Peeling the Layers of UPPAAL

From a User’s Perspective to the Engine

Alexandre David
Gerd Behrmann, Kim G. Larsen, Wang Yi
Paul Pettersson, Didier Lime, ...
Model-Checking

- **Real-time systems:**
  - Systems where correctness depends on the logical order of events and on their timings!
  - ... in addition to correct computation.

- **Real Time Model-checking:**
  - Model the environment + the tasks.
  - Model $\vdash \varphi$? *Automated* proof.
Controller Synthesis

- Controller synthesis:
  - Model the environment + what a controller *can* do.
  - Generate the controller so that controller $\models \varphi$!
    Generate the *right* code automatically.
  - 2-player timed game: environment moves vs. controller moves.
    $\Rightarrow$ Timed Game Automata.
Refinement

- I/O Automata used to model specifications.
- Check for refinement between models.
- Combine specifications with operators.
Overview

- Part 1: Model-checking with UPPAAL.

- Part 2: Controller synthesis with UPPAAL-TIGA.

- Part 3: Compositional verification with ECDAR.
Part 1 - Overview

- Tool Overview
- Modelling Language
- Specification Language
- Verification Engine
- Implementation
- Verification Options
- Modelling Patterns
Model-Checking - Overview

- Requirements: invariant/safety (something bad never happens), liveness (something good eventually happens).
- Good: intuitive formalism, press-button technology.
- Bad: state-space explosion – how to fight it?

Model: network of TA

Specification: formula in TCTL

UPPAAL

YES
+ some trace

NO
+ some trace
Application to Scheduling

If possible find schedule for all four men to reach safe side in 60 min.
Can be modeled and solved with timed automata in UPPAAL.
Toolkit Overview

Modeling

TA + LSC editor

Simulation

TA + MSC (+Gantt chart)

Verification
Architecture

GUI (Java)

`uppaal2k.jar`

Local or remote

Server

CLI

Engine (C++)

Linux, Windows, MacOS
Modelling Language
TA in a Nutshell

off

push?

x=0

x>5

push?

x==1000

push?

low

x≤1000

x≤5

push?

x==1000

x=0

high

x≤1000

x==1000

x=0

use

push!
Timed Automata in UPPAAL

- Timed Automata with Invariants
  - shake-hand and broadcast communication,
  - urgent action channels,
  - urgent and committed locations,
  - data-variables (with bounded domains),
  - arrays of data-variables,
  - constants,
  - guards and assignments over data-variables and arrays...
  - templates with local clocks, data-variables, and constants.
  - C subset
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`
More on Expressions

- Operators (not clocks):
  - Logical:
    - && (logical and), || (logical or), ! (logical negation),
  - Bitwise:
    - ^ (xor), & (bitwise and), | (bitwise or),
  - Bit shift:
    - << (left), >> (right)
  - Numerical:
    - % (modulo), ? (max)
  - Assignments:
    - +=, -=, *=, /=, ^=, <<=, >>=, :=
  - Prefix and postfix:
    - ++ (increment), -- (decrement)
  - Quantifiers: forall, exists.
  - Sums: sum.
Un-timed Example: Jugs

