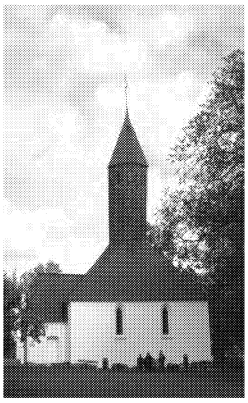
MÄLARDALENS HÖGSKOLA

ESSES 2003
European Summer School on Embedded Systems

Lecture Notes
Part XIX

Real-Time Systems: Synchronous Language Paradigm and Formal Methods in Real-time Systems



European Summer School on
Embedded Systems
Jul 14 - Oct 10 2003
Västerås Sweden

Editors: Ylva Boivie, Hans Hansson, Jane Kim, Sang Lyul Min

Västerås, September 29-October 3, 2003
ISSN 1404-3041
ISRN MDH-MRTC-112/2003-1-SE

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

www.mrtc.mdh.se

TRUST BUT ISOLATE, CHECK AND MONITOR

EUROPEAN SUMMER SUMMER SCHOOL ON EMBEDDED SYSTEMS (ESSES 2003)

October 2, 2003

Vasteras, Sweden

Aloysius K. Mok

The University of Texas at Austin

Today's Schedule

- 09:15 – 10:15 When Can We Trust
- 10:15 – 10:30 Break
- 10:30 – 11:45 Temporal Isolation
- 11:45 – 13:15 Lunch
- 13:15 – 14:30 Verification and Checking
- 14:30 – 14:45 Break
- 14:45 – 16:00 Real-Time Event Monitoring
- 16:00 – 16:30 Discussion

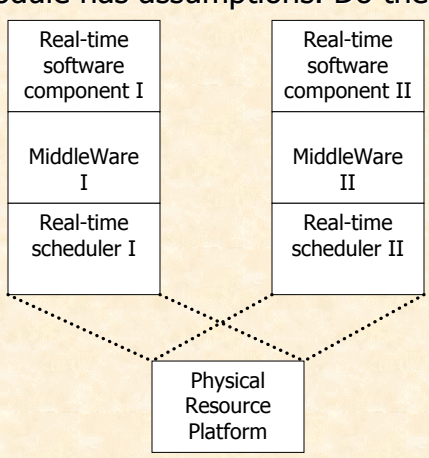
TRUST BUT ISOLATE, CHECK AND MONITOR

ABSTRACT

Formal methods are powerful techniques for guaranteeing that a real-time system meets its design requirements. However, because of economic considerations, engineers must live with at least some software and hardware components that have not been formally verified. Most legacy software and hardware components may not even have formal specification in the first place. This is the state of affairs and probably will remain so for the foreseeable future. How should a real-time system designer cope with this situation? There are of course well known engineering principles such as providing isolation to limit the interaction of components to narrowly defined interfaces, monitoring system behavior for abnormality and pinpointing violations to offending components in real time. In this lecture, we shall look at some of these commonsense engineering principles and the technical issues in their application to ascertain temporal properties of embedded systems.

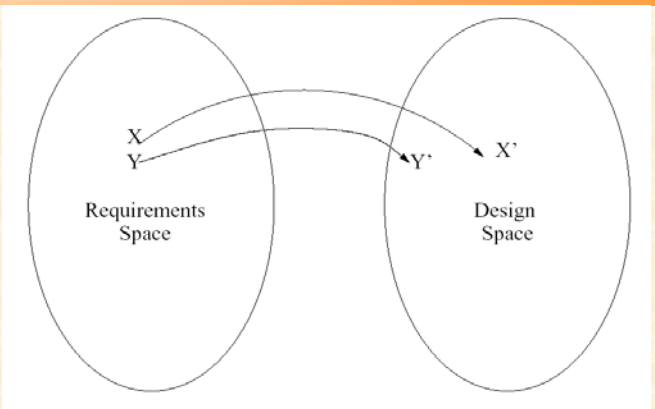
TRUST BUT ISOLATE, CHECK AND MONITOR

- What can we trust? What must we trust?
- Every module has assumptions. Do these assumptions hold?



Tracking Real-Time Systems Requirements

- Legacy software might have been patched too many times. How well does the history of patches track original design requirements and assumptions?
- A robust design might withstand reasonably large modifications; fragile designs might not. Before we can trust, we need to know how to evaluate the robustness of a design.
- Interactions between components can be subtle, especially timing interference. Let us look at the robustness issue to gain an appreciation of how much we can trust.



Want || Y' – X' || ~ || Y – X ||

Figure 1. Tracking Relation

TRACKING RELATION SHOUD HAVE NICE PROPERTIES

- Locality
  - ✓ Local changes in requirements induces local changes in design
- Scalability
  - ✓ Small changes in requirements induces small changes in design
- Robustness
  - ✓ Let us formalize the notion of "robustness" w.r.t real-time performance

WHAT IS ROBUSTNESS?

- Reduction in system load means
  - Decrease execution time of some task
  - Decrease execution frequency of some task
  - Use a faster processor
  - …
- Reduction in system load should preserve schedulability
- Robustness depends on the scheduling policy and type of timing constraints

PREEMPTIVE VS. NON-PREEMPTIVE SCHEDULING POLICIES (1)

- We know a whole lot about preemptive policies
  - Earliest-deadline
  - Fixed priority
- We know relatively little about the performance of non-preemptive policies
  - NP-completeness
- Why bother with non-preemptive policies?

PREEMPTIVE VS. NON-PREEMPTIVE SCHEDULING POLICIES (2)

- Processors are much faster, i.e., job are shorter
- Processors are more pipelined, i.e., context switch overheads are relatively high
- Communication networks bandwidth will be much higher, i.e., buffering and processing in batches
- Open systems environment, non-interference among partitions

PREEMPTIVE SCHEDULING POLICIES AND LIU&LAYLAND TASK MODEL

- Liu & Layland task systems:
  $T_i = (C_i, P_i)$
- Earliest-deadline-first policy
  $\Sigma_{1 \le j \le N} C_i / P_i \le 1.0$
- Fixed priority policy
  $\exists\, t \in (0, P_i], \Sigma_{1 \le j \le N} C_i \cdot \lceil t / P_i \rceil \le t$

ROBUSTNESS OF PREEMPTIVE POLICIES (1)

- Earliest-deadline-first policy
  $\Sigma_{1 \le j \le N} C_i / P_i \le 1.0$
- Decrease some C_k by δ
  $\Sigma_{1 \le j \le N} C_i / P_i - \delta / P_k \le 1.0$
- Increase some P_k by δ
  $\Sigma_{1 \le j \le N} C_i / P_i - (C_k / P_k - C_k / (P_k + \delta)) \le 1.0$
- Reduce all task execution times by same proportion α
  $\Sigma_{1 \le i \le N} (1 - \alpha) C_i / P_i \le 1.0$

