MoDeST:
From Theory to Industrial Experience

Joost-Pieter Katoen

Department of Electrical Engineering, Mathematics and Computer Science
University of Twente

joint work with: Henrik Bohnenkamp, Pedro R. D'Argenio, Holger Hermanns
Ric Klaren, Angelika Mader, Yaroslav Usenko

School on Formal Methods: Real-Time, September 2004

Relevant system properties

- Is the system correct? verification
- Are two models the “same”? equivalence checking
- Does the implementation conform to the model? testing
- Is it timely? real-time verification
- What is its performance? performance evaluation
- What is its availability? dependability analysis

Diversity of (rigorous) tools and models

- Verification: model checkers — SPIN and NuSMV
- Model comparison: bisimulation checkers — μCRL toolset and CADP
- Conformance testing: test generation tools — ToRX and TGV
- Real-time verification: model checkers — UPPAAL, Kronos
- Performance evaluation: simulators and analyzers — Marca, GreatSPN
- Dependability evaluation: — Möbius

Our philosophy

- How to ensure consistency between the different models used?
- Who wants to specify all these models anyway?
- Our view:

  A single-specification, multi-solution approach

- One (core) formalism, with one semantics
- Extract models for particular property of interest
- Provide mappings from other modeling languages (UML statechart)

Overview

- Introduction and motivation

⇒ The MoDeST modeling language

  - language constructs and examples
  - operational semantics and design rationales

- The MOTOR tool environment

  - tool architecture + current status
  - integration with MÖBIUS

- Experience in industrial case studies

  - checking device absence in UPnP (Philips)
  - assessment of schedules for lacquer plants (Axxom AG)

- Epilogue and future work

What is MoDeST?

- A formal specification language for

  MOdelling and DEscription of Stochastic and Timed systems

- Inspired by:
  - process algebra FSP [Kramer & McGee]
  - stochastic process algebra (with nondeterminism) [Bravetti, D'Argenio & Katoen, Hermanns]
  - timed automata [Alur & Dill, Bornot & Sifakis]
  - (simple) probabilistic automata [Segala]
  - Promela [Holzmann]
  - . . . a bit of Java

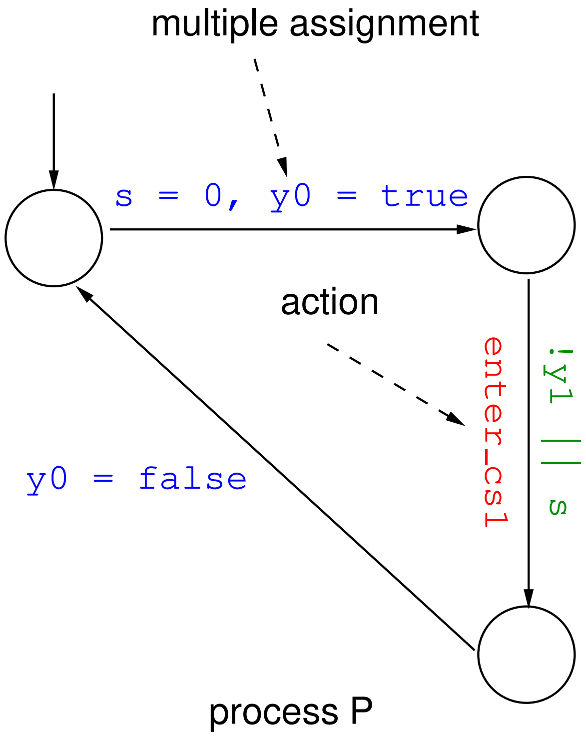
- Exploiting results from concurrency theory for probabilistic systems

Design issues of concern

- Compositionality
- Orthogonality of language concepts
- Usability: aim at modern and “light-weight” language
  - syntax (and data types) close to C and Promela
  - control structures like loops and exception handling
  - possibilities to link user-defined libraries
  - (limited) control of the atomicity of assignments
- Expressiveness: transition- and stochastic systems
  - nondeterminism of actions and timing
  - probabilistic behaviour of actions and timing
- Clean and formal semantics

Shared variable communication

```
action enter_cs1, enter_cs2;
bool s = true, y0 = 0, y1 = 0;

process P() {
  do { :: {= s = false, y0 = true =};
          when (!y1 || s)
            enter_cs1;  // CS
          y0 = false;   // leave CS
}     }

process Q() {
  do { :: {= s = true, y1 = true =};
          when (!y0 || !s)
            enter_cs2;  // CS
          y1 = false
}     }

par { :: P() :: Q() }
```



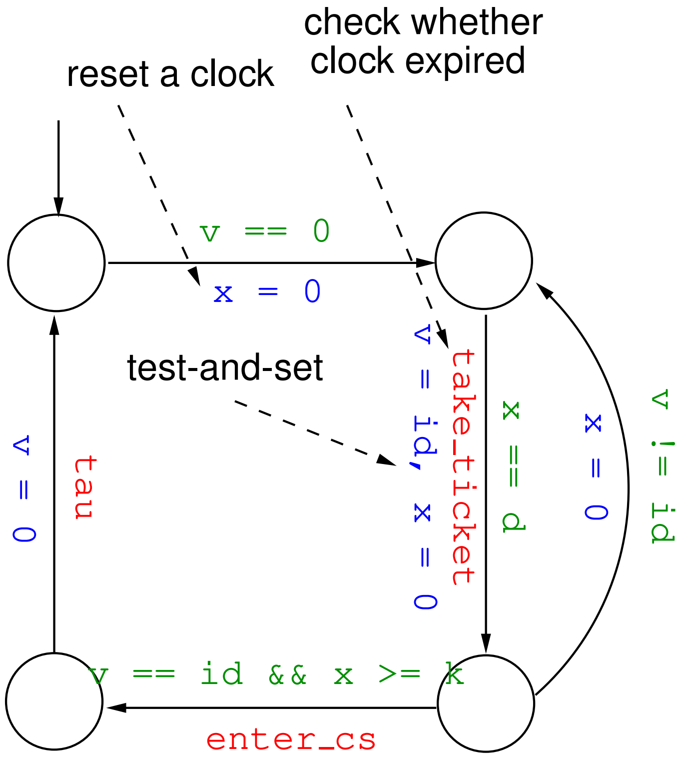
Pnueli's mutual exclusion protocol

Nondeterministic timing

```
int v;                                 // ticket

process P (int id) {                   // behaviour of a single thread
  clock x;                             // P's private timer
  const float k = 2.0, d = 1.0;
  do {
  :: when (v == 0) x = 0;              // once ticket is free, start timer
     do {
     :: when (x == d)                  // wait for exactly d time units
          take_ticket {= v = id, x = 0 =};  // grab the ticket
        alt {
        :: when (v != id) x = 0;       // no longer own ticket
        :: when (v == id && x >= k) // ticket is still yours
             enter_cs;
           break {= v = 0 =}           // release ticket
} }  }  }

par {
:: relabel { take_ticket, enter_cs } by { take_ticket1, enter_cs1 } in P(1)
:: relabel { take_ticket, enter_cs } by { take_ticket2, enter_cs2 } in P(2)
}
```

A timed automaton

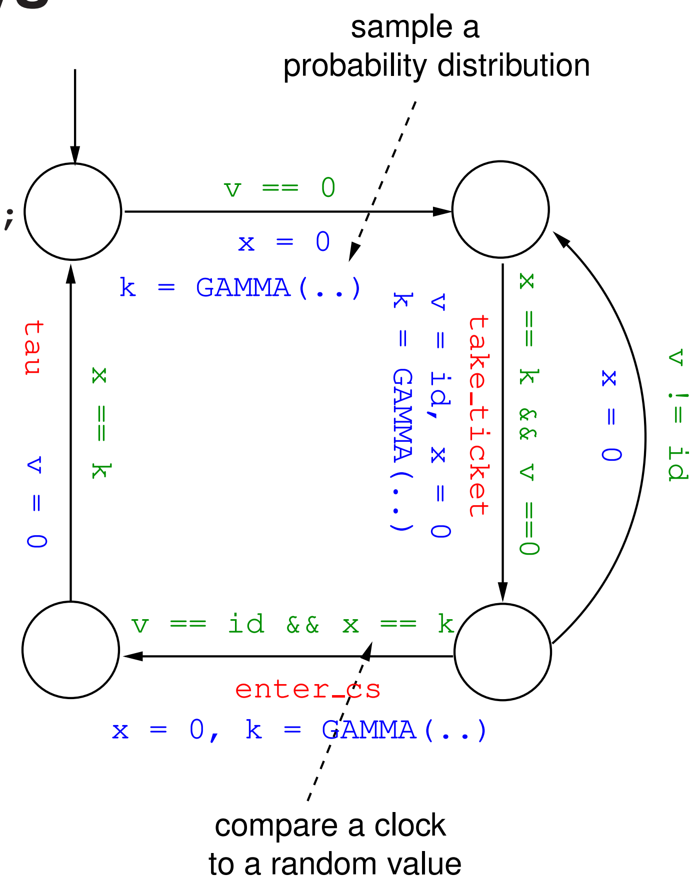
```
process P (int id) {
  clock x;
  const float k = 2.0, d = 1.0;
  do {
  :: when (v == 0) x = 0;
     do {
     :: when (x == d)
          take_ticket {= v = id, x = 0 =};
        alt {
        :: when (v != id) x = 0;
        :: when (v == id && x >= k)
             enter_cs;
           break {= v = 0 =}
} }  }  }
```



Fischer's mutual exclusion protocol

Random delays

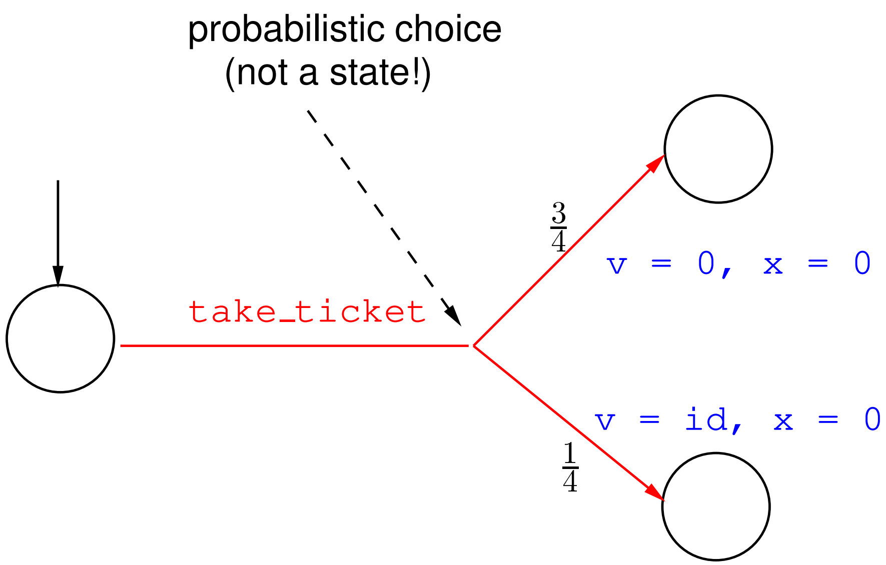
```
process P (int id) {
  clock x;
  float k;
  do {
  :: when (v == 0) {= k = GAMMA(...), x = 0 =};
     do {
     :: when ( (x == k) && (v == 0) )
          take_ticket
          {= v = id, k = GAMMA(...), x = 0 =};
        alt {
        :: when ( (v != id) ) break
        :: when ( (x == k) && (v == id) )
             enter_cs {= k = GAMMA(...) =};
           when (x == k) break {= v = 0 =}
} }  }  }

par {
:: relabel .... in P(1)
:: relabel .... in P(2)
}
```



Gafni and Mitzenmacher's random mutual exclusion protocol

Probabilistic choice

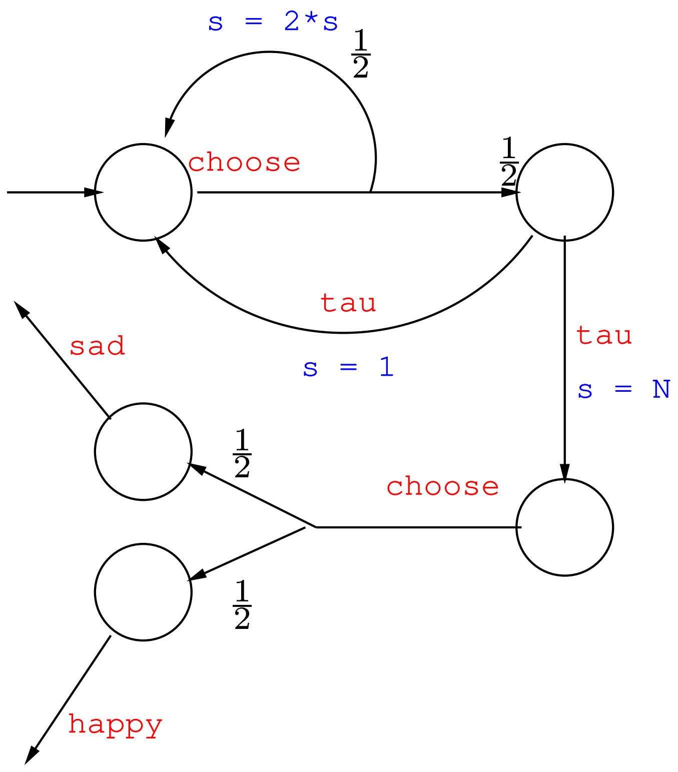
```
take_ticket palt {
            :3: {= v = 0, x = 0 =} P(v)
            :1: {= v = id, x = 0 =} P(v)
            }
```



action-guardedness is required to avoid asynchronous parallel composition

Mixed probability and nondeterminism

```
int s = 1;   // stake

do { :: choose palt {
                :1: s *= 2;
                :1: alt {
                     :: s = N; break;
                     :: s = 1;
   }                 }  }
choose palt {
        :1: happy;
        :1: sad;
       }
```



Language ingredients

- Parallelism
  - shared variable communication
  - handshaking (as in process algebras)
  - interleaving of autonomous actions
- Multiple assignments (= atomicity) and actions
- Control structures
  - loops, alternative, and probabilistic choice
  - exception raising and handling
- Continuous timing (both deterministic and random)
- Nondeterminism (both in actions and in timing)