- 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;
      level[i] -= poured;
      level[j] += poured;
  }
  ```

**Auto-instantiation:** system Jug;
Train-Gate Crossing

Stopable Area

[10,20]

[7,15]

[3,5]

Crossing

River
Train-Gate Modeling

- Scale the model:
  - const int N = 6; typedef int[0,N-1] id_t;
- Trains have their local clocks.
- The gate has its local list & functions.

Communication via channels.
chan appr[N], stop[N], leave[N];
urgent chan go[N];
Implementation of the Queue

id_t list[N+1];
int[0,N] len;

id_t front() { return list[0]; }
id_t tail() { return list[len - 1]; }
void enqueue(id_t element) { list[len++] = element; }

void dequeue()
{
    int i = 0;
    len -= 1;
    while (i < len)
    {
        list[i] = list[i + 1];
        i++;
    }
    list[i] = 0;
}
Scalar Sets

- **Use:** `typedef scalar[N] setA;`
  - defines a set of N scalars,
  - `typedef scalar[N] setB;`
    defines another set of N scalars,
  - it is very important to use the `typedef`.
  - `chan a[setA];` is an array of channels ranging over a scalar set – similarly for other types.
  - limited operations to keep scalars symmetric.
- **A way to specify symmetries in the model.**
  - UPPAAL uses symmetry reduction automatically.
  - Reduction: Project the current state to a representative of its equivalence class (w.r.t. symmetry).
Specification Language
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: $P \rightarrow_{\leq t} Q$

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 (and arrays of these).
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: $P \rightarrow \leq t Q$
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: $P \rightarrow \leq t Q$
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: $P \rightarrow \leq t Q$
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: $P \rightarrow \leq t \ Q$
Jug Example

- Safety: Never overflow.
  - $\forall (i:id_t) \text{ level}[i] \leq \text{ capa}[i]$

- Validation/Reachability: How to get 1 unit.
  - $\exists (i:id_t) \text{ level}[i] == 1$
Train-Gate Crossing

- Safety: One train crossing.
  - $A[] \forall i : \text{id}_t \forall j : \text{id}_t$  
    $\text{Train}(i).\text{Cross} \&\& \text{Train}(j).\text{Cross}$ imply $i == j$

- Liveness: Approaching trains eventually cross.
  - $\text{Train}(0).\text{Appr} \rightarrow \text{Train}(0).\text{Cross}$
  - $\text{Train}(1).\text{Appr} \rightarrow \text{Train}(1).\text{Cross}$
  - ...

- No deadlock.
  - $A[]$ not deadlock
UPPAAL Verification Engine
Outline

- Symbolic Exploration with Zones
- Difference Bound Matrices
  - Operations
- Reachability Algorithm
- Liveness Algorithm
Zones in a Nutshell

From Infinite to Finite

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

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

Zone:
conjunction of $x-y \leq n,
x \leq n,$
$x \geq n$
Symbolic Transitions

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

reachable? x

y <= 2, x >= 4

x = 0, y = 0

y <= 2, x >= 4

L0

L1

Reachable?
Symbolic Exploration

Symbolic Exploration

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

Reachable?

\[ y \]
\[ x \]

Delay
Symbolic Exploration

- \( y := 0 \) leads to \( L_0 \)
- \( y \leq 2 \) leads to \( x := 0 \) and \( x \leq 2 \)
- \( y \leq 2, x = 4 \) leads to \( L_1 \)

Reachable?
Symbolic Exploration

y = 0, x = 0
y ≤ 2
x ≤ 2
y ≤ 2, x ≥ 4
L0
L1
Reachable?

y
y

x

Left

18-11-2010
AFSEC
Symbolic Exploration

\[ y := 0 \]
\[ x := 0 \]
\[ y \leq 2 \]
\[ x \leq 2 \]
\[ y \leq 2, x \geq 4 \]

Reachable?

Delay
Symbolic Exploration

L0: y := 0, x := 0
L1: y <= 2, x <= 2
Left: y <= 2, x >= 4

Reachable?
Symbolic Exploration

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

Reachable?

Left
Symbolic Exploration

reachable?

L0

L1

y := 0

y := 2

y <= 2, x >= 4

x := 0

x <= 2

Delay
Symbolic Exploration