ROBUSTNESS OF PREEMPTIVE POLICIES (2)

- Fixed Priority policy

  $\exists\, t \in (0, P_i]\,,\ \Sigma_{1 \le j \le N} C_i . \lceil t/P_i \rceil \le 1.0$
- Decrease some C_k by δ

  $\exists\, t \in (0, P_i]\,,\ \Sigma_{1 \le j \le N} C_i . \lceil t/P_i \rceil - \delta . \lceil t/P_k \rceil \le t$
- Increase some P_k by δ

  $\exists\, t \in (0, P_i]\,,\ \Sigma_{1 \le j \le N} C_i . \lceil t/P_i \rceil - (C_k . \lceil t/P_k \rceil - C_k . \lceil t/(P_k + \delta) \rceil) \le t$
- Reduce all task execution times by same proportion α

  $\exists\, t \in (0, P_i]\,,\ \Sigma_{1 \le j \le N} \alpha . C_i . \lceil t/P_i \rceil \le t$

  $\alpha < 1.0$

NON-PREEMPTIVE POLICIES ARE NOT ROBUST

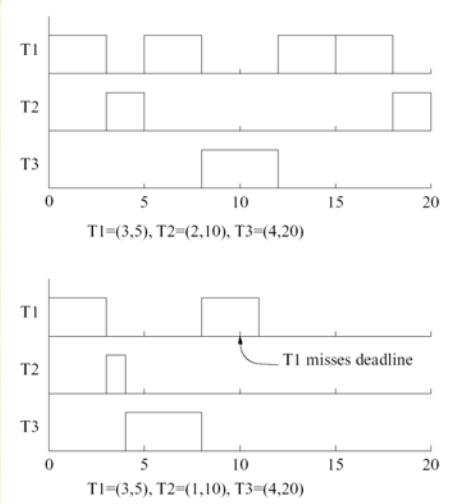
- Both earliest-deadline-first and fixed priority policies are NOT robust
- Anomaly occurs when:

  Decrease some C_k by δ

  Increase some P_k by δ

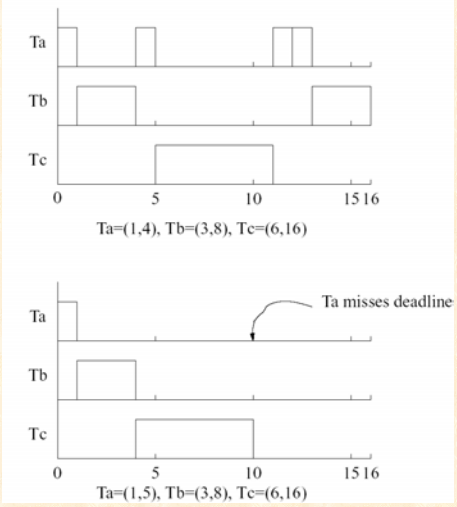
  Reduce all task execution times by same proportion α
- In general, anomaly occurs for any eager scheduler (which does not idle CPU when there is a ready task)

ANOMALY (1)



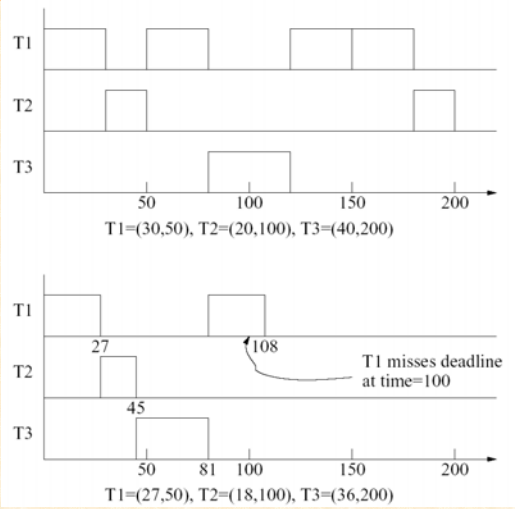
REDUCING C2 FROM 2 TO 1 CAUSES DEADLINE MISS

ANOMALY (2)



REDUCING Pa FROM 4 TO 5 CAUSES DEADLINE MISS

ANOMALY (3)



FASTER CPU CAUSES T1 TO MISS DEADLINE ALL EXECUTION TIMES DECREASED BY 10%

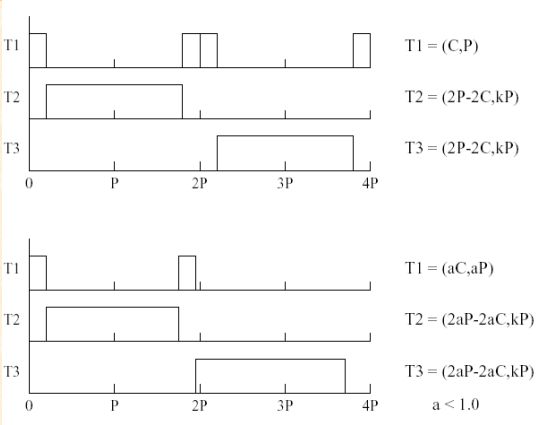
HOW BAD IS THE ROBUSTNESS PROBLEM (1)

- Conjecture:

  Anomaly occurs because processor utilization is too high

- Maybe we can determine a utilization bound α so that anomalies cannot occur when load is lower than α

  NOT SO!



Reducing All Execution Time by Same Factor Causes T1 to Miss Deadline

$U = C/P + 2(2P-2C)/kP = C/P + 4(1-C/P)/k$

$U \rightarrow 0$ as $P, k \rightarrow \infty$

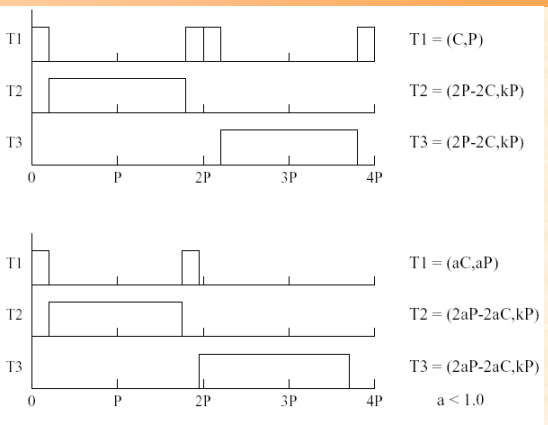
HOW BAD IS THE ROBUSTNESS PROBLEM (2)

- Conjecture:

  Anomaly occurs because jobs come in too many sizes

- Maybe if we restrict job sizes to a selected set, we can avoid anomalies

  NOT IF THERE IS MORE THAN ONE SIZE



Reducing All Execution Time by Same Factor Causes T1 to Miss Deadline

THERE ARE ONLY TWO JOB SIZES IN THIS EXAMPLE

P AND 2P-2C

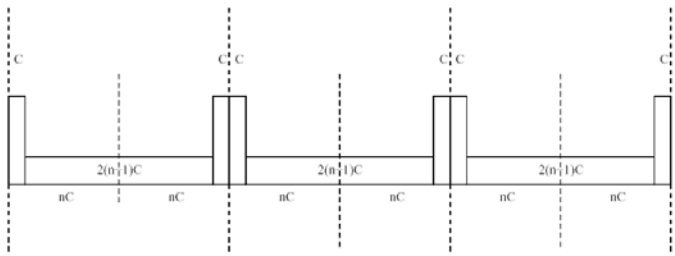
HOW BAD IS THE ROBUSTNESS PROBLEM (3)

- Conjecture:

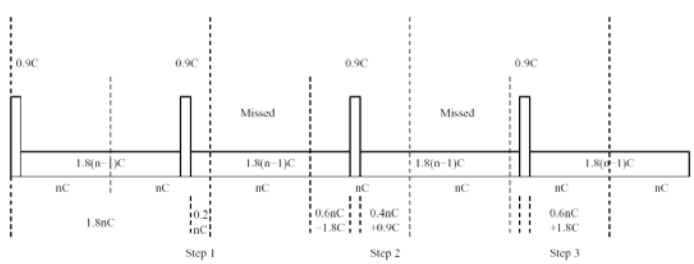
  In the worst case, a task may miss only a small fraction of its deadlines because of the anomaly.

  A TASK MAY MISS ½ OF ITS DEADLINES!

	Before CPU upgrade
T_1	(C, nC)
T_2	$(2(n-1)C, 2knC)$
$\vdots$	$\vdots$
T_{k+1}	$(2(n-1)C, 2knC)$



	After CPU upgrade
T_1	$((1-\alpha)C, nC)$
T_2	$((1-\alpha)2(n-1)C, 2knC)$
$\vdots$	$\vdots$
T_{k+1}	$((1-\alpha)2(n-1)C, 2knC)$

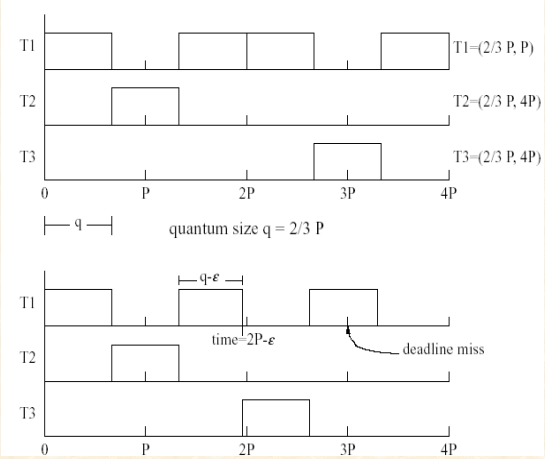


T1 MAY MISS ½ OF ITS DEADLINES

KERNELIZED MONITOR

- CPU allocation is in time quantums (size a system parameter). When a process is granted CPU time, it is allowed to occupy the CPU for a time quantum, say q time units. If the process releases the CPU early, another ready process will be immediately selected for execution, I.e., no forced idle time.
- The kernelized monitor is robust for Liu and Layland tasks under the condition: the task set is schedulable even if the execution time of each task increases by q time units

KERNELIZED MONITOR IS NOT UNCONDITIONALLY ROBUST



SECOND JOB OF T1 FINISHES EARLY BY ε CAUSING KERNELIZED MONITOR TO FAIL

Engineering solutions

- Rule of thumb:

  Keep non-preemptive tasks small compared with their periods

  May test for robustness by computing utilization with the period of each non-preemptive task decreased by q time units

- Use inserted idle time; there are two ways:
  1. Idle jobs that finish before nominal execution time.
  2. Do not start a job in "forbidden regions"

     Use parameteric scheduling

TRUST BUT ISOLATE, CHECK AND MONITOR

TRUST BUT ISOLATE, CHECK AND MONITOR

Real-time Virtual Resource: a Timely Abstraction for Embedded Systems

A. Mok & X. Feng

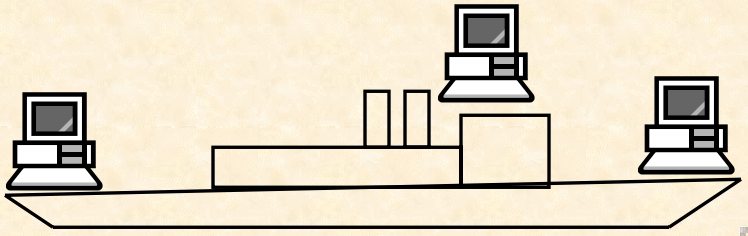
Synopsis

- Introduction
- Real Time Virtual Resource
- Task Level Scheduling Issues
- Resource Level Scheduling Issues
- Related Work
- Conclusion

Introduction

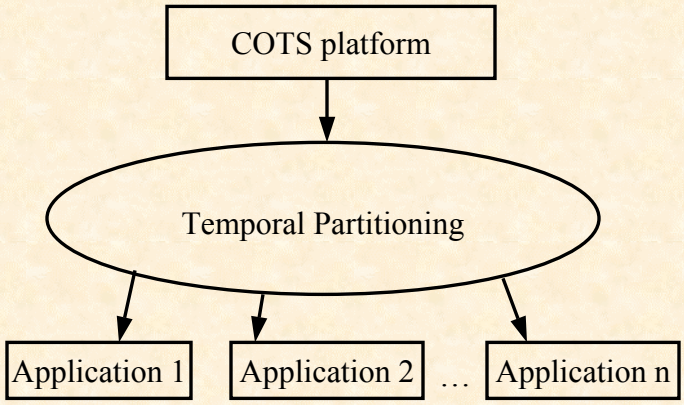
Security
Timeliness
Fault Tolerance

Applications
Navigation
Communication
Damage control
…



Introduction

Open System Architecture



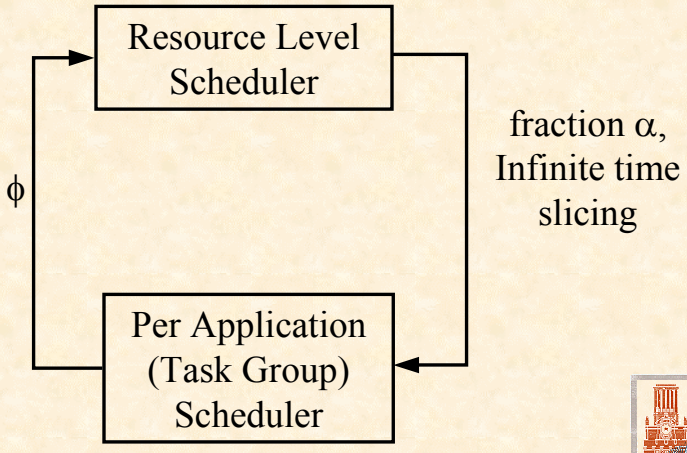
Introduction

Ideally, we want

- Temporal firewall between application task groups: no detectable timing interference
- No change to application level (task group) scheduler
- No global schedulability analysis
- No interaction between application level scheduler and resource level scheduler needed after admitting application
- Full utilization of resource