Syntax of simple processes

act	action
stop	deadlock
error	unhandled error
break	break process
throw$(excp)$	throw exception
process *ProcName*$(t_1\ x_1, \ldots, t_k\ x_k)$ $\{dcl\ P\}$	process declaration
ProcName$(e_1, \ldots, e_k)$	process instantiation
extend $\{act_1, \ldots, act_k\}$ P	alphabet extension
relabel $\{act_1, \ldots, act_k\}$ by $\{act'_1, \ldots, act'_k\}$ P	relabeling

process bodies need to be guarded

Syntax of composed processes

$\mathsf{when}(b)\ P$	enabling
$\mathsf{urgent}(b)\ P$	urgency
$\mathsf{par}\{::\ P_1\ \ldots\ ::\ P_k\}$	parallel
$P_1;\ P_2$	sequential
$\mathsf{do}\{::\ P_1\ \ldots\ ::\ P_k\}$	loop
$\mathsf{alt}\{::\ P_1\ \ldots\ ::\ P_k\}$	alternative
$act\ \mathsf{palt}\ \{:w_1:\ \ asgn_1\ ;\ P_1\ \ldots\ :w_k:\ \ asgn_k\ ;\ P_k\}$	probabilistic choice
$\mathsf{try}\{P\}\ \mathsf{catch}\ \ excp_1\ \{P_1\}\ \ldots\ \mathsf{catch}\ \ excp_k\ \{P_k\}$	exception handling

data types include `bool`, `int`, `float`, `clock`, `array` and `struct`

weights may be arbitrary arithmetic expressions

Some handy syntactic sugar

$$\mathsf{alt}\{::\ \mathsf{when}(b_1)\ P_1\ \ldots\ ::\ \mathsf{when}(b_k)\ P_k\ ::\ \mathsf{else}\ Q\}$$

$$\stackrel{\mathrm{def}}{=}\ \mathsf{alt}\{::\ \mathsf{when}(b_1)\ P_1\ \ldots\ ::\ \mathsf{when}(b_k)\ P_k\ ::\ \mathsf{when}(\neg \textstyle\bigvee_{i=1}^{k} b_i)\ Q\}$$

$$\langle x_1 = e_1, \ldots, x_k = e_k\rangle \stackrel{\mathrm{def}}{=} \mathsf{urgent}(tt)\ \tau\ \mathsf{palt}\ \{:1:\ \langle x_1 = e_1, \ldots, x_k = e_k\rangle\ ;\ \surd\}$$

$$\mathsf{hide}\ \{act_1, \ldots, act_k\}\ P \stackrel{\mathrm{def}}{=} \mathsf{relabel}\ \{act_1, \ldots, act_k\}\ \mathsf{by}\ \{\underbrace{\tau, \ldots, \tau}_{k\ \text{times}}\}\ P$$

$$\mathsf{invariant}(b)\ P \stackrel{\mathrm{def}}{=} \mathsf{alt}\{::\ P\ ::\ \mathsf{urgent}(\neg b)\ \mathsf{when}(\mathit{ff})\ \mathsf{throw}(inv)\ \}$$

$$\not\stackrel{\mathrm{def}}{=} \mathsf{urgent}(\neg b)\ P$$

$$\mathsf{while}(b)\{P\} \stackrel{\mathrm{def}}{=} \mathsf{do}\{::\ \mathsf{when}(b)\ P\ ::\ \mathsf{else\ break}\}$$

Overview

- Introduction and motivation

⇒ The MODEST modeling language

  - language constructs and examples
  - operational semantics and design rationales

- The MOTOR tool environment

  - tool architecture + current status
  - integration with MÖBIUS

- Experience in industrial case studies

  - checking device absence in UPnP (Philips)
  - assessment of schedules for lacquer plants (Axxom AG)

- Epilogue and future work

Two layers of semantics

MoDeST Language

$\Downarrow$ Structural Operational Semantics

Discrete State Model: Stochastic Timed Automata

$\Downarrow$ Layered Semantics

Continuous State Model: Timed Probabilistic Transition Systems

Stochastic timed automata

- S, a set of locations
- Act, a set of actions
- $\longrightarrow$, a transition relation such that $s \xrightarrow{a,g,d} \mathcal{P}$ where
  - s is the current location
  - action $a \in \mathsf{Act}$ is offered
  - g is a guard (boolean expression)
  - d is a deadline (boolean expression)
  - $\mathcal{P}$ is a (discrete) probability spaces over pairs (assignments, location)

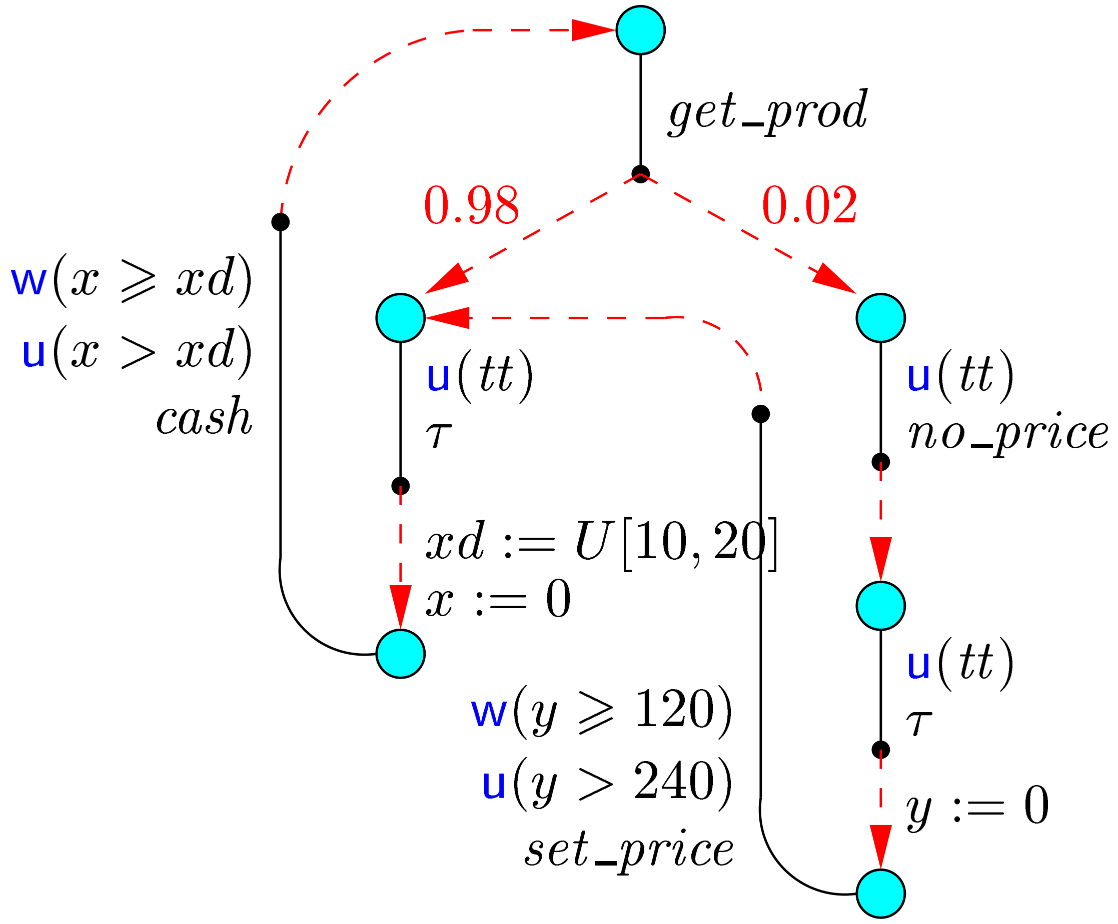
mixture of timed automata with deadlines [Bornot & Sifakis], stochastic automata [D'Argenio *et al.*], and simple probabilistic automata [Segala]

Interpretation of stochastic timed automata

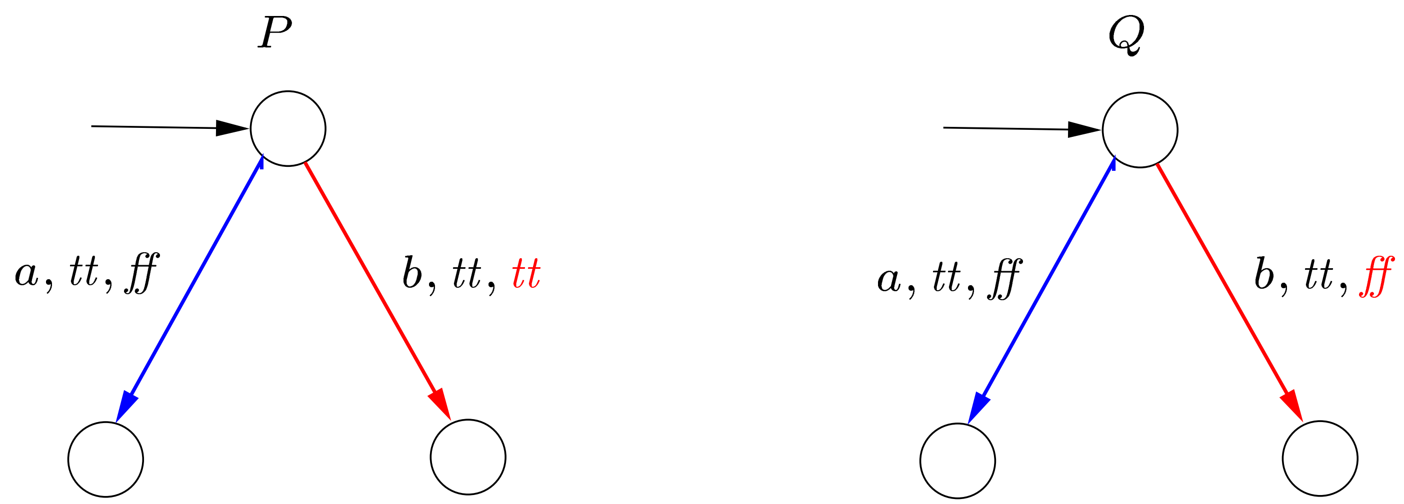
- A state records the current location and valuation of all variables
- If $s \xrightarrow{a,g,d} \mathcal{P}$ and the current valuation satisfies g, then:
  - with probability $\mathbf{P}(A, s')$ where $\mathbf{P}$ being the probability measure of $\mathcal{P}$
  - the valuation is changed according to the sequence of assignments A, and
  - the next location is s'
- Time is advanced with $t > 0$ if in the current location:
  - no deadline is met for any $t' \leqslant t$
  - increasing all clock variables with t
  - letting all other variables unchanged

continuous space model: timed probabilistic transition system

An example stochastic timed automaton

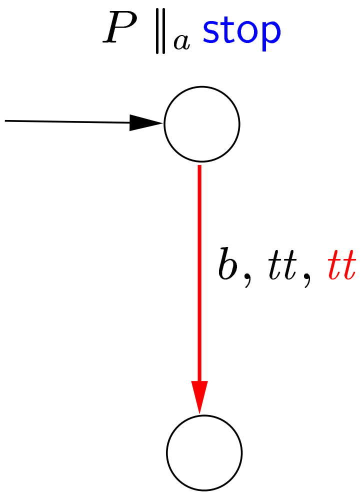


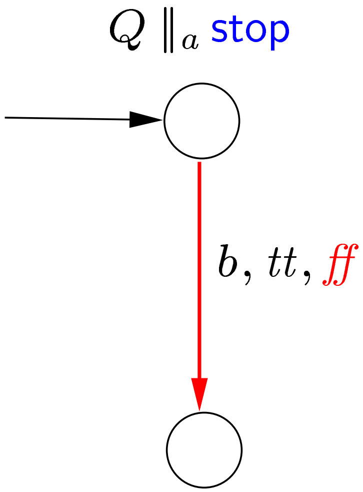
A note on bisimulation



both P and Q can perform a or b only immediately

A note on bisimulation





$P \parallel_a$ stop can idle arbitrarily before performing b but $Q \parallel_a$ stop cannot

a congruence is obtained when e.g., only invariants are used

Submodels

	LTS	PTS/MDP	TA	PTA	DTMC	CTMC	CTMDP	GSMP	SA	STA
probabilistic branching	-	+	-	+	+	+	+	+	+	+
clocks	-	-	+	+	-	R	R	+	+	+
random variables	-	-	-	-	-	R	R	+	+	+
delay nondeterminism	-	-	+	+	-	-	-	-	-	+
action nondeterminism	+	+	+	+	-	-	+	-	+	+

except for action nondeterminism, all other ingredients can be syntactically checked

Operational semantics: simple processes

$$a \xrightarrow{a,tt,ff} \mathcal{P} \text{ with } \mathbf{P}(\varnothing, \surd) = 1$$

$$\mathsf{error} \xrightarrow{\perp,tt,ff} \mathcal{P} \text{ with } \mathbf{P}(\varnothing, \mathsf{error}) = 1$$

$$\mathsf{break} \xrightarrow{\dagger,tt,ff} \mathcal{P} \text{ with } \mathbf{P}(\varnothing, \surd) = 1$$

$$\frac{P \xrightarrow{a,g,d} \mathcal{P}}{\mathit{Proc}(e_1, \ldots, e_k) \xrightarrow{a,g,d} \mathcal{P}} \quad \text{provided } \mathsf{process}\ \mathit{Proc}(x_1, \ldots, x_k)\{P\}$$

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad f = [a_1/a_1', \ldots, a_k/a_k']}{\mathsf{relabel}\ P\ \mathsf{by}\ f \xrightarrow{f(a),g,d} \mathcal{R}}$$

where $\mathbf{R}(A, \surd) = \mathbf{P}(A, \surd)$ and $\mathbf{R}(A, \mathsf{relabel}\ P'\ \mathsf{by}\ f) = \mathbf{P}(A, P')$

Operational semantics: composed processes

$$\frac{P \xrightarrow{a,g,d} \mathcal{P}}{\textsf{when}(b)\ P \xrightarrow{a,b\,\wedge\,g,d} \mathcal{P}}$$

$$\frac{P \xrightarrow{a,g,d} \mathcal{P}}{\textsf{urgent}(b)\ P \xrightarrow{a,g,b\,\vee\,d} \mathcal{P}}$$

$$\frac{P_i \xrightarrow{a,g,d} \mathcal{P}_i \qquad (1 \leqslant i \leqslant k)}{\textsf{alt}\{::\ P_1\ \ldots\ ::\ P_k\} \xrightarrow{a,g,d} \mathcal{P}_i}$$

$$a\ \textsf{palt}\ \{:w_i:\ A_i\ ;\ P_i\}_{i\in I} \xrightarrow{a,tt,ff} \mathcal{P} \text{ with } \mathbf{P}(A_i, P_i) = \frac{w_i}{\sum_{j\in I} w_j}$$

Iteration

$$\mathsf{do}\{::\ P_i\}_{i \in I} \stackrel{\text{def}}{=} \mathsf{auxdo}\{\mathsf{alt}\{::\ P_i\}_{i \in I}\}\{\mathsf{alt}\{::\ P_i\}_{i \in I}\}$$