The simulator shows you symbolic states!
A zone $Z$ is a conjunctive formula:

$$g_1 \& g_2 \& \ldots \& g_n$$

where $g_i$ is a clock constraint:

$$x_i \sim b_i \text{ or } x_i - x_j \sim b_{ij}$$

Use a zero-clock $x_0$ (constant 0)

A zone can be re-written as a set:

$$\{x_i - x_j \sim b_{ij} \mid \sim \text{ is } < \text{ or } \leq, i,j \leq n\}$$

This can be represented as a MATRIX, DBM (Difference Bound Matrices)
Solution Set as Semantics

- Let $Z$ be a zone (a set of constraints)

- Let $[Z] = \{ u \mid u$ is a solution of $Z \}$
  - The semantics

(We write $Z$ instead $[Z]$ )
Strongest post-condition (Delay): $SP(Z)$ or $Z^\uparrow$
- $[Z^\uparrow] = \{u+d | d \in \mathbb{R}, u \in [Z]\}$

Weakest pre-condition: $WP(Z)$ or $Z^\downarrow$ (the dual of $Z^\uparrow$)
- $[Z^\downarrow] = \{u | u + d \in [Z] \text{ for some } d \in \mathbb{R}\}$

Reset: $\{x\}Z$ or $Z(x:=0)$
- $[\{x\}Z] = \{u[0/x] | u \in [Z]\}$

Conjunction
- $[Z \& g] = [Z] \cap [g]$
Theorem on Zones

- The set of zones is closed under all constraint operations (including $x:=x-c$ or $x:=x+c$)
- That is, the result of the operations on a zone is a zone
- That is, there will be a zone (a finite object i.e a zone/constraints) to represent the sets: $[Z^\uparrow]$, $[Z^\downarrow]$, $[\{x\}Z]$
One-Step Searchability: \( Si \rightarrow Sj \)

- **Delay:** \((n, Z) \rightarrow (n, Z')\) where \(Z' = Z\uparrow \land \text{inv}(n)\)

- **Action:** \((n, Z) \rightarrow (m, Z')\) where \(Z' = \{x\}(Z \land g)\)

Successors\((n, Z) = \{(m, Z') \mid (n, Z) \rightarrow \rightarrow (m, Z'), Z' \neq \emptyset\}\)

- Sometime we write: \((n, Z) \rightarrow (m, Z')\) if \((m, Z')\) is a successor of \((n, Z)\)
Implementation: Difference Bound Matrices

\[
x_i - x_j \leq c_{ij}
\]

\[
\begin{align*}
x_0 - x_0 & \leq 0 \\
x_1 - x_0 & \leq 6 \\
x_2 - x_0 & \leq 5 \\
x_0 - x_1 & \leq -2 \\
x_1 - x_1 & \leq 0 \\
x_2 - x_1 & \leq 1 \\
x_0 - x_2 & \leq -1 \\
x_1 - x_2 & \leq 3 \\
x_2 - x_2 & \leq 0
\end{align*}
\]
### Difference Bound Matrices

<table>
<thead>
<tr>
<th>x₀ - x₀ ≤ 0</th>
<th>x₀ - x₁ ≤ -2</th>
<th>x₀ - x₂ ≤ -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x₁ - x₀ ≤ 6</td>
<td>x₁ - x₁ ≤ 0</td>
<td>x₁ - x₂ ≤ 3</td>
</tr>
<tr>
<td>x₂ - x₀ ≤ 5</td>
<td>x₂ - x₁ ≤ 3</td>
<td>x₂ - x₂ ≤ 0</td>
</tr>
</tbody>
</table>

\[ xᵢ - xⱼ ≤ cᵢⱼ \]

**Canonical** representation: All constraints **as tight as possible**. Needed for **inclusion checking**. → **Unique** DBM to represent a zone.
DBMs

- How to make them canonical: Floyd-Warshall algorithm.
  
  ```
  for k in 1..dim do
    for i in 1..dim do
      for j in 1..dim do
        dbm[i,j] = min(dbm[i,j],dbm[i,k]+dbm[k,j])
  ```

- Why?
  - Inclusion checking.
  - Unique representation per zone – storage.
  - Note 1: The algorithm leaves negative values on the diagonal for empty zones.
  - Note 2: DBMs can also be seen as graphs.
DBMs

- Future:
  
  ```
  for i in 2..dim do
      dbm[i,1] = infinity
  ```

- Constrain (tighten bounds):
  
  ```
  if old[i,j] ≥ new[i,j] then
      old[i,j] = new[i,j]
      floyd_dim(i,j,old)
  ```

- Reset:
  
  ```
  dbm[k,0] = (value)
  dbm[0,k] = (-value)
  for i in 1..dim do
      dbm[k,i] = dbm[k,0] + dbm[0,i]
      dbm[i,k] = dbm[i,0] + dbm[0,k]
  ```

- More in the DBM library.

- Important: Preserve canonicity.
(The DBM Library

- DBM library (GPL).
  - federations,
  - subtractions,
  - merge.
- Ruby binding (GPL).
- UTAP (UPPAAL TA Parser) library (LGPL).
  - syntax of UPPAAL,
  - canonical TA representation.

http://www.cs.aau.dk/~adavid/UDBM/

http://www.cs.aau.dk/~behrmann/utap/
# DBM Library - Overview

<table>
<thead>
<tr>
<th>Ruby (udbm-sys)</th>
<th>Ruby (udbm-gtk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High level abstraction.</td>