Introduction

Ideal Real-Time Resource Sharing



Introduction

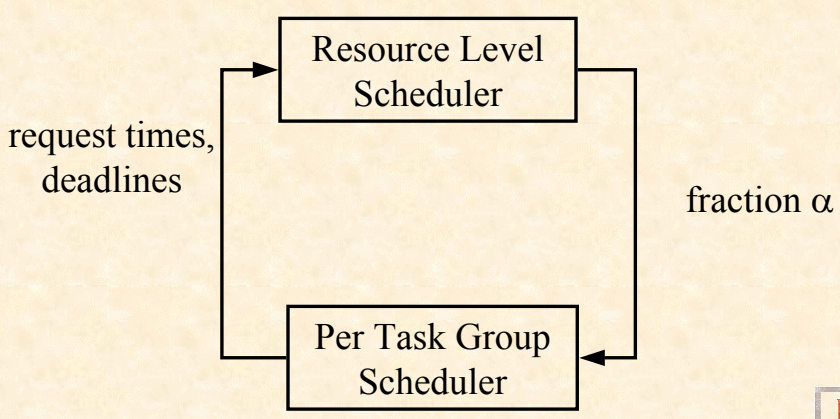
Infinite time slicing is impractical ☹

Issues with practical time partitioning schemes:

- How often must we switch partitions?
- What information needs to be exchanged between schedulers on different levels?
- When should information exchange occur?
- How is admission/schedulability analysis handled?

Introduction

Liu & Layland Task Groups



Real Time Virtual Resource

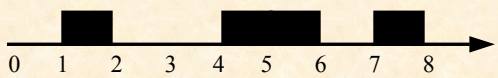
Observation

From the application programmer's point of view, time partitioning a CPU is as if program executes on a slower CPU that runs at a varying speed.

Question

What is a good way to bound the speed variation?

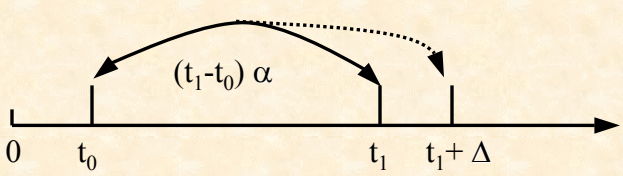
Partition Π = {(1, 4, 5, 7) 8}*



Real Time Virtual Resource

Delay Bound (also called jitter bound)

Maximum delay Δ that a partition must wait to get its share α of the resource for any time interval starting at any point in time



Eg., for a partition with delay bound = 0.1 second, 10% of a CPU that executes 10^8 instructions per second will provide 10^7 instructions in any 1.1 second

Real Time Virtual Resource

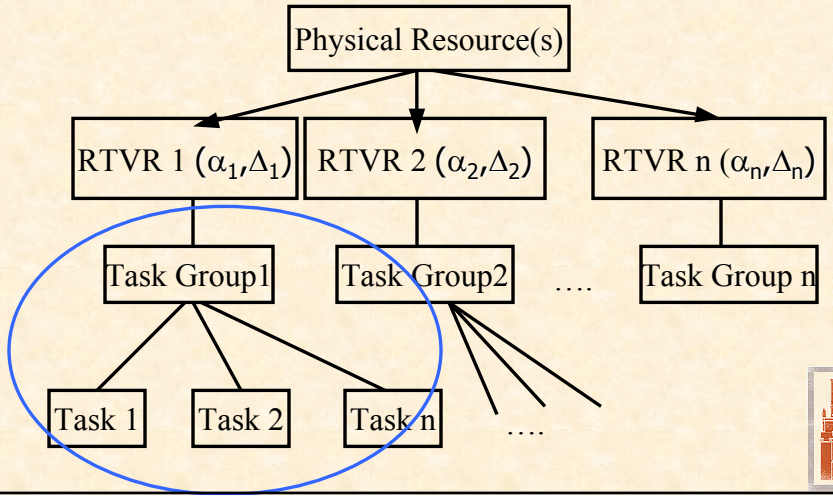
A **Bounded Delay Resource Partition** Π is a tuple (α, Δ) where

α, Availability factor of Π (fraction of resource requested)

Δ, Delay bound of Π (in real time)
Temporal Regularity of Π (in integral units)

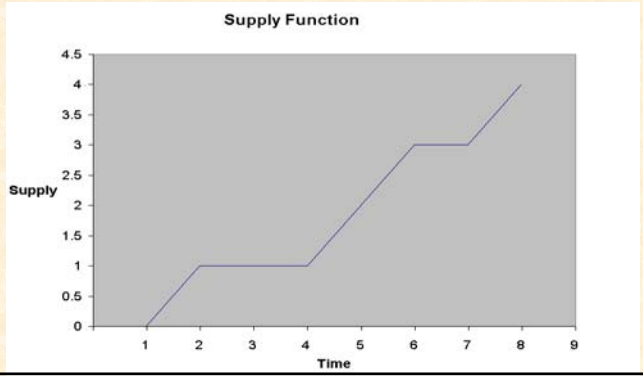
Intuitively, Δ depends on how uniformly distributed the time slots of the partition are.

Real Time Virtual Resource



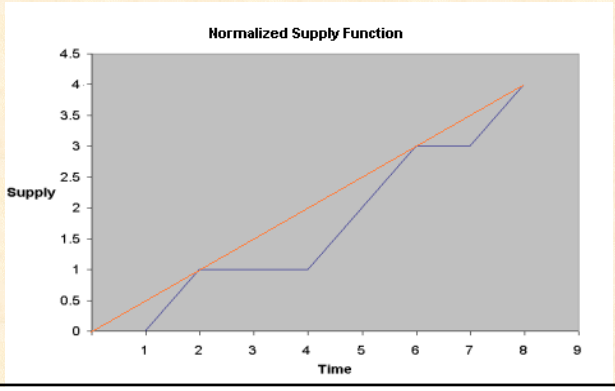
Real Time Virtual Resource

Definition: **Supply Function** S(t) of a partition is the total amount of time that is available to this partition from time 0 to time t.



Real Time Virtual Resource

Definition: **Normalized Execution** of a partition Π is an allocation of resource at a uniform, uninterrupted rate equal to the availability factor of the partition.