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad (a \neq \dagger)}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{a,g,d} \mathcal{R}}$$

where $\mathbf{R}(A, \mathsf{auxdo}\{Q\}\{Q\}) = \mathbf{P}(A, \surd)$ and $\mathbf{R}(A, \mathsf{auxdo}\{P'\}\{Q\}) = \mathbf{P}(A, P')$

$$\frac{P \xrightarrow{\dagger,g,d} \mathcal{P}}{\mathsf{auxdo}\{P\}\{Q\} \xrightarrow{\tau,g,d} \mathcal{R}} \text{ where } \mathbf{R}(\varnothing, \surd) = 1$$

Exception handling

raising an exception $\mathsf{throw}(excp) \xrightarrow{excp,tt,ff} \mathcal{P}$ with $\mathbf{P}(\varnothing, \mathsf{error}) = 1$

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad \forall i \in I \,.\, a \neq excp_i}{\mathsf{try}\{P\}\{\mathsf{catch}\ excp_i\ \{P_i\}\}_{i \in I} \xrightarrow{a,g,d} \mathcal{R}}$$

where $\mathbf{R}(A, \mathsf{try}\{P'\}\{\mathsf{catch}\ excp_i\ \{P_i\}\}_{i \in I}) = \mathbf{P}(A, P')$ and $\mathbf{R}(A, \surd) = \mathbf{P}(A, \surd)$

$$\frac{P \xrightarrow{excp_i,g,d} \mathcal{P} \qquad i \in I}{\mathsf{try}\{P\}\{\mathsf{catch}\ excp_i\ \{P_i\}\}_{i \in I} \xrightarrow{\tau,g,d} \mathcal{R}} \text{ with } \mathbf{R}(\varnothing, P_i) = 1$$

Parallel composition

note that: $\texttt{par}\{:: P_1 \ \ldots \ :: P_k\} \stackrel{\text{def}}{=} (\ldots((P_1 \parallel_{B_1} P_2)\ldots)) \parallel_{B_{k-1}} P_k$

with $\parallel_B$ is CSP parallel composition and $B_j = \left(\bigcup_{i=1}^{j} \alpha(P_i)\right) \cap \alpha(P_{j+1})$

performing autonomous actions

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad a \notin B}{P \parallel_B Q \xrightarrow{a,g,d} \mathcal{R}} \quad \text{with} \quad \begin{array}{lcl} \mathbf{R}(A, P' \parallel_B Q) & = & \mathbf{P}(A, P') \\ \mathbf{R}(A, Q \backslash B) & = & \mathbf{P}(A, \surd) \end{array}$$

Synchronization

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad Q \xrightarrow{a,g',d'} \mathcal{Q} \qquad a \in B}{P \parallel_B Q \xrightarrow{a,g \wedge g',d \wedge d'} \mathcal{R}}$$ if a is patient

$$\frac{P \xrightarrow{a,g,d} \mathcal{P} \qquad Q \xrightarrow{a,g',d'} \mathcal{Q} \qquad a \in B}{P \parallel_B Q \xrightarrow{a,g \wedge g',d \vee d'} \mathcal{R}}$$ if a is impatient

where

$$\begin{aligned}
\mathbf{R}(A \cup A', P' \parallel_B Q') &= \mathbf{P}(A, P') \cdot \mathbf{Q}(A', Q') \\
\mathbf{R}(A \cup A', \surd) &= \mathbf{P}(A, \surd) \cdot \mathbf{Q}(A', \surd) \\
\mathbf{R}(A \cup A', Q' \backslash B) &= \mathbf{P}(A, \surd) \cdot \mathbf{Q}(A', Q') \text{ and symmetric}
\end{aligned}$$

if $A \cup A'$ is not a proper assignment, a predefined exception is raised

Adapted rules for call-by-value

$$a \; \textsf{palt} \; \{ :w_i: \; A_i \; ; \; P_i \}_{i \in I} \xrightarrow{a, tt, f\!f} \mathcal{P} \text{ with } \mathbf{P}(A_i \cup \mathbf{A}(P_i), P_i) = \frac{w_i}{\sum_{j \in I} w_j}$$

$$\frac{P \xrightarrow{a,g,d} \mathcal{P}}{P \; ; \; Q \xrightarrow{a,g,d} \mathcal{R}}$$

where $\mathbf{R}(A \cup \mathbf{A}(Q), \surd) = \mathbf{P}(A, \surd)$ and $\mathbf{R}(A, P' \; ; \; Q) = \mathbf{P}(A, P')$

$$\frac{P \xrightarrow{excp_i, g, d} \mathcal{P} \qquad i \in I}{\textsf{try}\{P\}\{\textsf{catch} \; excp_i \; \{P_i\}\}_{i \in I} \xrightarrow{\tau, g, d} \mathcal{R}} \text{ with } \mathbf{R}(\mathbf{A}(P_i), P_i) = 1$$

where $\mathbf{A}(P)$ are the (possible) assignments due to initial process invocations in P

Design decisions

- action-guarded probabilistic branching
  - impossible to probabilistically select an action prior to changing state
  - $\Rightarrow$ avoids awkward and complex asynchronous composition
- clocks and distribution sampling
  - essential to obtain an expansion law (as in CCS/CSP)
  - . . . . . . when supporting general (non-exponential) distributions
- patience and impatience
  - to accomplish compositionality for hard- and soft timing
  - a deadline for a patient sync is met when all deadlines are met
    (i.e., it may happen as long as one deadline (= alarm) is still false)
  - a deadline for an impatient sync is met when one participant meets its deadline
    (i.e., it may happen as long as no deadline (= alarm) is met)

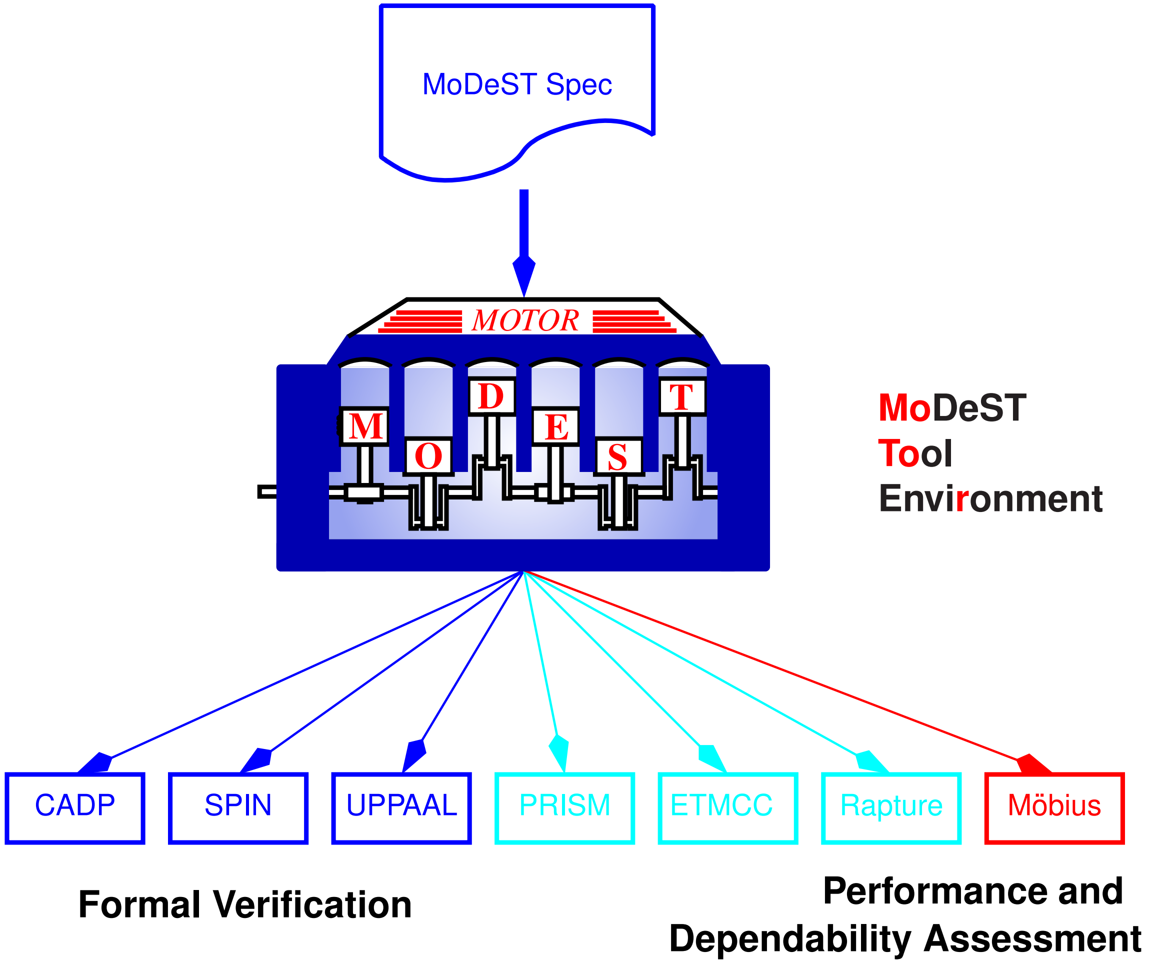
Design decisions

- data model and assignments
  - C-code fragments might be allowed in assignment section
  - . . . but sampling requires some special attention (termination)
- synchronization discipline and value passing
  - synchronizing on shared actions (as in CSP and FSP)
  - more flexible mechanisms could be supported though (as in NT-LOTOS)
  - value passing by means of shared variables (as in UPPAAL)
- exception handling and scoping
  - unhandled exceptions terminate local but not global processes (as in Ada)
- priorities are not included (yet)
  - to keep matters simple . . . . . .

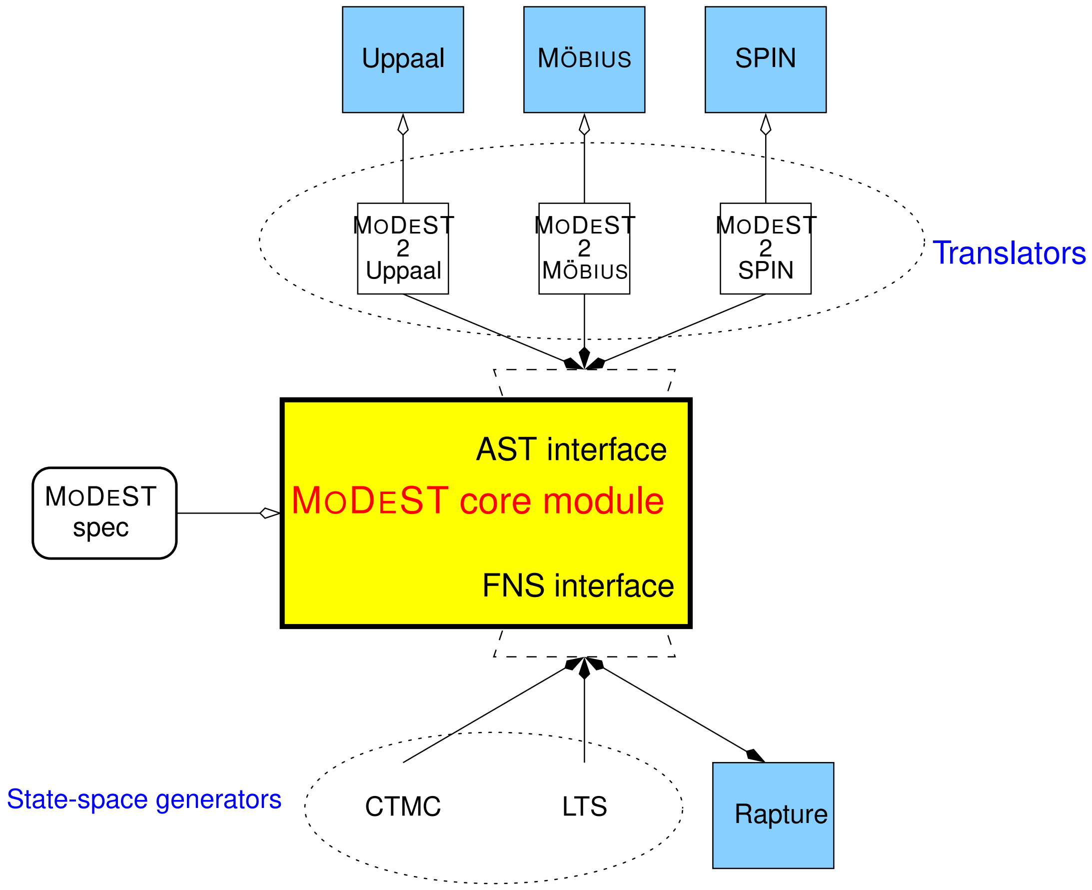
Overview

- Introduction and motivation
- The MODEST modeling language
  - language constructs and examples
  - operational semantics and design rationales
- ⇒ The MOTOR tool environment
  - tool architecture + current status
  - integration with MÖBIUS
- Experience in industrial case studies
  - checking device absence in UPnP (Philips)
  - assessment of schedules for lacquer plants (Axxom AG)
- Epilogue and future work

Aim of the MoTor tool environment

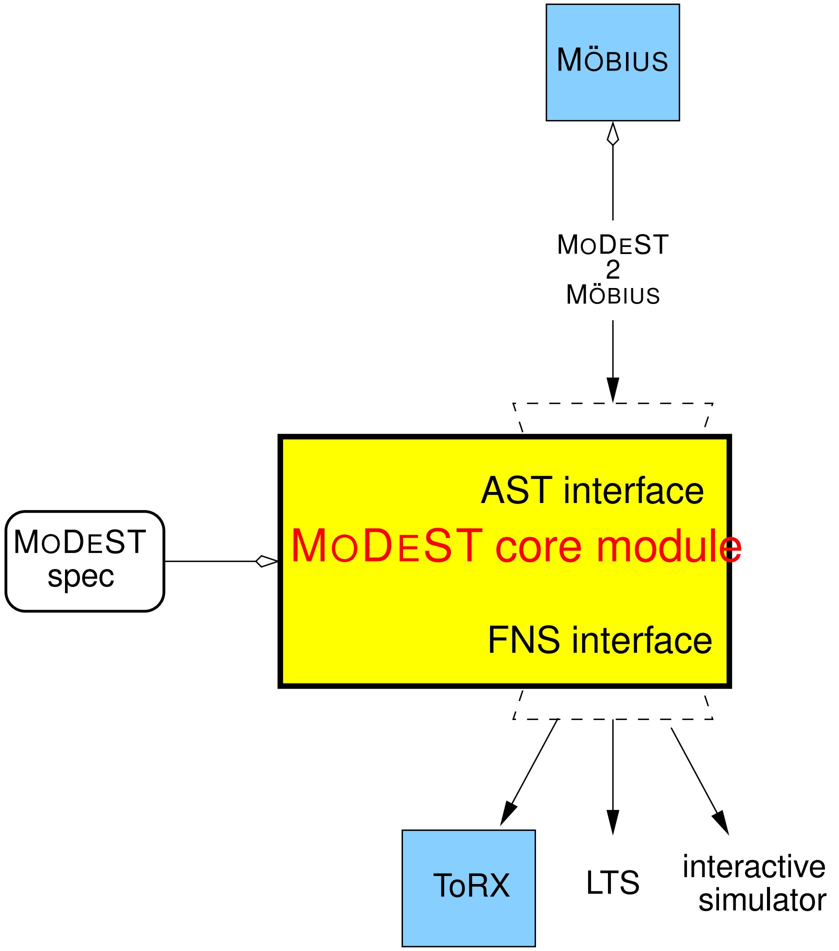


The MoTor tool architecture: our dream



- Core module
  - parser (using ANTLR)
  - operational semantics
- First/Next-State API
  - access global state space
  - current "state"
  - enabled transitions
- AST API
  - access abstract syntax tree
  - useful for tree walkers
  - used by FNS API
- Satellite modules
  - adapting to other tools
  - "native" analysis means