<td>Graphical viewer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ruby (udbm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fed wrapper.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C++ API</th>
</tr>
</thead>
<tbody>
<tr>
<td>High level types.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic functions.</td>
</tr>
</tbody>
</table>
C/C++ API

- Basic functions: delay, constrain, intersection, minimal graph, relation... all basic operations.
- High level types: dbm_t and fed_t.
  - Transparent memory management.
  - Copy-on-write semantics (transparent).
  - Support for different merging/reduction algorithms of federations.
  - More complex operators, e.g., subtractions, predt...
Ruby API

- Fed wrapper.
  - All operations of fed_t.
  - Hooks to the graphical viewer (transparent).
- High level abstraction.
  - Set to represent a set of clock valuations defined by a system of constraints.
  - Context of Clock(s).
- Graphical viewer.
  - Observer for Fed and Set.
- Great educational & research tool!
Forward Reachability Algorithm

\[ \text{INITIAL} \quad \text{Passed} := \emptyset; \]
\[ \text{Waiting} := \{(n_0,Z_0)\} \]

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

\[ \text{UNTIL } \text{Waiting} = \emptyset \]
\[ \text{return false} \]
**Forward Reachability Algorithm**

**INITIAL**

\[
\text{Passed} := \emptyset; \\
\text{Waiting} := \{(n_0, Z_0)\}
\]

**REPEAT**

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

**UNTIL** \[\text{Waiting} = \emptyset\]

**return false**
**Forward Reachability Algorithm**

**INITIAL**

- `Passed := \emptyset;`
- `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 = \emptyset`

return `false`
Forward Reachability Algorithm

**INITIAL**

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

**REPEAT**

- pick \((n,Z)\) in Waiting
- if \((n,Z) = \text{Final}\) return true

**UNTIL** Waiting = Ø

return false
**Forward Reachability Algorithm**

\[\text{INITIAL} \quad \text{Passed} := \emptyset;\]
\[\text{Waiting} := \{(n_{0}, Z_{0})\}\]

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

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

\textbf{Initial} Passed := \emptyset;
Waiting := \{(n_0,Z_0)\}

\textbf{REPEAT}
\begin{itemize}
  \item pick (n,Z) in Waiting
  \item if (n,Z) = Final return true
  \item for all (n,Z)\rightarrow(n',Z'):
    \begin{itemize}
      \item if for some (n',Z'') Z' \subseteq Z'' continue
      \item else add (n',Z') to Waiting
    \end{itemize}
\end{itemize}
\textbf{UNTIL} Waiting = \emptyset
return false
Forward Reachability Algorithm

**INITIAL**

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

**REPEAT**

1. pick \((n,Z)\) in **Waiting**
2. if \((n,Z) = \text{Final}\) return true
3. 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** **Waiting** = Ø

return false
**Forward Reachability Algorithm**

**INITIAL**
- Passed := Ø;
- Waiting := {(n₀,Z₀)}

**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** Waiting = Ø
return false
Liveness Algorithm

proc Eventually(S₀, φ) ==
ST := Ø
Passed := Ø
Search(delay(S₀, ¬φ))
exit(true)
end

proc Search(S) ==
if loop(S, ST) then exit(false) fi
S := S ∧ ¬φ
push(ST, S)
if unbounded(S) ∨ deadlocked(S) then
exit(false) fi
if ∀S' ∈ Passed : S ⊈ S'
then foreach S' : S ⇒ S' do
Search(delay(S', ¬φ))
end
fi
Passed := Passed ∪ {pop(ST)}
end
Liveness Algorithm

\[
\text{proc} \; \text{Eventually}(S_0, \varphi) =
\]
\[
ST := \emptyset
\]
\[
\text{Passed} := \emptyset
\]
\[
\text{Search}(\text{delay}(S_0, \neg \varphi))
\]
\[
\text{exit}(\text{true})
\]
\end

\[
\text{proc} \; \text{Search}(S) =
\]
\[
\text{if} \; \text{loop}(S, ST) \; \text{then} \; \text{exit}(\text{false}) \; \text{fi}
\]
\[
S := S \land \neg \varphi
\]
\[
\text{push}(ST, S)
\]
\[
\text{if} \; \text{unbounded}(S) \lor \text{deadlocked}(S) \; \text{then} \; \text{exit}(\text{false}) \; \text{fi}
\]
\[