Real Time Virtual Resource

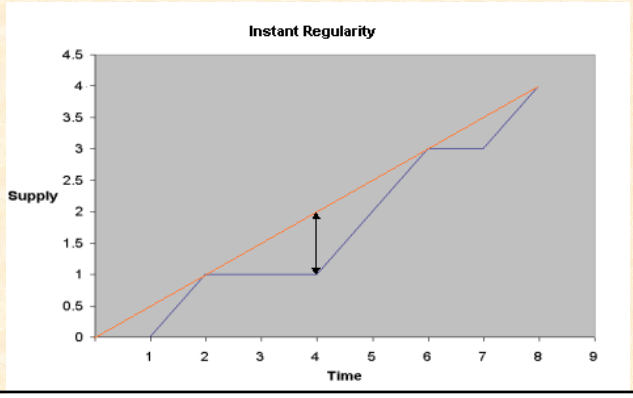
How to measure (non)uniformity of supply, i.e., the difference between normalized supply function and the partition's supply function?

Measurement	**Measurement objects**	**On what axis**
Instant Regularity	An arbitrary time point	Supply
Supply Regularity	arbitrary time intervals	Supply
Temporal Regularity	arbitrary time intervals	Time

Real Time Virtual Resource

Definition:

The **Instant Regularity** I(t) at time t on partition Π is given by S(t) -t $\alpha(\Pi)$.

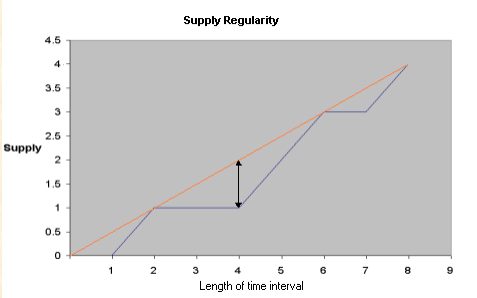


Real Time Virtual Resource

Definition:

Let a, b, k be non-negative integers, the **Supply Regularity** $R_S(\Pi)$ of Partition Π is equal to the minimum value of k such that $\forall a, \forall b.\ a<b,\ 0 \leq k,$

$|I(b)-I(a)|<k$

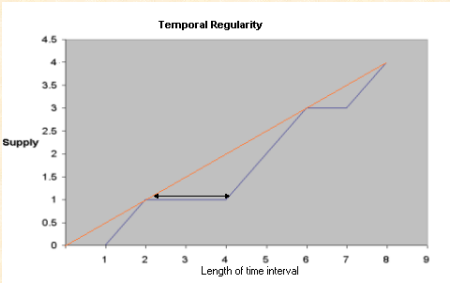


Real Time Virtual Resource

Definition:

Let a, b, e, k be non-negative integers, the **Temporal Regularity** $R_T(\Pi)$ of Partition Π is equal to the minimum value of k such that

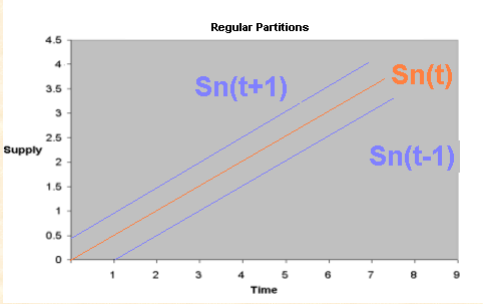
$\forall a, \forall b.\ a<b, \exists e \in [0,k], |I(b-e)-I(a)|<1$



Real Time Virtual Resource

Definition:

A **Regular Partition** is a partition with temporal regularity of 0.



Real Time Virtual Resource

Temporal regularity and supply regularity are related

Temporal	Supply
k	1+k(P/N)
$\lceil (k-1)P/N \rceil$	k

Where

P = length of period

N = number of unit slots assigned to the partition

Task Level Scheduling

Regular partitions are transparent to task scheduling

Rate Monotonic:

$$U(G) \leq m(2^{1/m}-1)\ \alpha(\Pi)$$

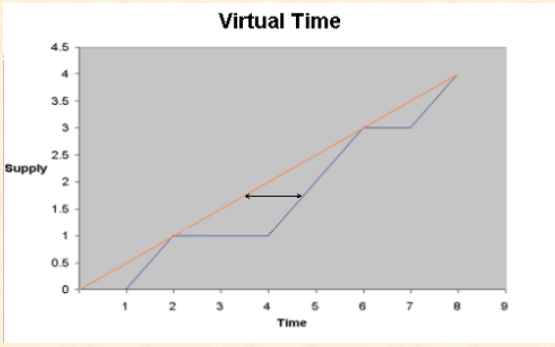
Earliest Deadline First:

$$U(G) \leq \alpha(\Pi)$$

Task Level Scheduling

Definition: **Virtual Time Scheduling**

Scheduling according to **Virtual Time**.



Task Level Scheduling

Irregular partitions are almost but not quite transparent to task scheduling

Rate Monotonic:

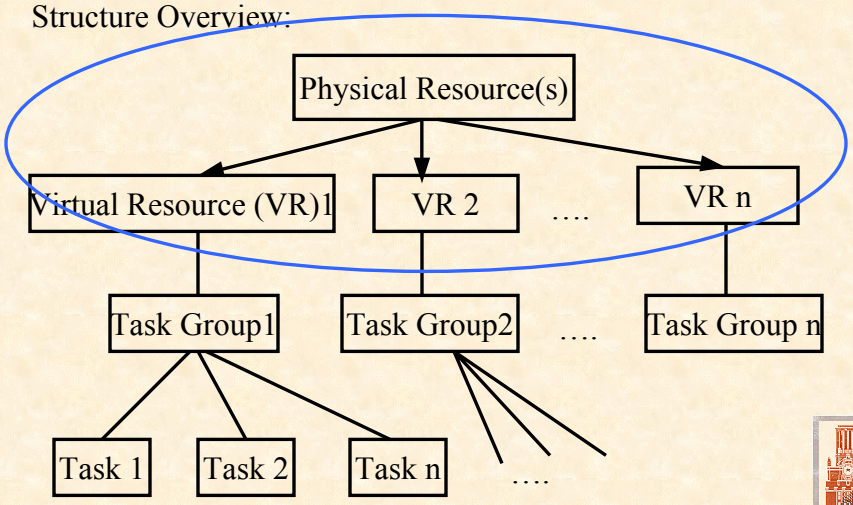
$$\sum(C_i/(p_i-k)) \leq m(2^{1/m}-1)\ \alpha(\Pi)$$

Earliest Deadline First (Shigero, Takashi & Kei):

$$\sum(C_i/(p_i-k)) \leq \alpha(\Pi)$$

Resource Level Scheduling

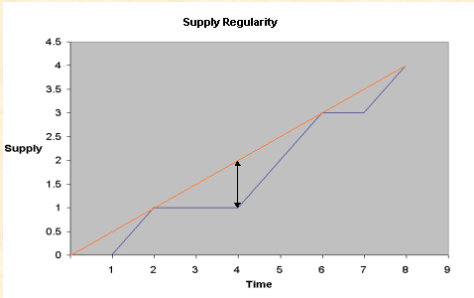
Structure Overview:



Resource Level Scheduling

Compositionality Theorem:

When two partitions Π_1 and Π_2 from the same resource are combined together they form a new partition Π_3 with supply regularity equal to the sum of supply regularities of Π_1 and Π_2.



$$S_3(t)=S_1(t)+S_2(t)$$

$$\downarrow$$

$$R_S(\Pi_3)= R_S(\Pi_1)+ R_S(\Pi_2)$$

Resource Level Scheduling

Static Scheduling Scheme:

Regular partitions with rates = powers of some number are realizable if the sum of rates $\leq$ 1.0.