Current state of MoTor implementation



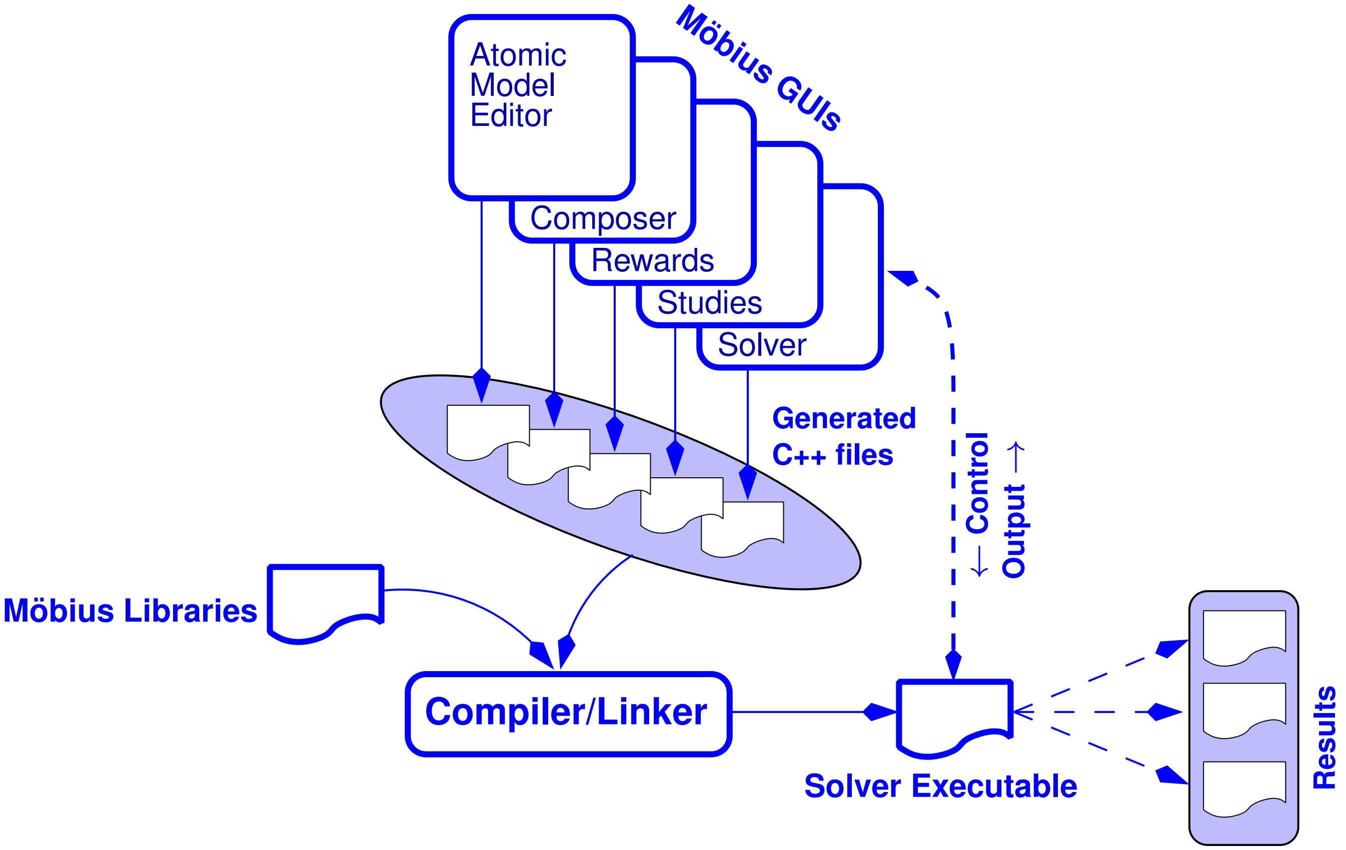
Open FNS interface

- `State* getInitialState()`
  - returns a pointer to the initial global state
- `State* getCurrentState()`
  - returns a pointer to the current global state
- `transition_list& getTransitions(State* state)`
  - returns a list of transitions outgoing from `state`
- `Guard* getGuard(),Guard* getDeadline()`
  - returns guard and deadline of transition
- `bool isNeverEnabled()`
  - returns true if and only if guard transition is constantly false
- `void Fire(Branch& branch, double delay)`
  - executes transition by taking `branch` and advance time by `delay`

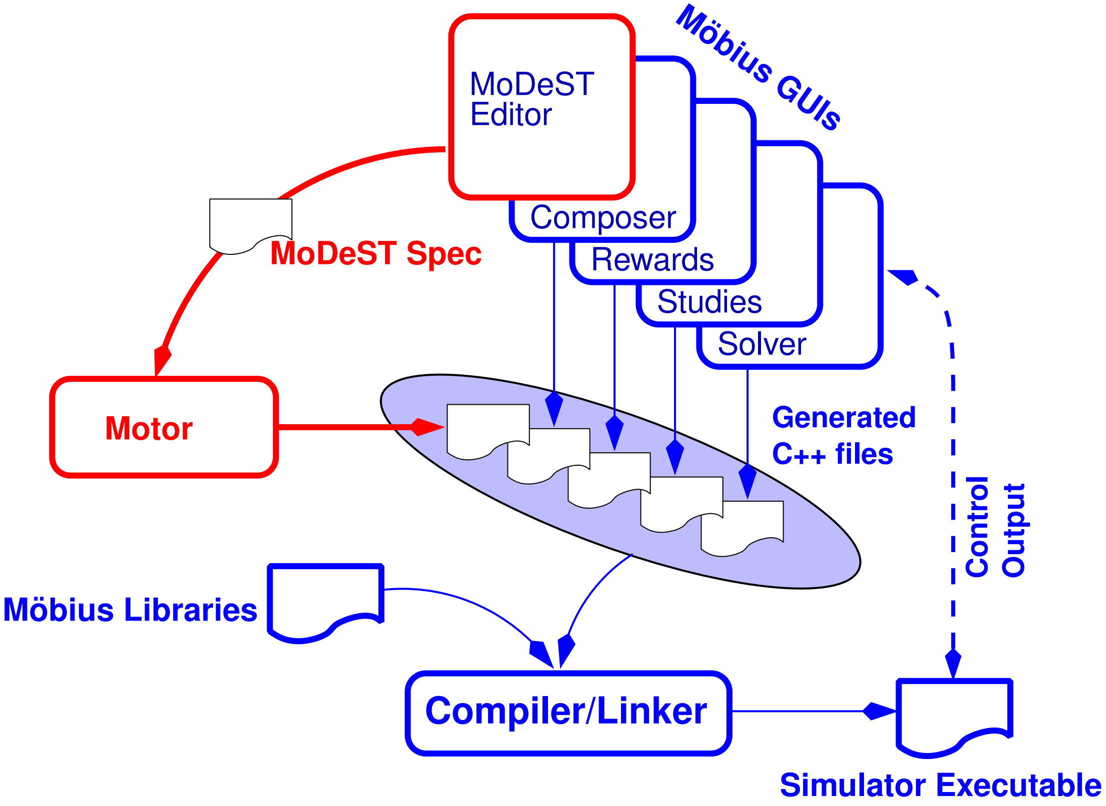
Some MoTor implementation details

- Simple interactive simulator
  - textual user interface to the FNS API
- State-space generator
  - can output `.dot`, `.aut` files of transition system
  - and compact `.bcg` format for the CADP toolset
  - useful for verification and visualization facilities of CADP
  - and for on-the-fly test generation (using ToRX)
- 23,000 lines of C++-code (compiled with `g++`)
  - platforms supported: `unix` and `linux`
- Command-line interface, input: plain ascii MODEST model

Möbius tool architecture



Integrating MoToR into the Möbius framework



this month downloadable from **fmt.cs.utwente.ml/tools/motor**

Overview

- Introduction and motivation
- The MODEST modeling language
  - language constructs and examples
  - operational semantics and design rationales
- The MOTOR tool environment
  - tool architecture + current status
  - integration with MÖBIUS

$\Rightarrow$ Experience in industrial case studies

  - checking device absence in UPnP (Philips)
  - assessment of schedules for lacquer plants (Axxom AG)
- Epilogue and future work

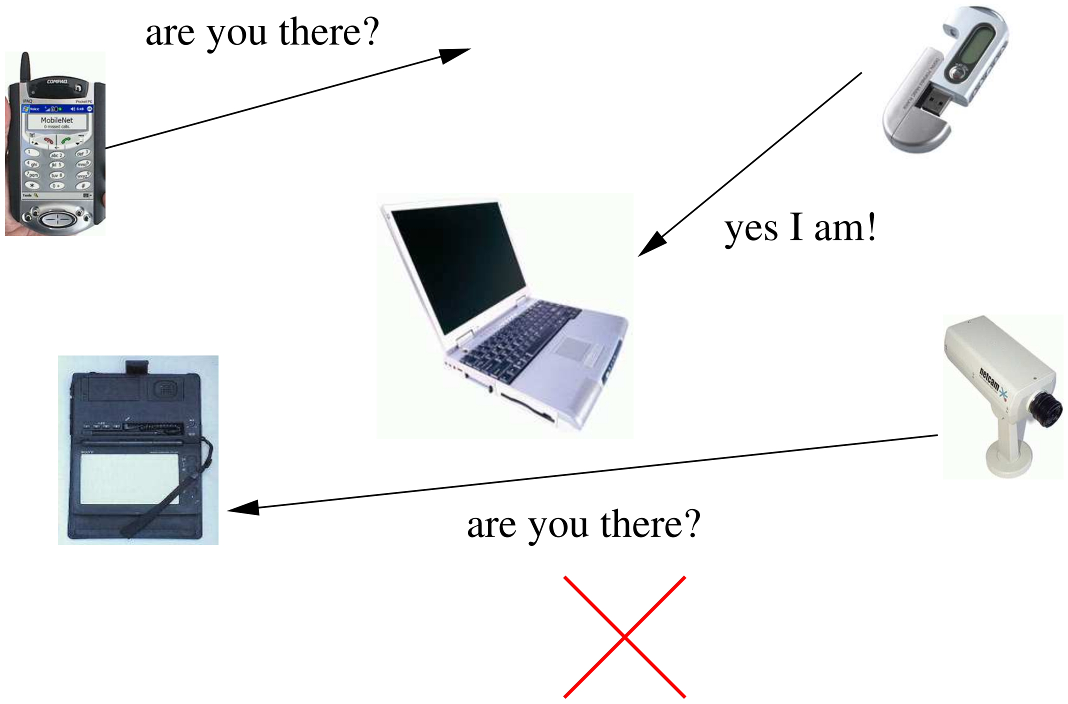
Distributed device-absence checking

- dynamically changing distributed environment (plug-and-play network)
- maintain up-to-date information about the presence of nodes
  - detect the absence of nodes fast while avoiding to overload nodes
- related to
  - *failure detection* protocols: identify whether one or more nodes stop executing
  - *monitoring* protocols: data gathering (such as availability information)
- protocol must be self-organizing
  - continues to operate properly without manual intervention under the joining and (un)intentional leaves of nodes