\text{if} \; \forall S' \in \text{Passed} : S \not\subseteq S' \; \text{then} \; \text{foreach} \; S' : S \Rightarrow S' \; \text{do} \; \text{Search}(\text{delay}(S', \neg \varphi)) \; \text{od}
\]
\[
\text{fi}
\]
\[
\text{Passed} := \text{Passed} \cup \{\text{pop}(ST)\}
\]
\end
Liveness Algorithm

\begin{verbatim}
proc Eventually(S₀, φ) ==
ST := ∅
Passed := ∅
Search(delay(S₀, ¬φ))
exit(true)
end
proc Search(S) ==
if loop(S, ST) then exit(false) fi
S := S ∧ ¬φ
push(ST, S)
if unbounded(S) ∨ deadlocked(S) then exit(false) fi
if ∀S' ∈ Passed : S ⊈ S'
then foreach S' : S ⊢ S' do
    Search(delay(S', ¬φ))
end
fi
Passed := Passed ∪ {pop(ST)}
end
\end{verbatim}
Liveness Algorithm

\[\text{proc } \text{Eventually}(S_0, \varphi) \equiv\]
\[ST := \emptyset\]
\[\text{Passed} := \emptyset\]
\[\text{Search}(\text{delay}(S_0, \neg \varphi))\]
\[\text{exit}(\text{true})\]
\[\text{end}\]

\[\text{proc } \text{Search}(S) \equiv\]
\[\text{if } \text{loop}(S, ST) \text{ then } \text{exit}(\text{false}) \text{ fi}\]
\[S := S \land \neg \varphi\]
\[\text{push}(ST, S)\]
\[\text{if } \text{unbounded}(S) \lor \text{deadlocked}(S) \text{ then } \text{exit}(\text{false}) \text{ fi}\]
\[\text{if } \forall S' \in \text{Passed} : S \not\subseteq S'\]
\[\text{then } \text{foreach } S' : S \Rightarrow S' \text{ do}\]
\[\text{Search}(\text{delay}(S', \neg \varphi))\]
\[\text{od}\]
\[\text{end}\]

Passed

\[\text{ST}\]

Unexplored
Liveness Algorithm

\[ \text{proc } \text{Eventually}(S_0, \varphi) \equiv \]
\[ ST := \emptyset \]
\[ Passed := \emptyset \]
\[ Search(\text{delay}(S_0, \neg \varphi)) \]
\[ \text{exit}(\text{true}) \]
\[ \text{end} \]

\[ \text{proc } \text{Search}(S) \equiv \]
\[ \text{if } \text{loop}(S, ST) \text{ then } \text{exit}(\text{false}) \text{ fi} \]
\[ S := S \land \neg \varphi \]
\[ \text{push}(ST, S) \]
\[ \text{if } \text{unbounded}(S) \lor \text{deadlocked}(S) \text{ then } \]
\[ \text{exit}(\text{false}) \text{ fi} \]
\[ \text{if } \forall S' \in \text{Passed} : S \not\subseteq S' \]
\[ \text{then } \text{foreach } S' : S \overset{\beta}{\Rightarrow} S' \text{ do} \]
\[ \text{Search(\text{delay}(S', \neg \varphi))} \]
\[ \text{od} \]
\[ \text{fi} \]
\[ Passed := Passed \cup \{ \text{pop}(ST) \} \]
\[ \text{end} \]
Liveness Algorithm

\[
\text{proc } \text{Eventually}(S_0, \varphi) \equiv \\
ST := \emptyset \\
Passed := \emptyset \\
\text{Search}(\text{delay}(S_0, \neg \varphi)) \\
\text{exit}(\text{true}) \\
\text{end}
\]

\[
\text{proc } \text{Search}(S) \equiv \\
\text{if } \text{loop}(S, ST) \text{ then } \text{exit}(\text{false}) \text{ fi} \\
S := S \land \neg \varphi \\
\text{push}(ST, S) \\
\text{if } \text{unbounded}(S) \lor \text{deadlocked}(S) \text{ then } \text{exit}(\text{false}) \text{ fi} \\
\text{if } \forall S' \in \text{Passed} : S \not\subseteq S' \\
\text{then } \text{foreach } S' : S \Rightarrow S' \text{ do } \\
\text{Search}(\text{delay}(S', \neg \varphi)) \\
\text{od} \\
\text{fi} \\
\text{Passed := Passed} \cup \{\text{pop(ST)}\} \\
\text{end}
\]
Liveness Algorithm

\[
\begin{align*}
\text{proc } & \text{Eventually}(S_0, \varphi) \equiv \\
& ST := \emptyset \\
& \text{Passed} := \emptyset \\
& \text{Search}(\text{delay}(S_0, \neg \varphi)) \\
& \text{exit}(\text{true}) \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{proc } & \text{Search}(S) \equiv \\
& \text{if } \text{loop}(S, ST) \text{ then } \text{exit}(\text{false}) \text{ fi} \\
& S := S \land \neg \varphi \\
& \text{push}(ST, S) \\
& \text{if } \text{unbounded}(S) \lor \text{deadlocked}(S) \text{ then } \\
& \quad \text{exit}(\text{false}) \text{ fi} \\
& \text{if } \forall S' \in \text{Passed} : S \not\subseteq S' \\
& \quad \text{then foreach } S' : S \Rightarrow S' \text{ do} \\
& \qquad \text{Search}(\text{delay}(S', \neg \varphi)) \\
& \quad \text{od} \\
& \text{fi} \\
& \text{Passed} := \text{Passed} \cup \{\text{pop}(ST)\} \\
\text{end}
\end{align*}
\]
**Liveness Algorithm**