Example:

For the case where the time unit bounds the precision of time interval measurement, a regular partition with rate=α receives at least $\lfloor \alpha.L \rfloor$ and at most $\lceil \alpha.L \rceil$ time units in any interval of length L. The time line below shows regular partitions with rates of (1/2, 1/4, 1/8).

Resource Level Scheduling

To compute a partition with supply regularity= 2 and rate = α, simply look for two regular partitions such that sum of rates = α

Example:

Partition Π with rate of 0.36 and supply regularity of 2:

Π: $1/4 < 0.36 < 1/2 \Rightarrow 0.36-1/4=0.11$

$1/16 < 0.11 < 1/8 \Rightarrow 1/4 + 1/8$

Two regular partitions with rates of ¼ and 1/8

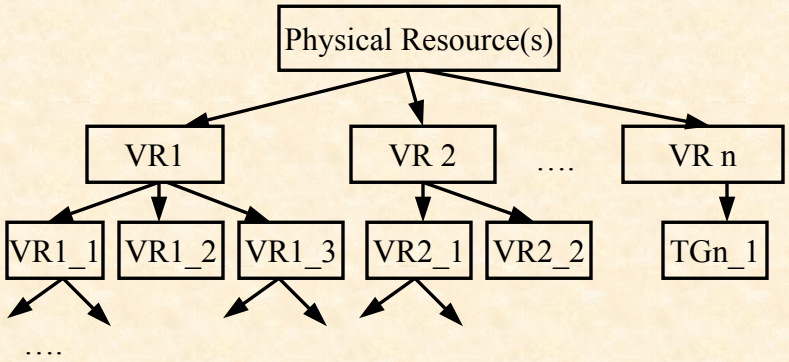
Resource Level Scheduling

Theorem:

Given a set $\{\alpha_i, 1 \leq i \leq n\}$ of availability factors of n k-supply-irregular partitions, they are schedulable if $\sum\alpha_i \leq 1-1/(2^k)$.

Resource Level Scheduling

Hierarchical Virtual Resource



Resource Level Scheduling

Theorem:

A partition group $\{ \Pi_I (\alpha_i, \Delta_I), 1 \leq i \leq n \}$ is schedulable on a partition $\Pi (\alpha, \Delta)$ if $\sum\alpha_i \leq \alpha$ and $\Delta_I > \Delta$ for all i, $1 \leq i \leq n$.

Decoupling Resource Sharing in Timeliness Verification

How do we verify timing properties of programs running on a Real Time Virtual Resource with delay bound = Δ?

- Add Δ to minimum allowable separation (delay) and subtract Δ from maximum allowable separation (deadline) between event pairs of interest. Caveat: be careful about computation's dependence on real time to make progress
- Think of it as verifying timing properties in systems where there is a jitter as big as Δ in the spacing between any two events

Related Works

- Open Systems
  - Deng & Liu
  - Baruah, Buttazzo, Gorinsky & Lipari
  - Kuo, Lin & Wang
  - …
- Proportionate Share, Resource Kernel
  - Rajkumar
  - …

Conclusion

- Rate variation bounded by temporal regularity and supply regularity
- Clean separation between application level and resource level scheduling, facilitating timeliness verification
- Real-time virtual resource is an abstraction of a resource with variable rate of service provision
- Utilization bounds of RM and EDF for regular partitions remain the same as for dedicated resources.
- Resource level scheduling can be efficiently performed by hierarchical decomposition and horizontal composition. But need Middleware availability.

TRUST BUT ISOLATE, CHECK AND MONITOR

TINMAN: A Resource Bound Security Checking System for Mobile Code

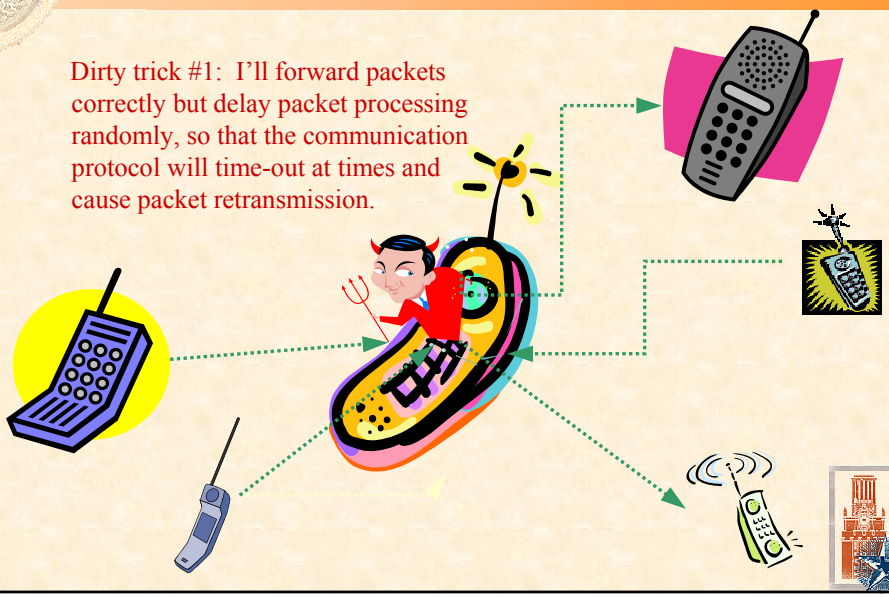
A. Mok & W. Yu

Outline

- Introduction - Resource Bound Security
- TINMAN Architecture
- Resource Usage Bound Prediction
- Usage Certificate Generation and Verification
- On-line Validation
- Conclusion