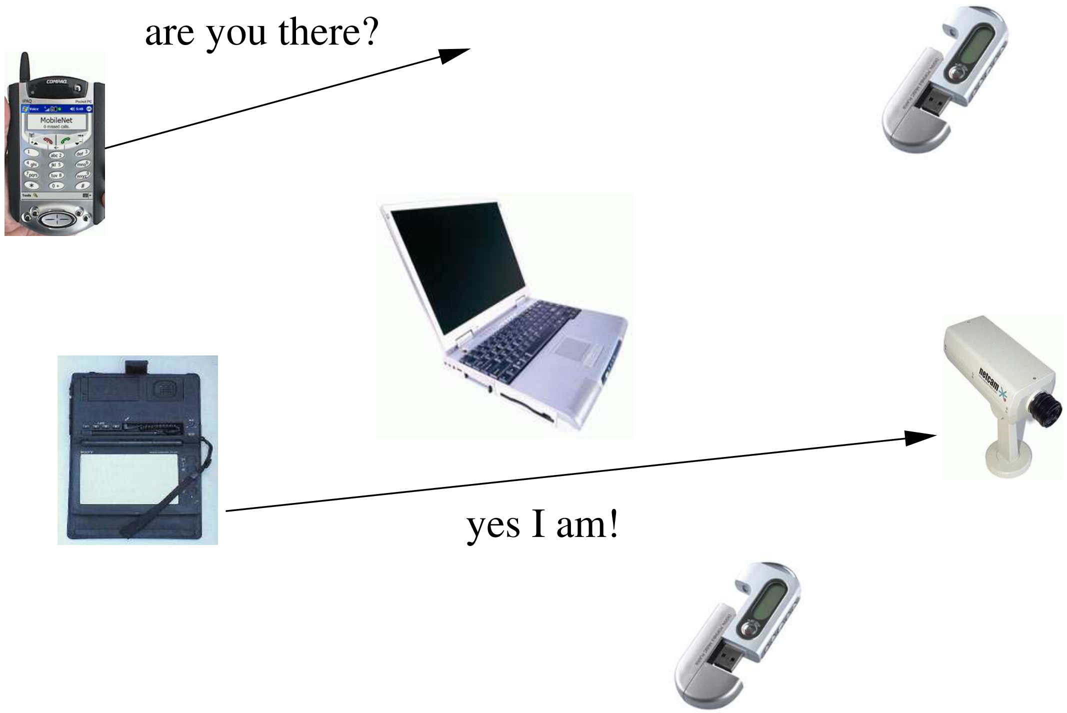
Distributed device-absence checking

Distributed device-absence checking

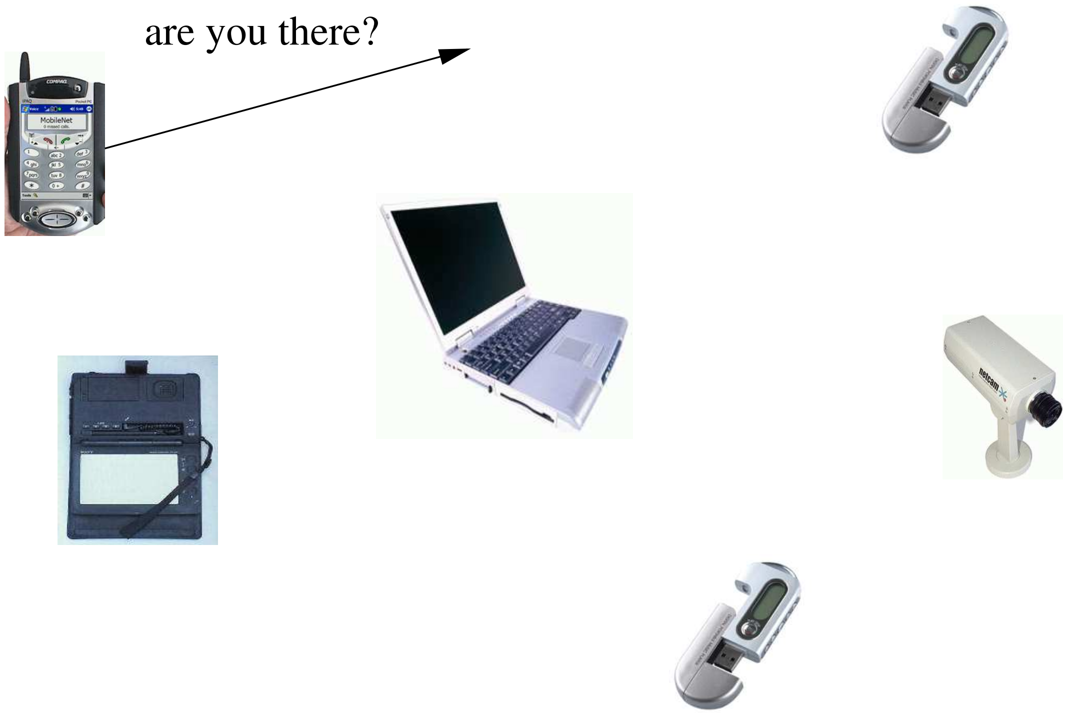
Distributed device-absence checking



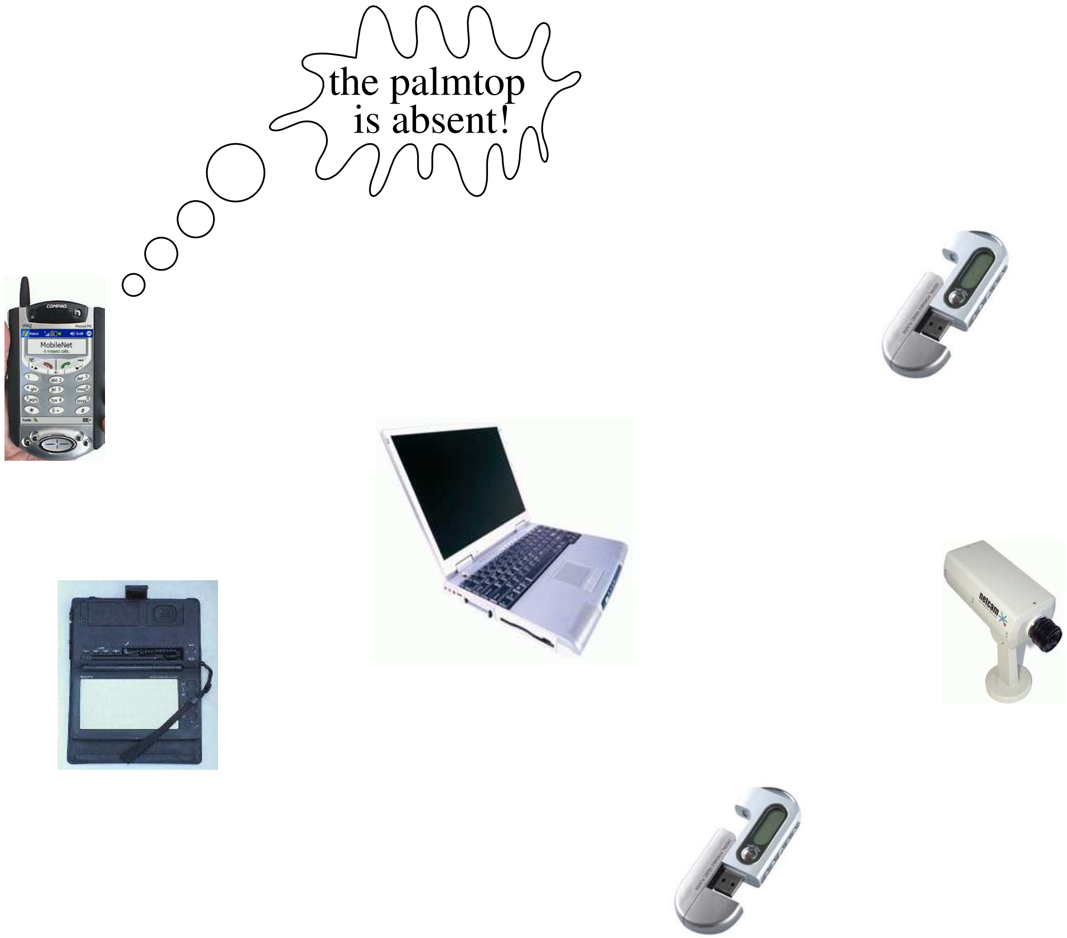
Distributed device-absence checking



Distributed device-absence checking



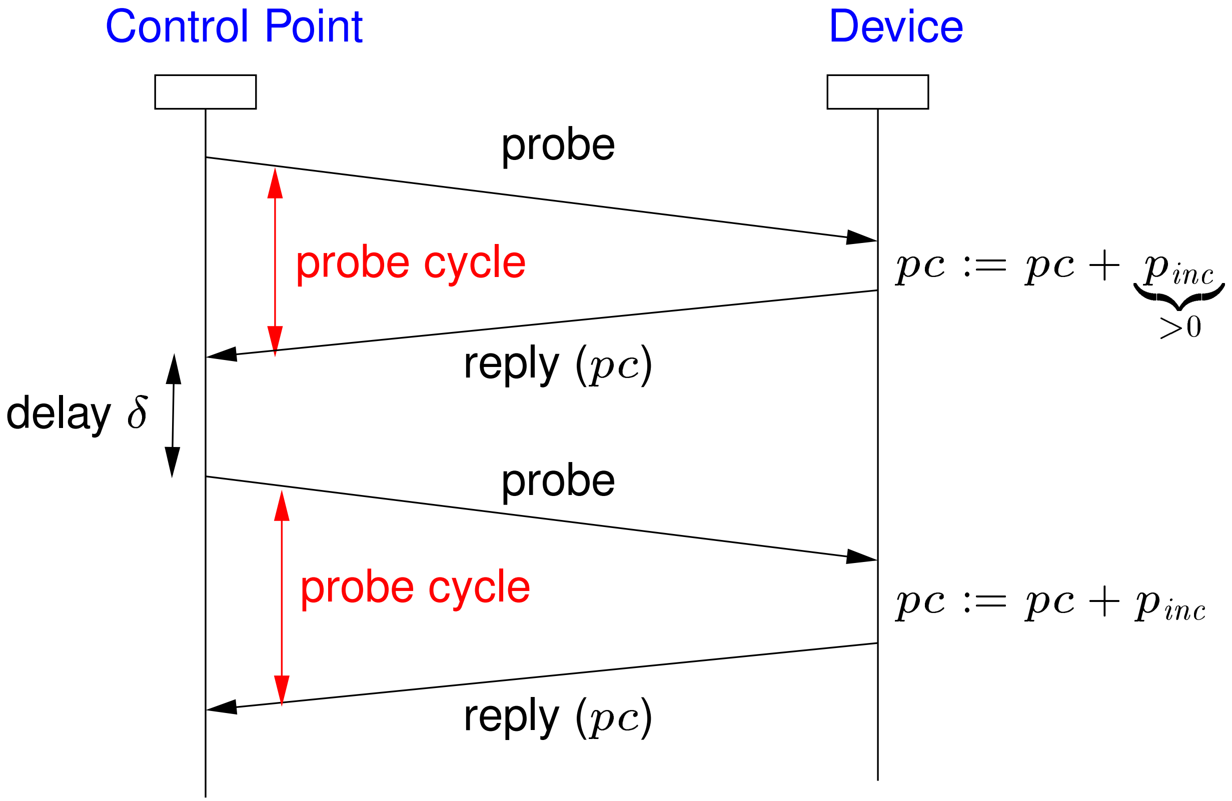
Distributed device-absence checking



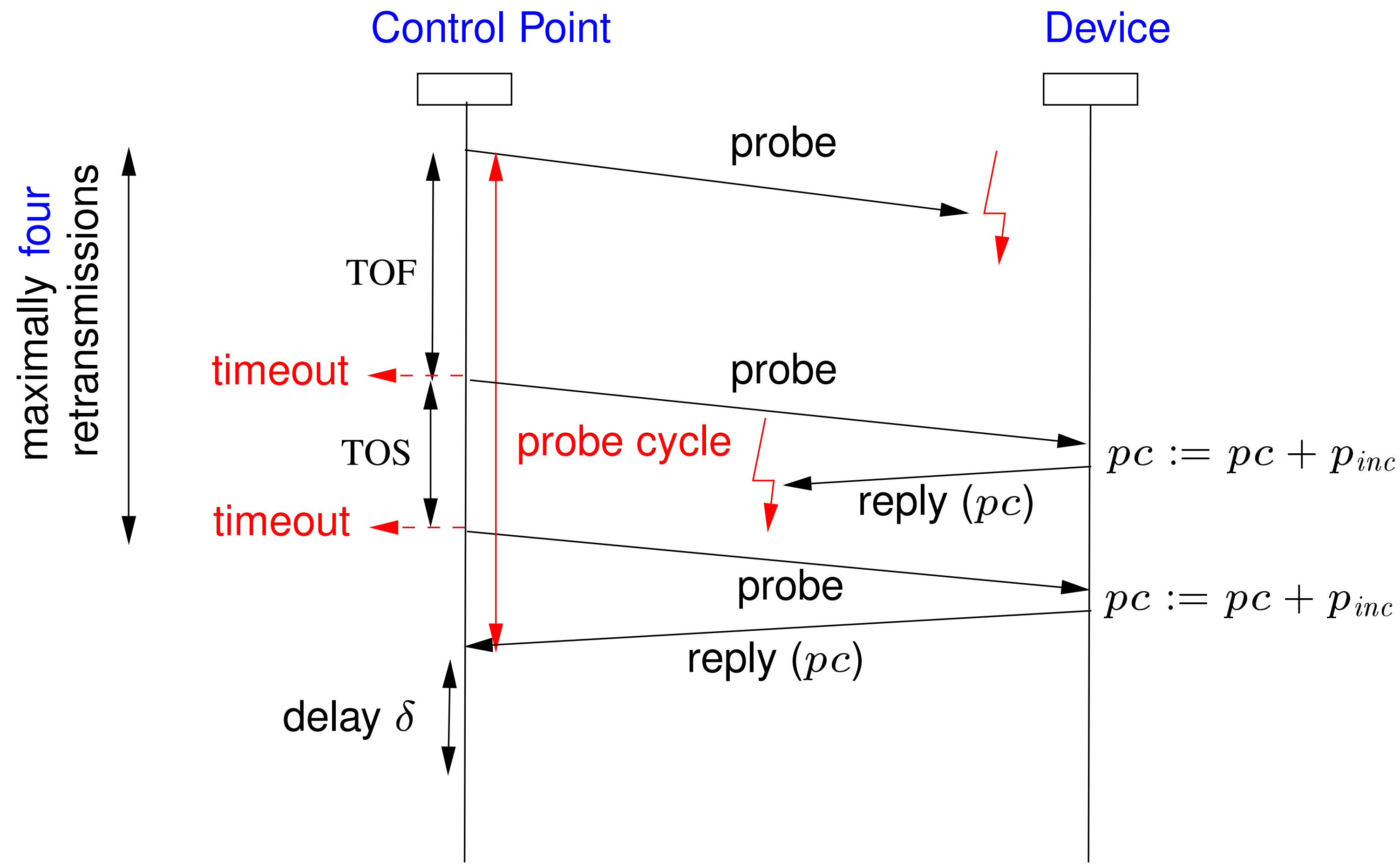
Case study: The probe protocol

- extension of the UPnP service discovery protocol
  - protocol developed by Philips Research Labs
  - can also be used as extension of SLP, Rendezvous and Jini
- two types of nodes: *devices* and *control points* (CPs)
  - CPs are dynamically organized in an overlay network
  - as devices return the ids of the last two distinct CPs that probed it
  - used to rapidly inform CPs about the leave of a device
- a CP continuously probes a device to check whether it is present
  - essence is to automatically adapt the probing frequency in case a device tends to get over- or underloaded

Elementary protocol



Bounded number of retransmissions



Probe loads and optimal probe delay

- L_{ideal} is the fixed ideal probe load of a device
  - known to the device (of course) and all CPs
- L_{nom} is the fixed (private) nominal probe load of a device
  - if device is too slow to handle the ideal probe load, it slows down the CPs
  - . . . by changing the probe increase: $p_{inc} = \left\lceil \frac{L_{ideal}}{L_{nom}} \right\rceil$

  $\Rightarrow$ CPs consider the device to be more busy and adapt their probing frequency
- optimal probe frequency for k CPs is L_{nom}/k
  - . . . . . . but neither k nor L_{nom} are known to a CP

  $\Rightarrow$ CPs should adapt their probing frequency

Adapting the probing frequency

- experienced probe load by a CP: $L_{exp} = (pc' - pc)/(t' - t)$
  - previous reply with pingcount pc received @ time t
  - current reply with pingcount pc' received @ time t'
- CP adapts inter-probe-cycle delay δ
  - to keep L_{exp} "close to" L_{ideal}: $\frac{1}{\beta} \cdot L_{ideal} \leqslant L_{exp} \leqslant \beta \cdot L_{ideal}$ for $\beta > 1$
  - while maintaining: $\delta_{min} \leqslant \delta \leqslant \delta_{max}$
- self-adaptive mechanism for a CP to determine new delay δ':
  - $\delta' = \begin{cases} \min\left(\alpha_{inc} \cdot \delta, \delta_{max}\right) & \text{if } L_{exp} > \beta \cdot L_{ideal} \\ \max\left(\frac{1}{\alpha_{dec}} \cdot \delta, \delta_{min}\right) & \text{if } L_{exp} < \frac{1}{\beta} \cdot L_{ideal} \\ \delta & \text{otherwise} \end{cases}$
  - where $\alpha_{inc} > 1$ and $\alpha_{dec} > 1$

Network model

MoDeST model of device

```
process Device (int id) {
  const int p_inc = (int)(#eval(L_IDEAL / (L_NOMINAL)));
  clock x;                     // timer for reply time
  int cp, pc = 0;              // probe counter
  float rpldel;                // reply delay

   do { :: when (dev_in.lck == 1)    // probe in input buffer
            handle_probe {= pc += p_inc,
                            rpldel = min + (max - min)*Uniform(0,1),
                            cp = dev_in.src,
                            x = 0 =};
          when ( x >= rpldel && dev_out.lck == 0)
            send_reply {= dev_out.lck = 1, dev_out.src = id,
                          dev_out.dst = cp, dev_out.pc = pc,
                          dev_in.lck = 0 =}

          }
      }
}
```

Sketch of CP behaviour in MoDeST

```
process CP (int id ) {
  do {
     :: probe {= i = 1, x = 0, to = TOF =};
     do { :: alt {       // wait for reply or timeout
             :: when ( (x >= to) && (i < 4) )
                  probe {= i += 1, x = 0, to = TOS =};
             :: when ( (x < to) && in_buffer )
                  handle_reply {= prd = y, pl = (CP[id].pc - lastpc)/prd,
                                  y = 0, lastpc = CP[id], i = 0 =};
                alt {
                 :: when (pl > beta * L_IDEAL) d = min(d * a_inc, d_max);
                 :: when (pl < (1/beta) * L_IDEAL) d = max(d * 1/a_dec, d_min);
                 :: else tau
                }
                when (x >= d) break                  // wait and re-probe
             :: when ( (x >= to) && (i == 4) )
                  dev_abs {= i = 0, lastpc = 0 =};
                when (x >= d) break                  // wait and re-probe
]       }      }
```

Measures of interest

- Do the delays δ stabilize?
- Do all CPs probe at more or less the same frequency?
- How to choose the parameters β, δ_{min}, δ_{max}, etc.?
- How far is the experienced probe load off the actual probe load?
- How fast is the absence of a device detected?
- . . . . . .