```
proc Eventually(S₀, φ) =
    ST := ∅
    Passed := ∅
    Search(delay($S₀$, ¬φ))
    exit(true)
end

proc Search(S) =
    if loop(S, ST) then exit(false) fi
    $S := S \land \neg φ$
    push(ST, S)
    if unbounded(S) ∨ deadlocked(S) then
      exit(false)
    fi
    if ∀$S' \in Passed : S \not\subseteq S'$
      then foreach $S' : S \Rightarrow S'$ do
        Search(delay($S'$, ¬φ))
      od
    fi
    Passed := Passed ∪ {pop(ST)}
end
```
Implementation
Outline

- Architecture of UPPAAL
  - Filters
  - Reachability + liveness + leadsto pipelines
  - PWList
- Other optimizations
  - Active clock reduction
  - Sharing
  - Symmetry
  - Reuse
  - Virtual machine
Architecture of UPPAAL

- **Pipeline** architecture
  - In terms of components and flow of data
  - Not with parallel processing units

- Basic components
  - Sink
  - Source
  - Buffer
  - Filter
Pipeline Components

- Source
- Sink
- Buffer
- Filter

Data
  - State
  - Successor
Reachability Pipeline

Initial state → Delay → Extrapolation → Active clock reduction

- Trace
- Successor
- Transition

Accept? (no) → Dealloc
Accept? (yes) → Expression

PWList
Features

- Reusable/exchangeable components
- Flexible architecture
- PWList = passed & waiting list
  - Unified structure
- Early termination
  - Check property after successor computation, not when taking states from waiting list
Delay

- Initial state pushed here
- Future operation + invariant
Extrapolation

- Different algorithms (choice automatic)
  - Correctness depends on which kind of constraints are used
  - Basic extrapolation:

\[ \max_x \]
\[ \max_y \]

+ active clock reduction: if bound = -\( \infty \) then free clock
PWList

- PWList = unified passed and waiting list
- Accept = add state if not included in passed + waiting states
- **IN**: add state to passed + waiting list
- **OUT**: remove from waiting list
Transition & Successor

- Transition computes possible transitions, not states

- Successor computes successor state

Possible resets + variable updates
Leadsto Pipeline

\[ A[\](p \Rightarrow A<> q) \]

Initial state \rightarrow Reachability \rightarrow Liveness

p leadsto q
Standard Passed + Waiting Lists

Searching:
• pop state
• hash
• push to passed (inclusion check)
• successor computation
• hash
• push to waiting queue (inclusion check)

2 hash tables
2 inclusion checks
1 queue
Searching:
• pop state reference
• successor computation
• hash
• push to unified list

(inclusion check) and append state reference
Clock $x$ is inactive at $S$ if on all paths from $S$, $x$ is always reset before being tested.
Active Clock Reduction

Clock $x$ is inactive at $S$ if on all paths from $S$, $x$ is always reset before being tested.

$$\text{Act}(S) = \bigcup_i \text{Clock}(g_i)$$
$$\bigcup_i (\text{Act}(S_i)/\text{Clock}(r_i))$$

Only save constraints on active clocks.

Definition
Data Sharing

- Key idea: Working states different from stored states
  - Working states optimized for computation
    Symbolic state = discrete part (location+variables) + symbolic part (DBM).
  - Stored states optimized for memory
    Stored state = <lockey,varkey,dbmkey>. 
Data Sharing

Symbolic state for computation
Location vector
Variables
DBM

Symbolic state for storage (PWList)
lockey
varkey
dbmkey

Sharing of data
Discrete storage
Symbolic storage

~80% memory reduction.

Easy to change the implementation to favor speed over memory.
Data Sharing

- In practice: 80% reduction.
- Easy to change storage implementation to favor speed or memory.
  - Compression of integer paired with minimal graph
  - Convex hull is a special storage
### PWList & Sharing in Figures

<table>
<thead>
<tr>
<th>Model</th>
<th>Before</th>
<th>Unification</th>
<th>Unification &amp; Sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Audio</td>
<td>$\leq 0.5,s$</td>
<td>$\leq 0.5,s$</td>
<td>$\leq 0.5,s$</td>
</tr>
<tr>
<td>Engine</td>
<td>$\leq 0.5,s$</td>
<td>$\leq 0.5,s$</td>
<td>$\leq 0.5,s$</td>
</tr>
<tr>
<td>Dacapo</td>
<td>3s</td>
<td>3s</td>
<td>3s</td>
</tr>
<tr>
<td>Cups</td>
<td>43s</td>
<td>37s</td>
<td>36s</td>
</tr>
<tr>
<td>BC</td>
<td>428s</td>
<td>359s</td>
<td>345s</td>
</tr>
<tr>
<td>Master</td>
<td>306s</td>
<td>277s</td>
<td>267s</td>
</tr>
<tr>
<td>Slave</td>
<td>440s</td>
<td>377s</td>
<td>359s</td>
</tr>
<tr>
<td>Plant</td>
<td>$19688,s$</td>
<td>$9207,s$</td>
<td>$8513,s$</td>
</tr>
</tbody>
</table>

[SPIN03]
Symmetry Reduction

- Exploitation of full symmetry may give factorial reduction.
- Many timed systems are inherently symmetric.
- Computation of canonical state representative using swaps.
Symmetry Reduction
Support For Symmetry

- **Scalar set based symmetry reduction**
  - `typedef scalarset[4] pid_t; scalarset[n] = {0,...,n-1}`
  - `int[0,4] = set of integers`

- **Template sets**
  - `process P[i:pid_t](...) {(i)}`

- **Iterators**
  - `for (i:pid_t) { a[i+1]=0 }`

- **Quantifiers**
  - `forall (i:int[0,4]) a[i+1]==0`
  - `exists (i:int[0,4]) a[i+1]==1`

- **Selection**
  - `select i: int[0,4]; guard...`

Martijn Henriks, Nijmegen U
Re-using the State-space

- Several properties to check:
  \[ A[] \text{ prop1} \]
  \[ A[] \text{ prop2} \]
  ...
- Search in existing passed list (from previous checks) first.
- Expand missing states (not all states stored).
Virtual Machine

- Expressions (guards & actions) are compiled to *bytecode* and executed by a virtual machine.
- Stack machine, minimal instruction set, peep-hole optimization.
- Open the door to other optimizations or use of 3rd party VM.

Verification Options
Verification Options

**Search Order**
- Depth First
- Breadth First

**State Space Reduction**
- None
- Conservative
- Aggressive

**State Space Representation**
- DBM
- Compact Form
- Under Approximation
- Over Approximation

**Diagnostic Trace**
- Some
- Shortest
- Fastest
Conservative Reduction

Passed list is not needed for termination when there is no loop...

but useful for efficiency.
Conservative Reduction

In case of loops, it is enough to store loop entry points to ensure termination.

*Slight loss in efficiency, good gain in memory.*
TACAS04: An **EXACT** method performing as well as Convex Hull has been developed based on abstractions taking max constants into account.
Under-approximation

Bitstate Hashing

Diagram: PW, Waiting, Final, Init, Passed
Under-approximation

Bitstate Hashing

Hash function

1 bit per passed state

Under-approx. Several states may collide on the same bit.

Inclusion check only with waiting states. “Equality” with passed.

Bit Array
Compact Representation

*Minimal Constraint Form*

\[
\begin{align*}
&x_1 - x_2 \leq 4 \\
&x_2 - x_1 \leq 10 \\
&x_3 - x_1 \leq 2 \\
&x_2 - x_3 \leq 2 \\
&x_0 - x_1 \leq 3 \\
&x_3 - x_0 \leq 5 \\
\end{align*}
\]

Shortest Path Closure $O(n^3)$

Shortest Path Reduction $O(n^3)$

Space worst $O(n^2)$

Verification option “CDS”.

Large gain in space. Small price in time.
Graph Reduction Algorithm

1. Equivalence classes based on 0-cycles.

G: weighted graph
Graph Reduction Algorithm

1. Equivalence classes based on 0-cycles.

2. Graph based on representatives. Safe to remove redundant edges.
Graph Reduction Algorithm

1. Equivalence classes based on 0-cycles.

2. Graph based on representatives. Safe to remove redundant edges.

3. **Shortest Path Reduction**
   =
   One cycle pr. class
   +
   Removal of redundant edges between classes

G: weighted graph

Canonical given order of clocks
Modelling Patterns
Variable Reduction

- Reduce size of state space by explicitly resetting variables when they are not used!

- Automatically performed for clock variables (active clock reduction)

