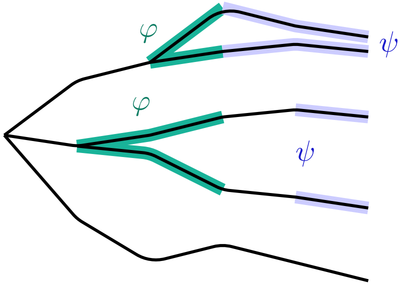
Remark

$\neg$(A[]φ)=E<>($\neg\varphi$) and $\neg$(E[]φ)=A<>($\neg\varphi$)

Operator -->

Leads to (response)

$$\varphi \texttt{ --> } \psi \overset{def}{\Longleftrightarrow} \texttt{A[]}(\varphi \Rightarrow \texttt{A<>}\psi)$$



State Property: `deadlock`

A deadlock is a state in which no action transition will ever be enabled again.

In other words $(l, u) \models \texttt{deadlock}$ iff:

$$\forall d \geq 0, a \in \mathcal{A}ct : (l, u + d) \stackrel{a}{\not\to}$$

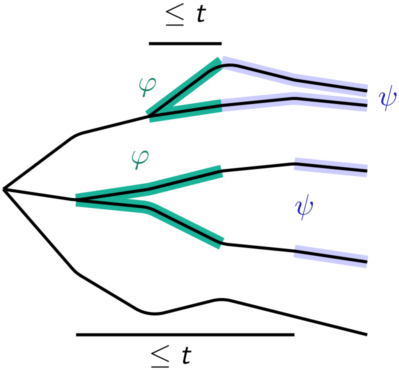
Checking for absence of deadlocks:

```
A[] not deadlock
```

Bounded Liveness

Whenever φ becomes true, then ψ becomes true within t.

$$\varphi\texttt{-->}_{\leq t}\psi$$

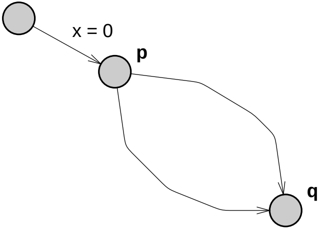


Bounded Liveness

Reduction to unbounded liveness

We can reduce `p -->`$_{\leq t}$ `q` to an unbounded liveness property:

- Add a clock x and reset it whenever p becomes true.
- Check `p --> (q and x <= t)`.



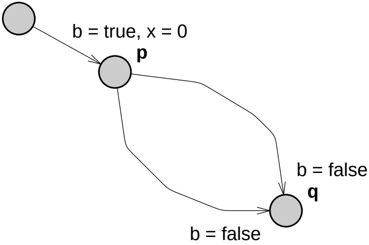
Care must be taken that x is not reset several times before q becomes true.

Bounded Liveness

Reduction to reachability by decoration

We can reduce `p -->`$_{\leq t}$ `q` to a reachability property:

- Add a clock x and reset it whenever p becomes true.
- Add a boolean b, set it to true when p starts to hold and to false when p ceases to hold.
- Check `A[] (b implies x <= t)`.

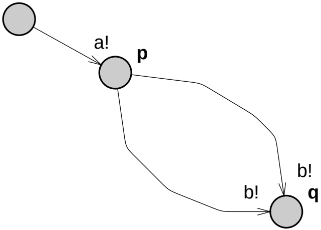


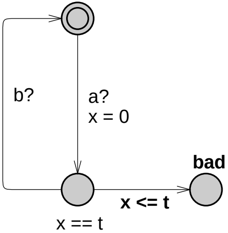
Bounded Liveness

Reduction to reachability with test automaton

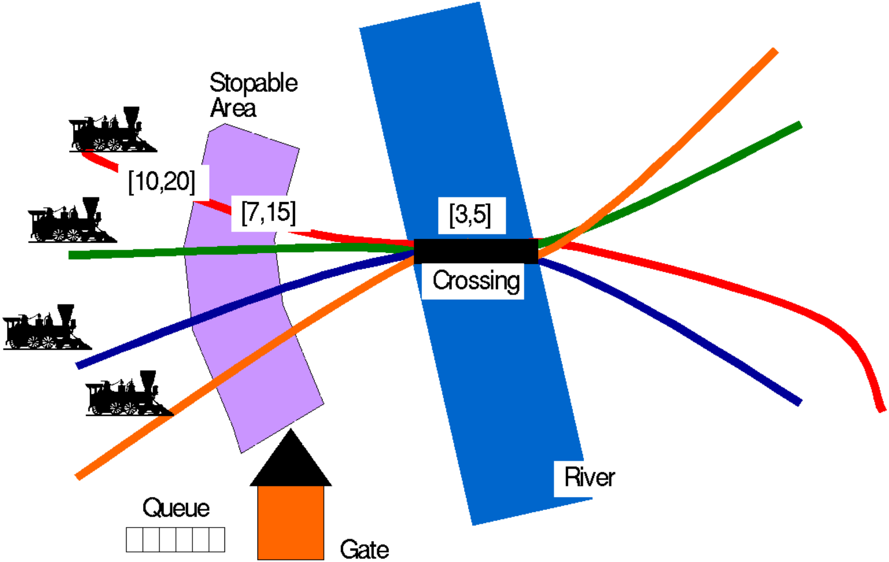
We can reduce p $\texttt{-->}_{<t}$ q to a reachability property:

- Add two broadcast channels *a* and *b*.
- Send on *a* when *p* becomes true, on *b* when *q* becomes true.
- Add a process that goes to an error state when the time between a signal on *a* and *b* reaches *t*.
- Check `A[] not Test.bad`.
- Works even when *p* becomes true several times before *q*.

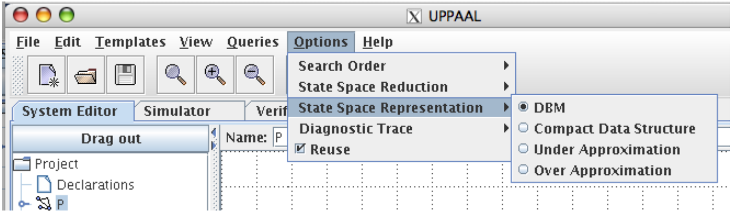




The Train Gate Example



Verification Options in UPPAAL



- Breadth-first
- Depth-first
- State space reduction
- Reuse state space
- State space representation
  - DBM
  - Compact
  - Under approximation
  - Over approximation
- Diagnostic trace

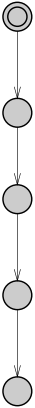
Reachability analysis in UPPAAL

```
$waiting = \{(l_0, Z_0 \wedge I(l_0))\}$
$passed = \varnothing$
while $waiting \neq \varnothing$ do
  $(l, Z) =$ select state from $waiting$
  $waiting = waiting \setminus \{(l, Z)\}$
  if $testProperty(l, Z)$ then return true
  if $\forall (l, Y) \in passed : Z \not\subseteq Y$ then
    $passed = passed \cup \{(l, Z)\}$
    $\forall (l', Z') : (l, Z) \Rightarrow (l', Z')$ do
      if $\forall (l', Y') \in waiting : Z' \not\subseteq Y'$ then
        $waiting = waiting \cup \{(l', Z')\}$
      endif
    done
  endif
done
return false
```

To store or not to store

State space reduction

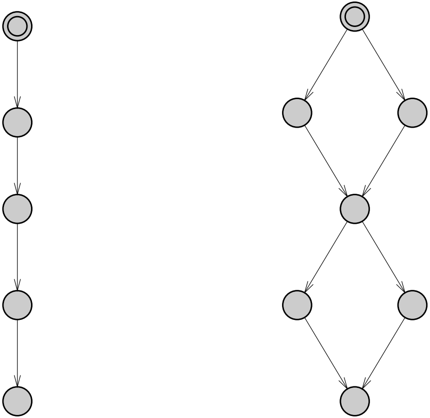
- For acyclic systems, a passed list is not needed to guarantee termination.



To store or not to store

State space reduction

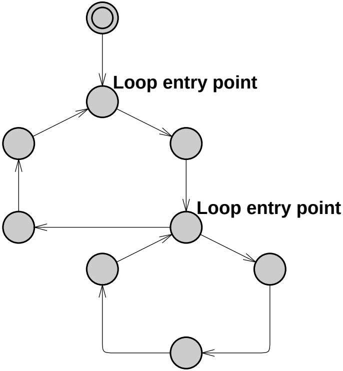
- For acyclic systems, a passed list is not needed to guarantee termination.
- However, it is useful for efficiency.



Loop entry points

State space reduction

Only symbolic states involving loop-entry points need to be stored in the passed list to guarantee termination.