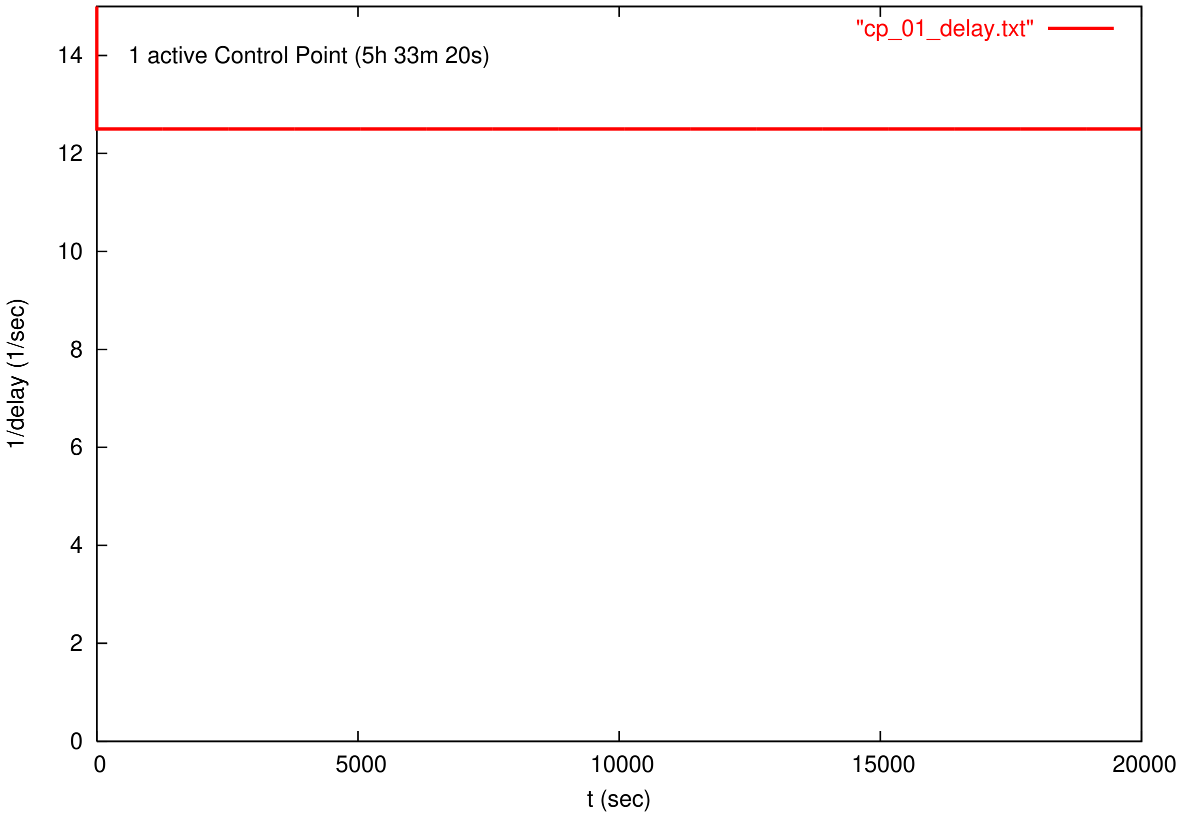
Steady-state simulation

- Batch-mean simulation using MOTOR and Möbius
- Parameters:
  - $\alpha_{inc} = 2$, $\alpha_{dec} = \frac{3}{2}$, $\beta = \frac{3}{2}$
  - $L_{ideal} = 10^6$, $L_{nom} = 10$, $k = 20$
  - network delay = probabilistic alternative of 3 uniform distributions
  - network buffer size is extremely large (20,000)
- Some results:

  $\Rightarrow$ focused on probe frequency $\frac{1}{\delta}$ of the CPs
  - mean freqs of almost all CPs is ≈ 10, but some (two) 0.4!
  - ...... the optimal probe frequency equals $k/L_{nom} = 0.5$
  - average buffer length is small (≈ 0.004)
  - high variance in delays (even one device with 13.5)

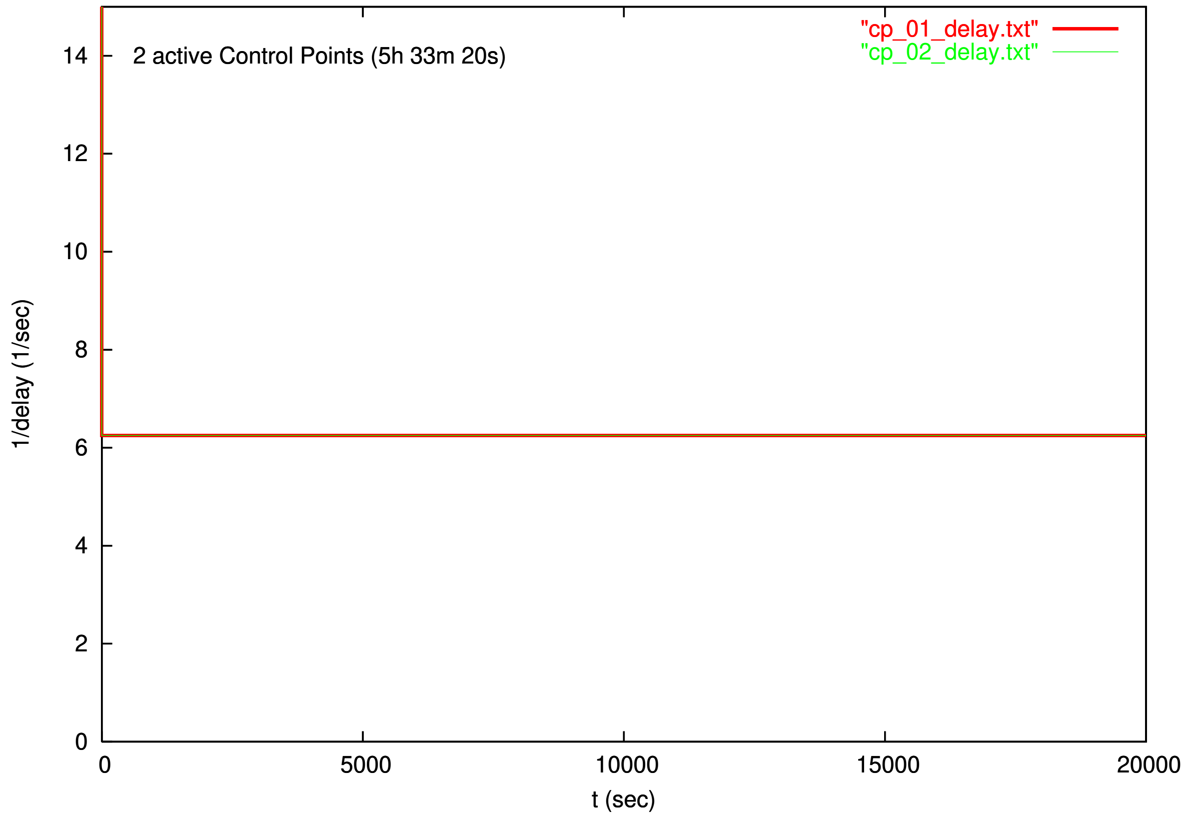
let's see in some more detail what happens

Probing frequency for just one CP



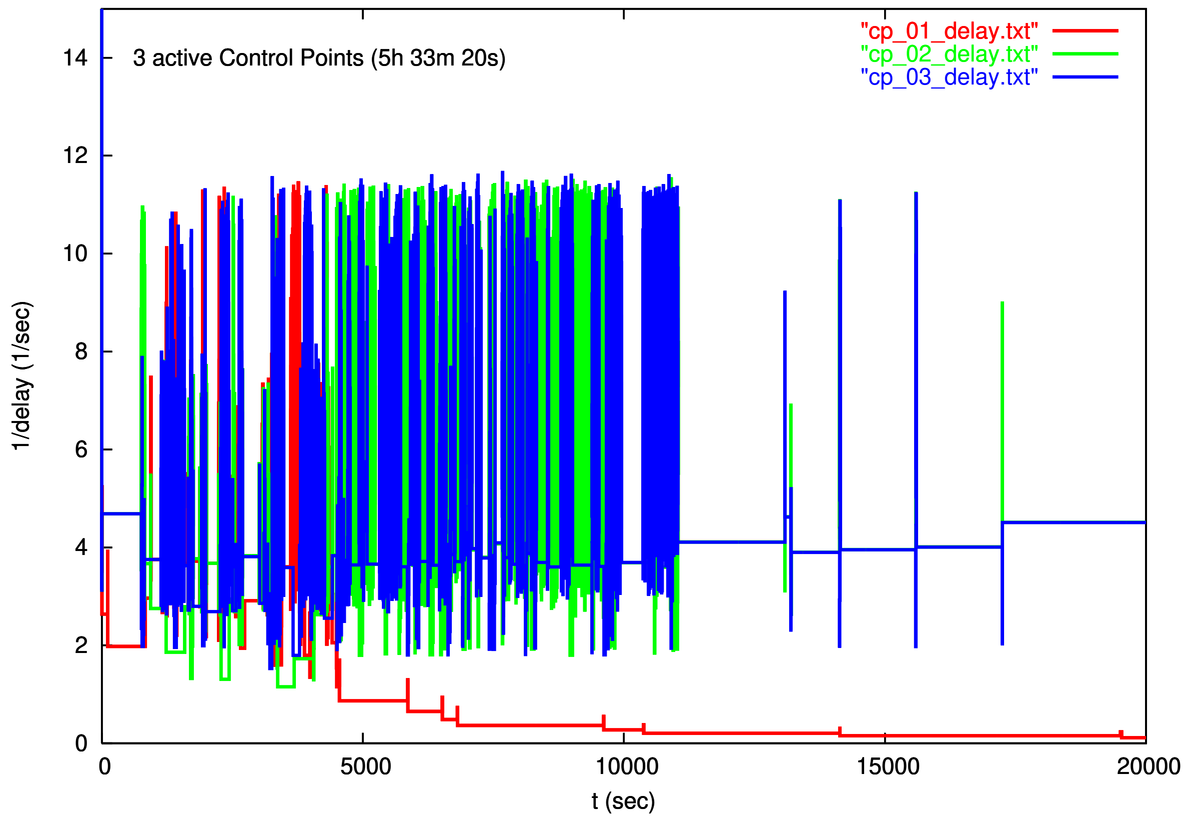
frequency stabilizes, but exceeds the ideal one (=10)

Probing frequency for just two CPs



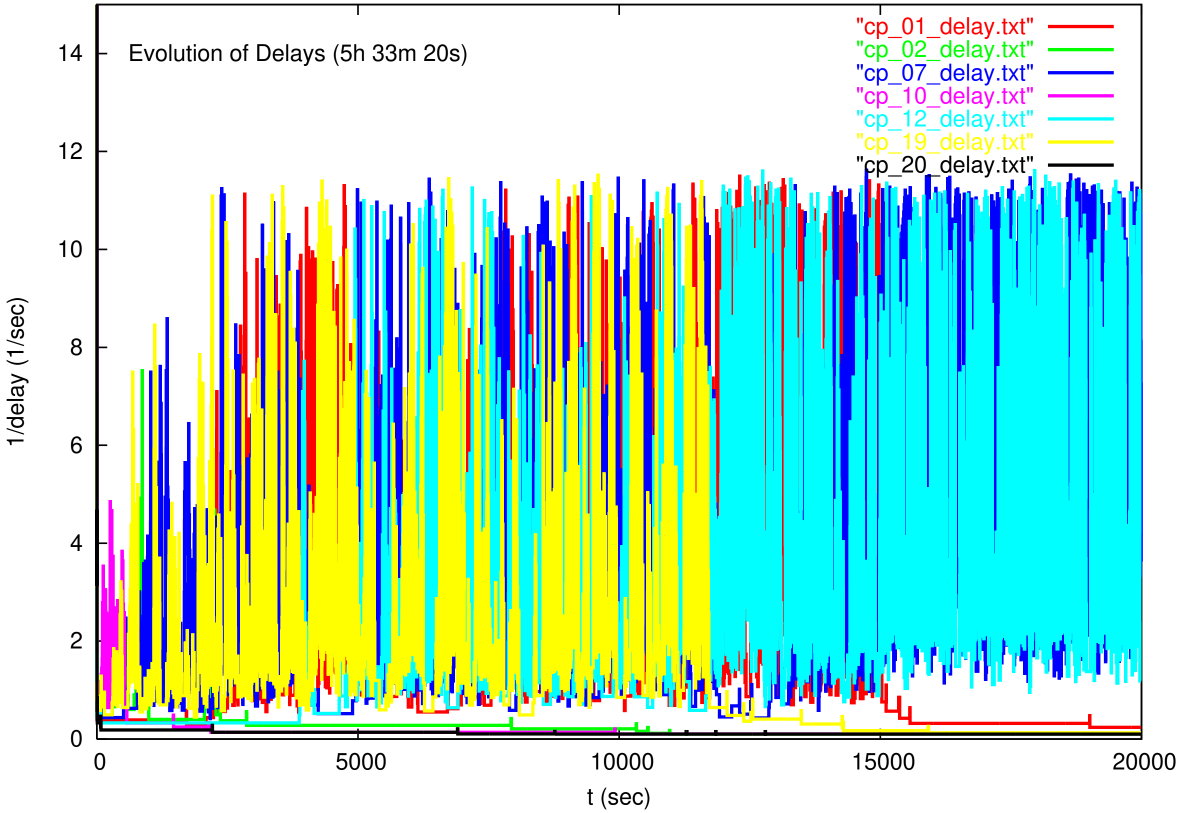
frequencies stabilize, but exceed the ideal one (=5)