```c
// Remove the front element of the queue
void dequeue()
{
    int i = 0;
    len -= 1;
    while (i < len)
    {
        list[i] = list[i + 1];
        i++;
    }
    list[i] = 0;
}
```
# Synchronous Value Passing

<table>
<thead>
<tr>
<th></th>
<th>Unconditional</th>
<th>Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>One-way</strong></td>
<td>![Diagram for One-way Unconditional]</td>
<td>![Diagram for One-way Conditional]</td>
</tr>
<tr>
<td><strong>Asymmetric two-way</strong></td>
<td>![Diagram for Asymmetric two-way Unconditional]</td>
<td>![Diagram for Asymmetric two-way Conditional]</td>
</tr>
</tbody>
</table>
Atomicity

- Loops & complex control structures: C-functions.
- To allow encoding of multicasting.
- Committed locations.
Bounded Liveness

- Leads to within: $\varphi \rightarrow_{\leq t} \psi$
  - More efficient than leadsto: $\varphi \leadsto_{\leq t} \psi$ reduced to $A\Box(b \Rightarrow z \leq t)$ with
    - bool $b$ set to true and clock $z$ reset when $\varphi$ holds.
    - When $\psi$ holds set $b$ to false.
**Bounded Liveness**

- The truth value of $b$ indicates whether or not $\psi$ should hold in the future.

\[
\begin{align*}
\neg \varphi & \quad b = \text{true} \quad z = 0 \quad b = \text{false} \\
\psi & \quad b = \text{false} \\
\varphi & \quad b = \text{false} \\
\neg \psi & \quad b \text{ true, check } z \leq t
\end{align*}
\]

A[] (b imply $z \leq t$)
E<> b (for meaningful check)
Timers

Parametric timer:

- (re-)start(value)
  start! var=value

- expired?
  active (bool)

- active go?
  (bool+urgent chan)

- time-out event
  timeout?

Declare 'to' with a tight range.
Urgent Edges

- **Intent:** take an edge as soon as it is enabled (without delay).
  - Condition on the edge, not the location.
  - Solution limit: no clock constraint (yet).