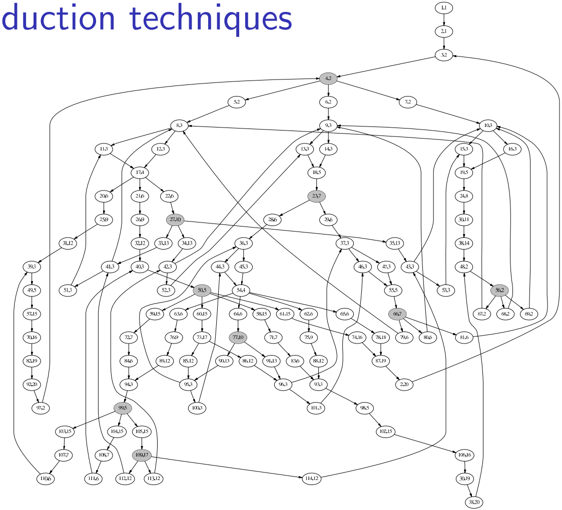
Options for state space reduction

None Store all states.

Conservative Store all non-committed states.

Aggressive Only store loop entry points.

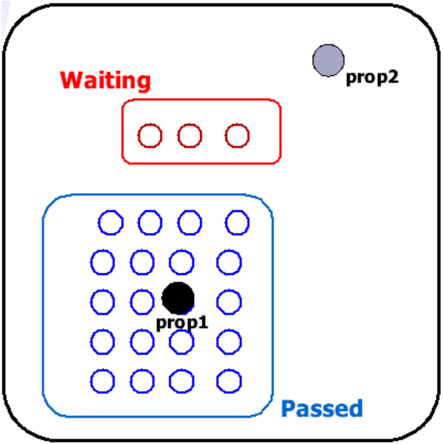
Effect of reduction techniques



- 117 symbolic states
- 81 loop entry points.
- 9 states identified by more extensive analysis.

Less than 10% time overhead.

Reuse state space

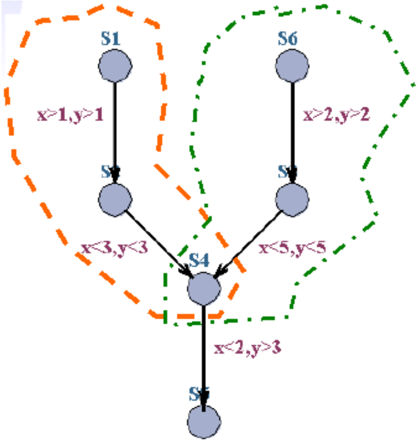


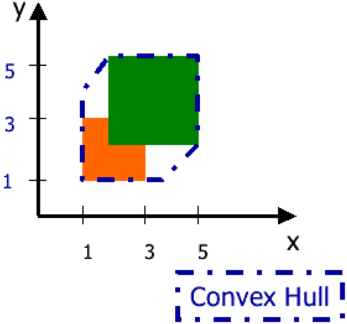
```
A[]  prop1

A[]  prop2
A[]  prop3
A[]  prop4
A[]  prop5
.
.
.
A[]  propn
```

Search in existing **Passed** list before continuing search

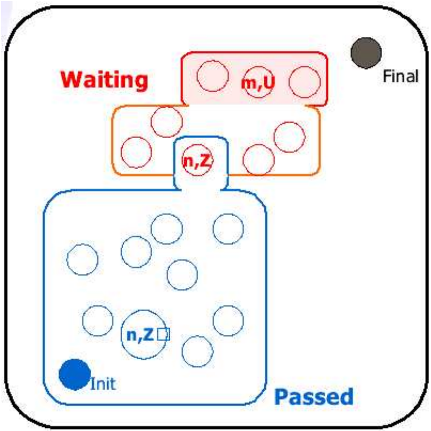
Which order to search?

Over-approximation
Convex Hull



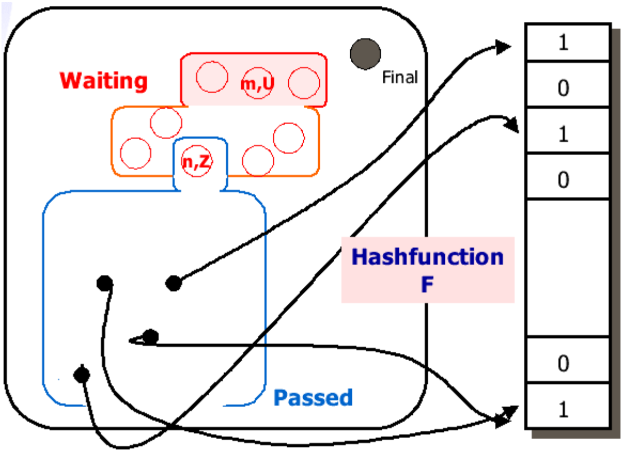


Under-approximation
Bit state hashing



Under-approximation

Bit state hashing


Waiting
m,U
n,Z
Final
Passed
Hashfunction
F
1
0
1
0
0
1