Three always has been a strange number



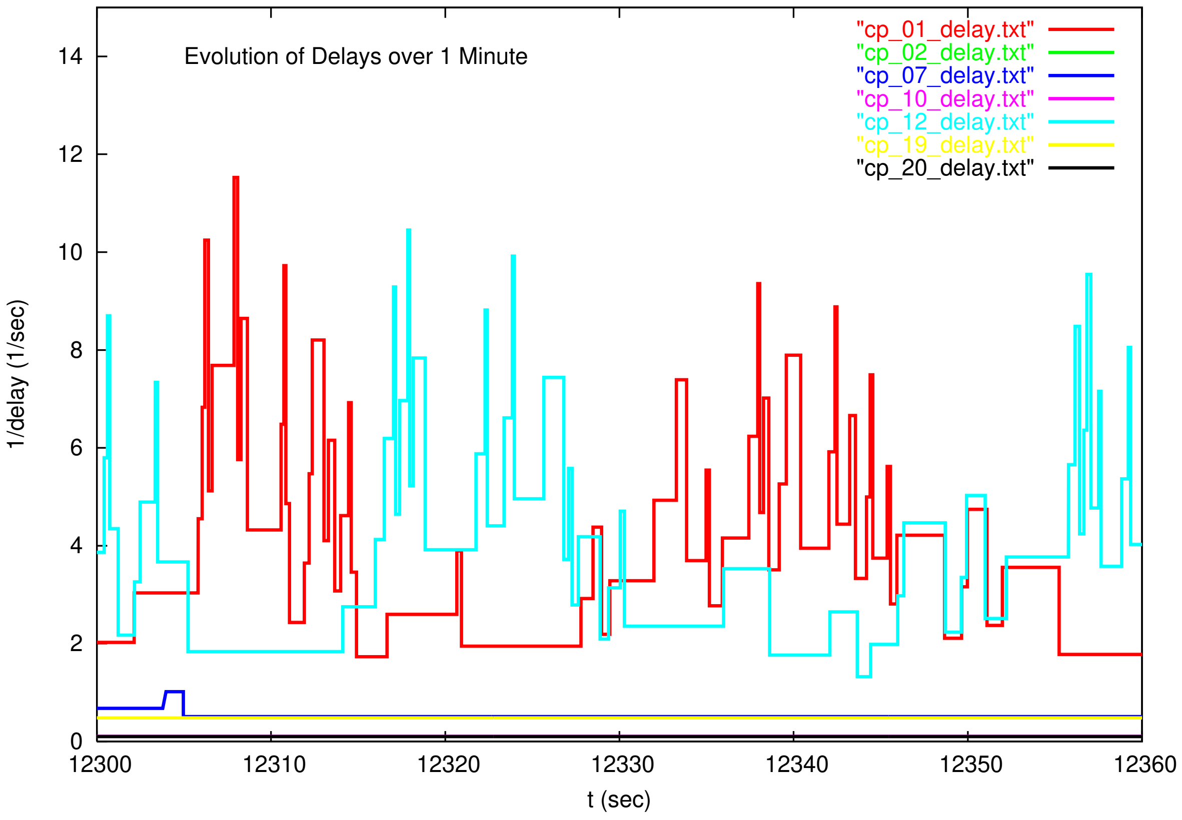
individual starvation?

When 20 CPs are around



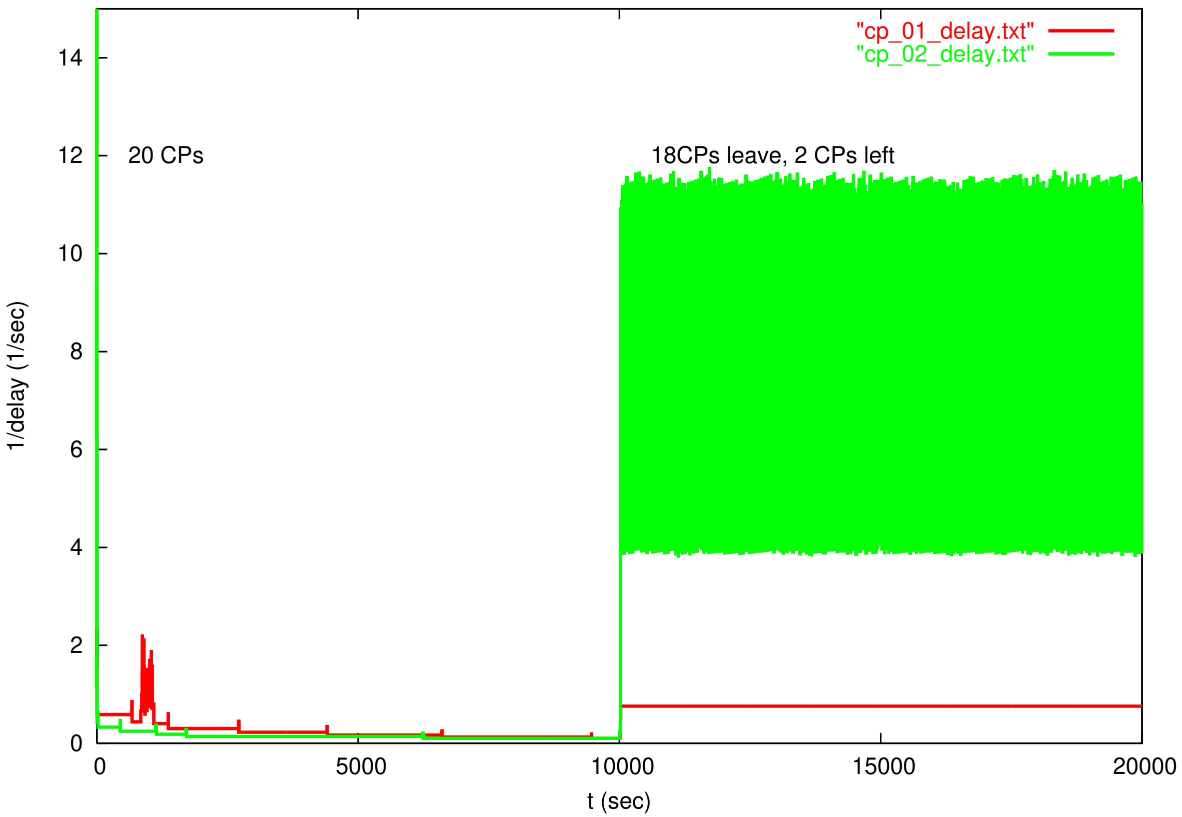
individual starvation is not a seldom phenomenon!

Zooming in



Evolution of Delays over 1 Minute
"cp_01_delay.txt"
"cp_02_delay.txt"
"cp_07_delay.txt"
"cp_10_delay.txt"
"cp_12_delay.txt"
"cp_19_delay.txt"
"cp_20_delay.txt"
1/delay (1/sec)
0
2
4
6
8
10
12
14
12300
12310
12320
12330
12340
12350
12360
t (sec)

When 20 CPs are there and 18 suddenly leave



even for two CPs there is an unfair balance

Some considerations

- the experienced ping load of a CP may be misleading
  - many probing CPs or a few probing many?
  - $\neq$ actual probe load at the device
  - why do devices not communicate their load to CPs?
- (at least) two undesired phenomena:
  - variance in probing frequencies is too large
  - CPs can be completely starving
  - . . . . . . could CPs be "synchronised" in some sense?
- Useful indication about parameter values obtained

$\Rightarrow$ Philips has agreed on a major re-design (under patent)

Overview

- Introduction and motivation
- The MoDeST modeling language
  - language constructs and examples
  - operational semantics and design rationales
- The Motor tool environment
  - tool architecture + current status
  - integration with Möbius

⇒ Experience in industrial case studies
  - checking device absence in UPnP (Philips)
  - assessment of schedules for lacquer plants (Axxom AG)
- Epilogue and future work

Lacquer production schedules

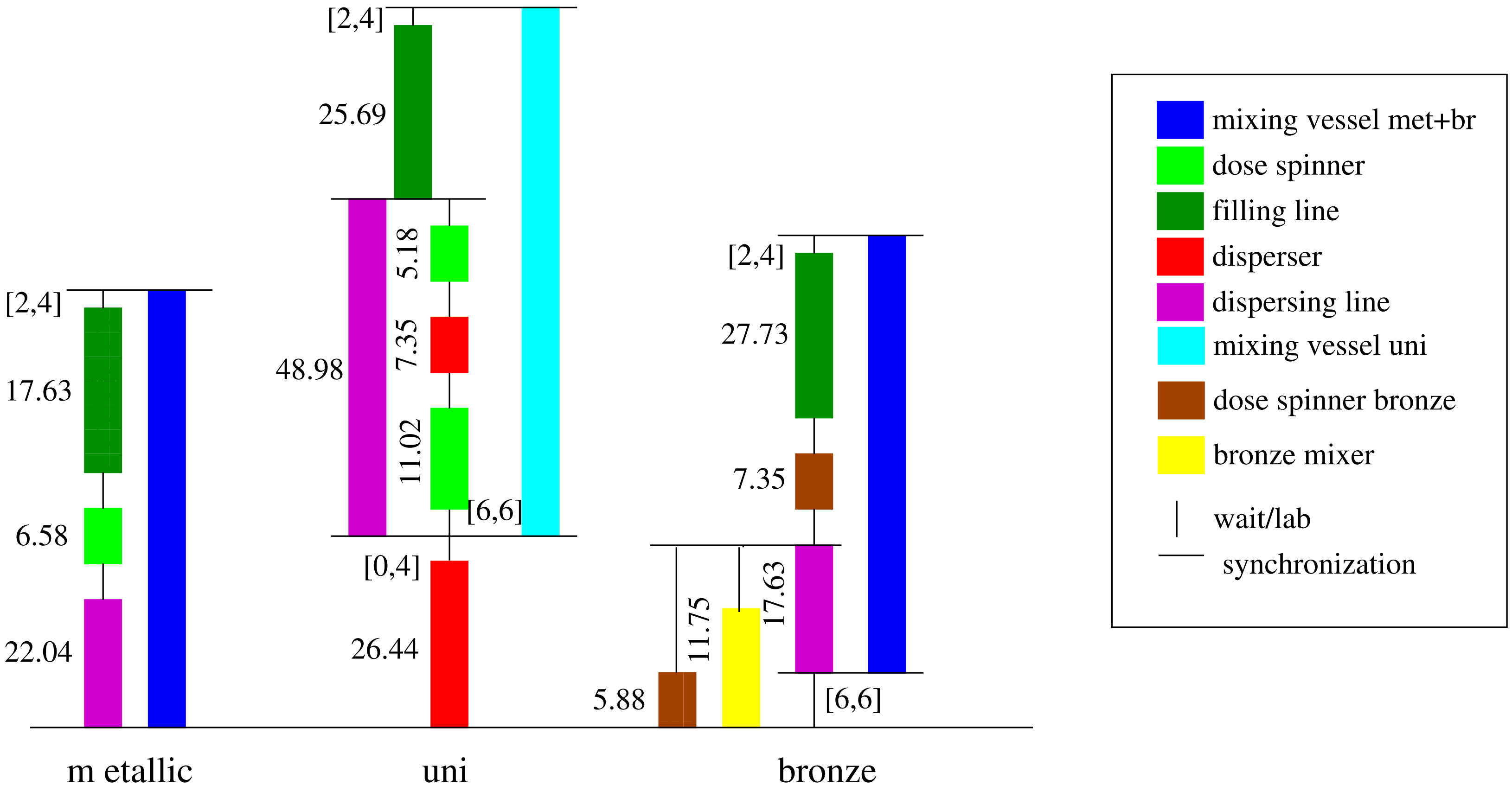
- Scheduling problems
  - hard deadlines with limited number of resources
  - solution using classical scheduling theory (plus heuristics)
- *En vogue:* model checking for scheduling synthesis
  - model consists of all possible (ordering of) production steps
  - scheduling infeasibility is expressed as temporal property

    "The production can<u>not</u> be finished by friday afternoon"
  - check property using heuristic to guide state-space search
  - counter-example is a concrete schedule
- Schedules for lacquer production (Axxom AG)
  - assess random aspects such as failures, repairs, cleaning etc.
  - starting from 20 synthesized feasible schedules
  - no job-shop scheduling problem: multiple resources + timing between tasks

Terminology

- Products: uni, metallic, and bronze lacquers
- Resources (14)
  - 2 mixing vessels for uni-lacquers, 3 mixing vessels for metallic and bronze lacquers, 2 filling stations, 1 disperser, 2 dose spinners, etc.
- Jobs (29)
  - cannot start before their earliest start time
  - have deadlines at which the production needs to be finished
- Recipes (3) describe:
  - order of production steps
  - the type of resource that is needed for a production step
  - timing behavior, e.g. the processing time with a certain resource,
  - offset times (= delays between production steps)

Example recipes



Performance and availability factor

- Performance factor p
  - average fraction of time the resource is not operational is $1-p$
  - due to unforeseeable circumstances (e.g., breakdowns)
  - no info on frequency of resource breaks, the duration of down-times or repair-times, etc.
- Availability factor a
  - the fraction of (operational) time the resource can be used
  - because necessary human support is present
- Performability factor
  - e.g., the fraction of time during which the machine works "full speed" on average

$$a \cdot p = \frac{\text{mean up time (MUT)}}{\text{MUT + mean time to repair (MTTR)}}$$

Schedule synthesis by model checking

- Model
  - recipe = timed automaton with parameterised earliest start time and deadline
  - clocks measure the durations of the processing steps and offset times
  - parallel composition of the 29 instantiations of the recipes
  - resource = shared integer variable
- Two types of processing times
  - ignore the performance and availability factors (be optimistic)
  - extend the processing time (be conservative):

    if a job needs T time units, schedule it for $T/(a \cdot p)$ time units
- Find schedule by random depth-first search + heuristics
  - e.g, if a resource is available, take it immediately

Random breaks and repairs

- What is the probability that a job will meet its deadline?
  - random delays are compensated by deterministic delays $\Rightarrow$ miss deadline?
  - this risk is (intuitively) even higher for schedules without overapproximations