```
x <= 2
```

```bash
urgent chan go;
```
Zenoness

- **Problem:** UPPAAL does not check for zenoness directly.
  - A model has “zeno” behavior if it can take an infinite amount of actions in finite time.
  - That is usually not a desirable behavior in practice.
  - Zeno models may wrongly conclude that some properties hold though they logically should not.
  - Rarely taken into account.

- **Solution:** Add an observer automata and check for non-zenoness, i.e., that time will always pass.
Detect by adding the observer:

- \( x = 0 \)
- \( x \leq 1 \)
- \( x = 1 \)
- \( x \leq 10 \)

Constant (10) can be anything (>0), but choose it well w.r.t. your model for efficiency. Clocks 'x' are local.

- and check the property
  \[ \text{ZenoCheck.A} \rightarrow \text{ZenoCheck.B} \]
Some Pitfalls

- Unbounded integers
  - Model uses the full range.
- Unsynchronized processes
  - Combinatorial explosion.
- Unused active variables specially in arrays
Case-Studies: Controllers

- Gearbox Controller [TACAS’98]
- Bang & Olufsen Power Controller [RTPS’99, FTRTFT’2k]
- SIDMAR Steel Production Plant [RTCSA’99, DSVV’2k]
- Real-Time RCX Control-Programs [ECRTS’2k]
- Experimental Batch Plant (2000)
- RCX Production Cell (2000)
- Terma, Memory Management for Radar (2001)
Case Studies: Protocols

- Philips Audio Protocol [HS’95, CAV’95, RTSS’95, CAV’96]
- Collision-Avoidance Protocol [SPIN’95]
- Bounded Retransmission Protocol [TACAS’97]
- Bang & Olufsen Audio/Video Protocol [RTSS’97]
- TDMA Protocol [PRFTS’97]
- Lip-Synchronization Protocol [FMICS’97]
- Multimedia Streams [DSVIS’98]
- ATM ABR Protocol [CAV’99]
- ABB Fieldbus Protocol [ECRTS’2k]
End Part 1