  $\Rightarrow$ "realistic" machine models with random break and repair times

- $a \cdot p = \mathrm{MUT}/(\mathrm{MUT} + \mathrm{MTTR})$ is the same for a resource
  - on average breaks every half a day and needs half a day to be repaired
  - on average breaks every week and needs another week to be repaired

- A pace parameter that indicates frequenqy of occurrence of events

$$\frac{1}{\textit{pace}} = \mathrm{MUT} + \mathrm{MTTR}$$

  - only mean values for MUT and MTTR are available

  $\Rightarrow$ use exponential distributions with these means

Resource model

```
process ABF1_machine() {                    // filling line no. 1
  clock w, y;                               // processing time, repair time
  float br, r, work, dl;
  br = EXP(pace/0.86*0.75);                 // next breaking time

  do {
  :: when (ABF1_work > 0) start_ABF1         // start processing
       {= work = ABF1_work, ABF1_work = 0,    // work = needed processing time
          dl = ABF1_dl, w = 0 =};             // set deadline, restart timer
         do { :: when (cjobs >= dl)
                   error_ABF1 {= ABF1_done = 2, br -= w =};
                 break
              :: when (br >= work && w >= work)  // processing step finished
                   done_ABF1 {= ABF1_done = 1, br -= w =};
                 break
              :: when (br < work && w >= br)    // break before finish
                   break_ABF1 {= work -= br, y = 0, r = EXP(pace/(1-0.86*0.75)) =};
                 when (y == r)                // wait for repair
                   repair_ABF1 {= w = 0, br = EXP(pace/0.86*0.75) =}
}    }      }
```

Jobs and locks

```
alt{
 :: when(ABF1_lock == 0) // wait to lock resource
      acq_ABF1_lock {= ABF1_lock = 1, ABF1_dl = dl-2, ABF1_work = 23 =};
    when(ABF1_done > 0) // wait until task is done
      rel_ABF1_lock {= ABF1_done = 0, ABF1_lock = 0 =}
 :: when(ABF2_lock == 0)
      acq_ABF2_lock {= ABF2_lock = 1, ABF2_dl = dl-2, ABF2_work = 23 =};
    when(ABF2_done > 0)
     rel_ABF2_lock {= ABF2_done = 0, ABF2_lock = 0 =}
}
```

- start a task immediately after the previous task has finished

  ⇒ we can be ahead of the synthesized schedule

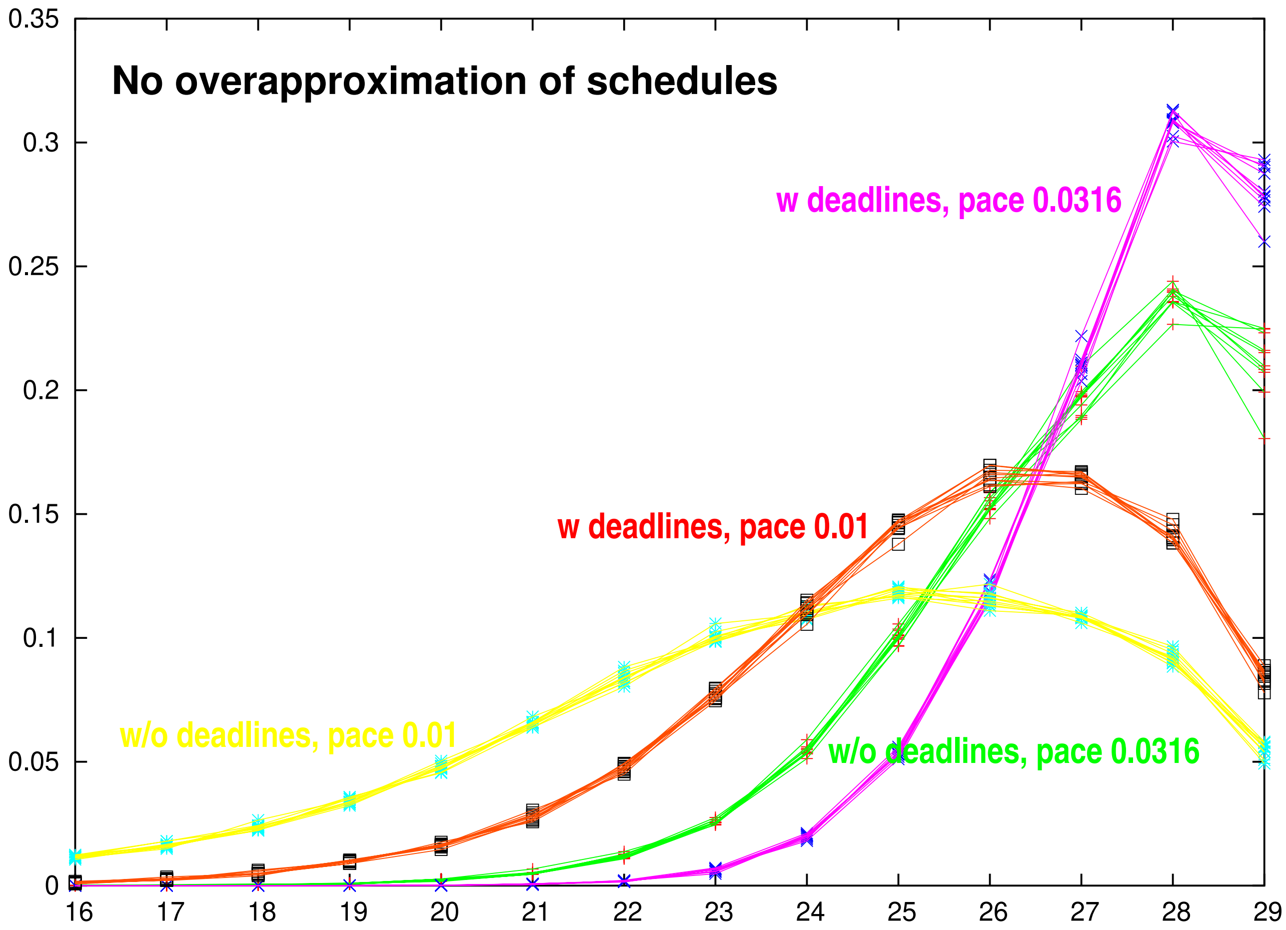
- nondeterminism when choosing among multiple (identical) machines
  - adopt Bernoulli's principle of insufficient reason:

    "all events have the same likelihood unless there is evidence for the contrary"

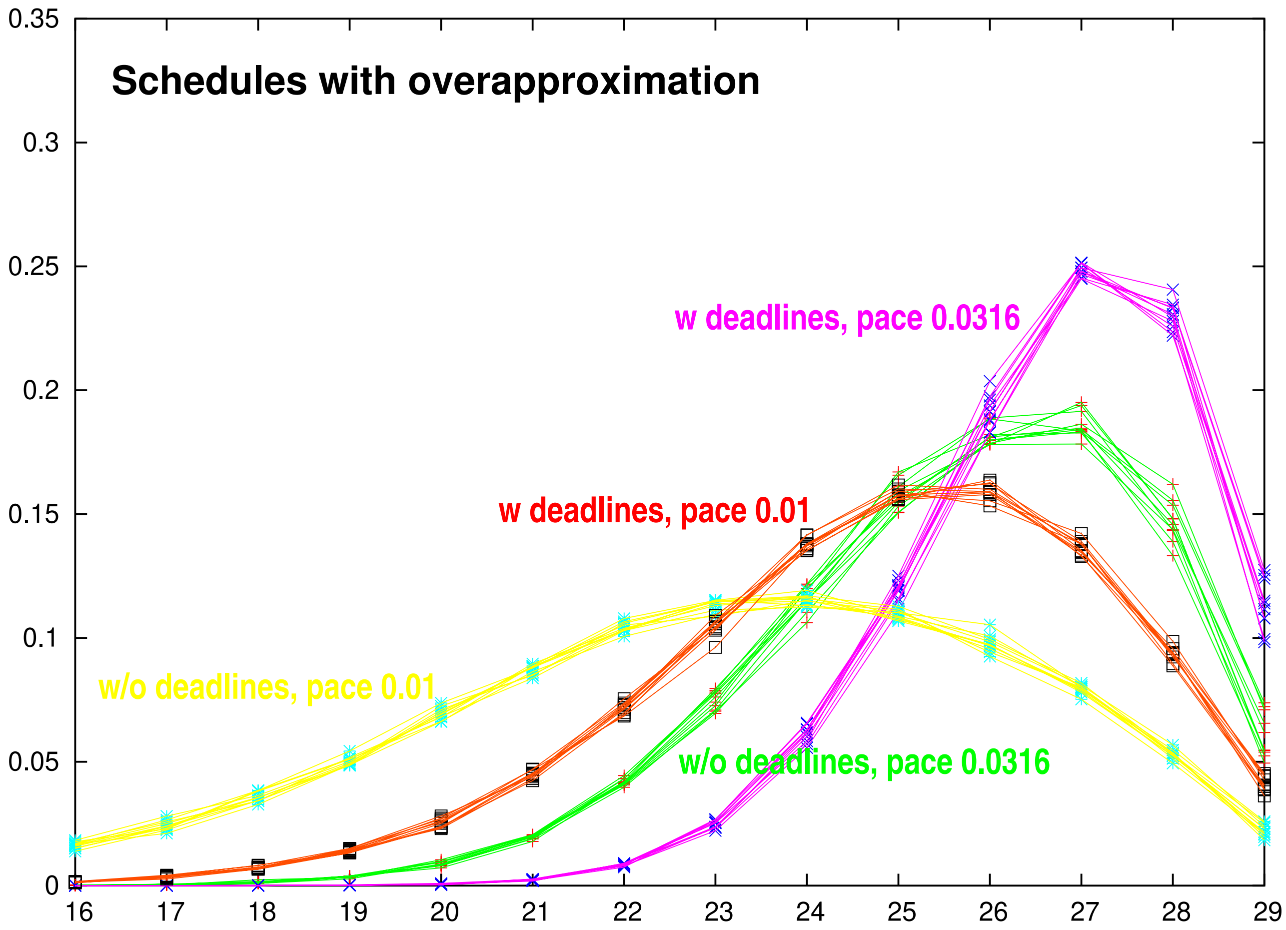
Experiments

- Goal:
  - evaluate the degree to which the schedules can tolerate the delays due to the breaks and repairs of the machines
- Batch mean simulation focused on for each schedule:
  - the number of jobs meeting the deadline? (0–29)
  - the success rate for each individual job
  - total time that each particular machine is working, repaired, and idle
  - . . . using 80 experiments, each executed for 20000 times
- Deadline behaviours
  - once a product fails to meet deadline, give up (w deadline)
  - continue a schedule even though deadlines have already passed (w/o deadline)

Success probability (I)



Success probability (II)



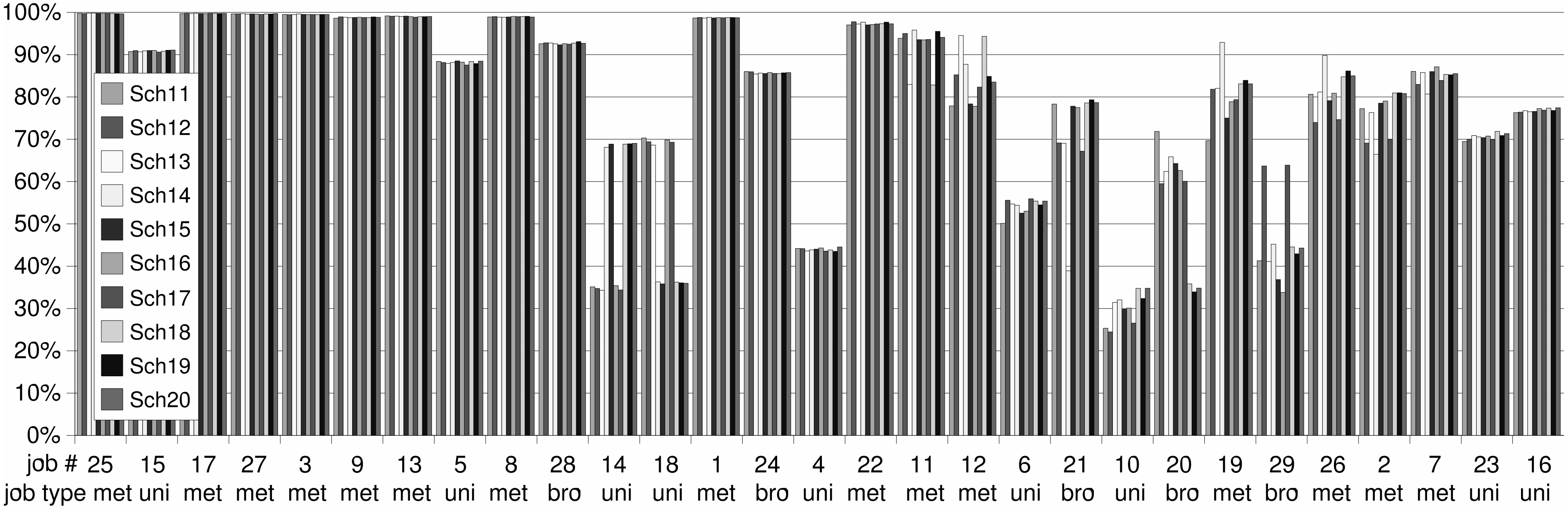
Observations

- Overapproximated schedules underperform the normal ones
  - some jobs are scheduled to start late, as earlier jobs are scheduled longer

  $\Rightarrow$ "later" jobs have less time remaining until the deadline
- It is less probable to finish jobs when ignoring deadlines
  - "stuck" jobs occupy resources, without improving their chance to reach the deadline
- Increasing the pace raises number of completed jobs
  - more detailed timing info on break and repair times

  $\Rightarrow$ less unused fragments of time intervals

Probability to meet deadline



overapproximated schedules (ordered by est), pace=0.01 and w/o deadlines

Overview

- Introduction and motivation
- The MoDeST modeling language
  - language constructs and examples
  - operational semantics and design rationales
- The Motor tool environment
  - tool architecture + current status
  - integration with Möbius
- Experience in industrial case studies
  - checking device absence in UPnP (Philips)
  - assessment of schedules for lacquer plants (Axxom AG)

$\Rightarrow$ Epilogue and future work

Epilogue

- Summarizing:
  - MoDeST is aimed at a single-specification multi-solution approach
  - a single, coherent specification used for various analysis phases
  - main ingredients: compositionality and formal semantics
  - prototype tool realised + linking to Möbius solvers
  - first industrial case studies conducted
- Future work:
  - integration in UML design flow
  - link to model checking $\Rightarrow$ measure specification in temporal logic CSL!
  - extend the MoTor tool environment (Uppaal, ETMCC)
  - safe (compositional) abstractions to simpler model classes

MoTor is downloadable from `fmt.cs.utwente.nl/tools/motor` soon