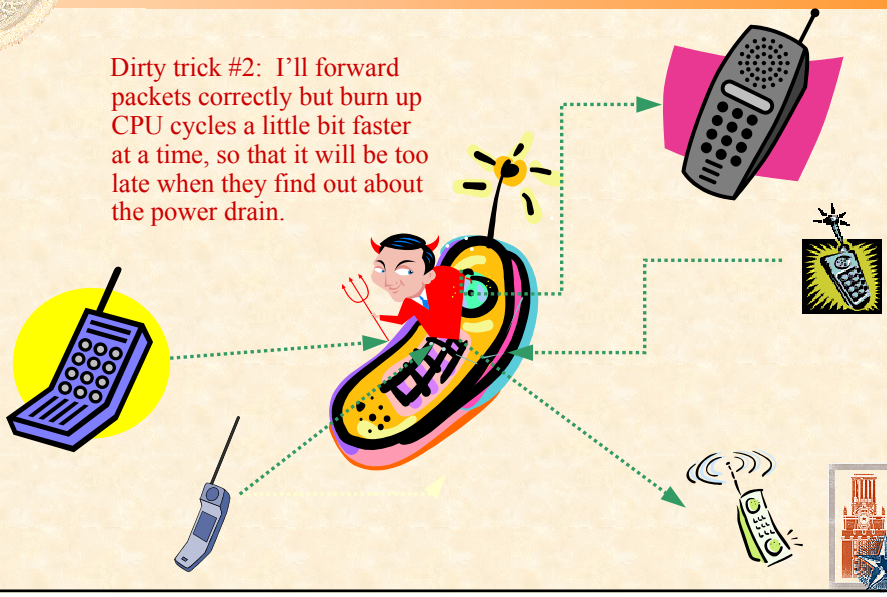
Introduction - Resource Bound Security

Dirty trick #1: I'll forward packets correctly but delay packet processing randomly, so that the communication protocol will time-out at times and cause packet retransmission.



Introduction - Resource Bound Security

Dirty trick #2: I'll forward packets correctly but burn up CPU cycles a little bit faster at a time, so that it will be too late when they find out about the power drain.



Introduction - Resource Bound Security

- Resource abuse by external code
  - ✓ malicious code intended for DoS attack
  - ✓ buggy code (e.g., infinite loops)
  - ✓ normal code exceeding its resource limit
- Defense techniques
  - ✓ Resource control
    - – Language Level (limit access, reduce expressiveness)
    - – Operating System Level (access control, runtime checking)
  - ✓ Self-certified code (PCC, TAL etc.)

Introduction - Resource Bound Security

No One Defense Is Perfect For All Settings

- Off-line verification
  - ✓ Can't be done all the time
    - – Undecidability
    - – Depends on run-time information, e.g., routing table size
  - ✓ Expensive computationally for large programs
- On-line monitoring (usage bounds by fiat)
  - ✓ Need to know what proper bounds to set
    - – Too loose ⇒ invites DOS attack
    - – Too tight ⇒ invites false alarms
  - ✓ Expensive at line speed

Introduction - Resource Bound Security

Our approach is a mixed strategy

Resource usage prediction ⇐ source code analysis

+

Code certification ⇐ theorem proving

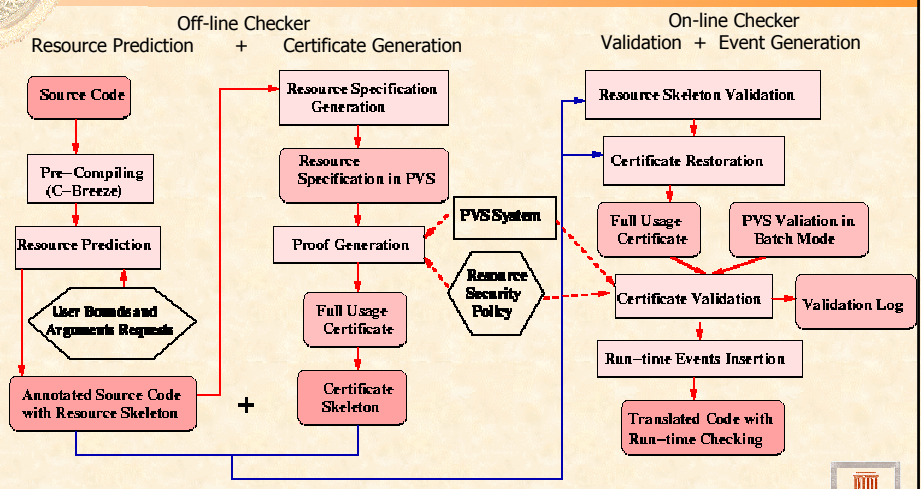
+

Runtime monitoring ⇐ event detection

Check the verifiable & Monitor the unverifiable

⇒ 100% coverage

TINMAN Architecture



TINMAN Architecture

<u>How TINMAN Works:</u>

1. Predict resource usage behavior
2. Generate usage certificate
   1. Formalize usage behavior in a proof system
   2. Mechanize proof system
3. Validate bounds on-line
   1. Authenticate usage certificate
   2. Generate event-monitoring code

- Leverage on existing tools as much as possible
- Currently TINMAN is Linux-based

Resource Usage Bound Prediction

- Source Code Analysis (Broadway C compiler)
  - ✓ Programming Constructs (loops, branches …)
  - ✓ Library functions (resource usage policy)
- Resource Usage Behavior Prediction
  - ✓ User-provided information
  - ✓ Parameterized resource usage bound
- Execution Time Evaluation
  - ✓ Timing schema approach [Park91]
- Live-memory Demand Analysis
  - ✓ Explicit memory allocation and path analysis

Result is a resource skeleton - an abstraction of resource usage behavior

Resource Usage Bound Prediction

```
B1: int dpt, i,
    sender = 126; group = 1;
U2: init_nodelist(LISTLENGTH);
B3: dpt = getrecord(
          group, sender);
    m = &n;
    m->time = n.time - 10;
C4: if(m->nodes!=NULL){
L5:  for(i=0; i<m->length; i++)
B6:     routefornode(&n,
            m->nodes[i]);
    }else{
B7:     delivertoapp(&n,dpt);}
    }
```

```
B1: {..}
/*@B1: T[T[Entry]+4]
       M[M[Entry]]*/
U2: {..}
/*@U2: T[T[C10]]
       M[M[C10]]*/
..

C4: {..}
/*@C4: TMAX[T[L5],T[B7]]
       MMAX[M[L5],M[B7]]*/
```

Source C Code → Annotated Code

Resource Usage Bound Prediction

```
void init_nodelist(int length)
{..
B9:   i = 0;
      n.time = 100;
      n.address = 100;
      n.length = length;
      rid=generateRandom(100);
C10:  if(rid>50){
B11:    n.nodes=(W_N*)malloc(
            sizeof(W_N)*length);
L12:    for(i=0; i<length; i++);
B13:        n.nodes[i] = i;
      }else
B14:    n.nodes = 0;
}
```

```
/*@U2 Entry: T[B1] M[B1]
B9: {.. }
/*@B9:
 T[T[Entry]+16+TgenerateRandom]
 M[M[Entry]+MgenerateRandom]*/
C10: {..
/*@L12lb=10*/
L12:{..}
/*@L12: T[T[B11]+L12lb*12+3]]
        M[M[B11]]*/
..
}
/*@C10: T[MAX[T[L12],T[B14]]]
        M[MAX[M[L12],M[B14]]]*/
```

Source C Code → Annotated Code

Usage Certificate Generation

Resource Specification: Extended Hoare Logic

Examples of proof obligations:

- **Basic block task B9**

  PRE9: {now = t0 + 4 ∧ mem = m0 ∧ terminate}
  {B9}

  POS9: {now <= t0 + 20 + TgenerateRandom ∧ mem <= m0 + MgenerateRandom ∧ terminate}

- **Loop task L12**

  PRE12: {now <= t0 + 22 + TgenerateRandom + Tmalloc ∧ L12lb = 10 ∧ mem <= MgenerateRandom + 40 ∧ terminate}
  {L5}

  POS12: {now <= t0+ 22 + TgenerateRandom + Tmalloc + L12lb*12+3 ∧ mem <= MgenerateRandom + 40 ∧ terminate}

Usage Certificate Generation

Proof System

- **Axiom 1** Basic Block Tasks

  $\forall$ *tb*: Time:

  {P[(now + *tb*)/now, mem/mem] $\wedge$ terminate} *BB (tb)* {P}

- **Axiom 2**: Service Call Task

  $\forall$*ts*: Time, $\forall$*ms*: Memory:

  {P[(now+*ts*)/now, (mem+*ms*)/mem] $\wedge$ terminate} *SRVC (ts, ms)* {P}

- **Axiom 3**: Condition Expression

  $\forall$*tb*: Time, $\forall$*mb*: Memory:

  {P[(now+*tb*)/now, (mem+*mb*)/mem] $\wedge$Terminate}*COND (tb, mb)* {P}

Usage Certificate Generation

- **Proof Rule 1**: Sequential Tasks

  $$\frac{\{P\}\ T1\ \{R\},\ \{R\}\ T2\ \{Q\}}{\{P\}\ T1;\ T2\ \{R\}}$$

- **Proof Rule 2**: Choice Task

  $\forall$*tb*: Time, $\forall$*mb*: Memory:

  $$\frac{\{P \wedge b\}COND(tb, mb);\ T1\ \{Q\},\quad \{P \wedge \neg b\}COND(tb, mb);\ T2\ \{Q\}}{\{P\}\ \textbf{\textit{if}}\ b\ \textbf{\textit{then}}\ T1\ \textbf{\textit{else}}\ T2\ \{Q\}}$$

- **Proof Rule 3**: Loop Task

  $\forall$*tb*: Time, $\forall$*mb*: Memory:

  $$\frac{\begin{array}{l}\{P \rightarrow Inv\},\ \{Inv \wedge b \wedge terminate\}\ COND(tb,mb);T\ \{inv\},\\ \{Inv \wedge \neg b \wedge terminate\}\ COND(tb,mb);\ \{R\},\\ \{R \vee \{Inv \wedge \neg terminate\}\} \rightarrow Q\end{array}}{\{P\}\ \textbf{\textit{while}}\ b\ \textbf{\textit{do}}\ T\ \{Q\}}$$

Usage Certificate Generation

- Translate Resource Specification into PVS Logic
- Automate proof generation using PVS strategies
  - ✓ Simple task, Sequential tasks , Choice task, Loop task
- Generate Certificate Skeleton

Usage Certificate Generation

Translate Resource Specification into PVS Logics

```
P9 : [State->bool] =
    ((LAMBDA s : state) : now(s) = t0 + 4
    AND mem(s) = m0 AND terminate(s) )
B9: program = seq (bb(16), srvc(TgenerateRandom,
                                MgenerateRandom))
```

(Note: Program is a predicate type relating two states)

```
Q9 : [State->bool] = (LAMBDA s :
    now(s) = t0 + TgenerateRandom + 20 AND
    mem(s) = m0 + MgenerateRandom AND
    terminate(s))
CORB9 : LEMMA B9 => spec (P9, Q9)
```

Usage Certificate Generation

Automate proof generation using PVS strategies

Example: Simple task strategy

```
(defstep simpletask (task p, q)
 (auto-rewrite task p q)
 (auto-rewrite "NOT" "AND" "OR" "IMPLIES" "Valid")
 (auto-rewrite "bb" "srvc" "seq"
                "ifthenelse" "ifthen" "spec")
 (expand "=>") (skosimp) (assert)
 (repeat (try (skosimp*) (assert) (skip)))))
```

Usage Certificate Generation

- PVS Proof for CORB9:

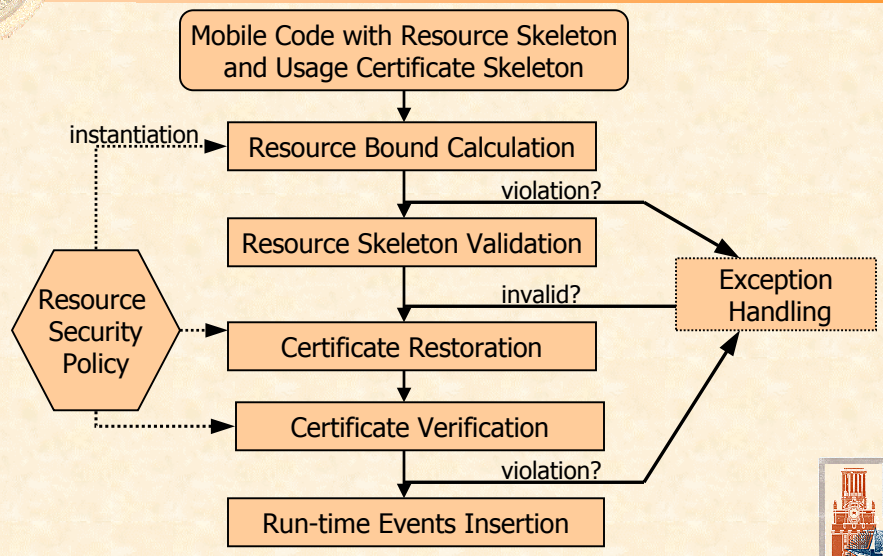
```
(|CORB9| "" (EXPAND "B9") (("" (EXPAND "P9")
(("" (EXPAND "Q9") (("" (EXPAND "seq") (("" (EXPAND "spec")
 (("" (EXPAND "bb") (("" (EXPAND "srvc") (("" (EXPAND "=>")
  (("" (ASSERT) (("" (SKOSIMP*) (("" (ASSERT)
   (("" (SKOSIMP*) (("" (ASSERT) (("" (SKOSIMP*)
    (("" (ASSERT) NIL) ..NIL))
```

- Corresponding Certificate Skeleton:

CORB9: S1 B9 P9 Q9

(Note: To prove CORB9, apply strategy S1 to B9 P9 Q9)

On-line Validation



Lessons Learned from Experiments

- Ongoing experimentation with open source NET-SNMP toolkit
- Code size augmentation (+18%-20%)
- Automatic certificate generation (seconds)
- On-line certificate validation (seconds)
- Run-time monitoring overhead (milliseconds)

Reported in *"Enforcing Resource Bound Safety for Mobile SNMP Agents", Proceedings of ACSAC, Las Vegas, December 2002*

A Target Application

Defense against DDOS by "Pushback"

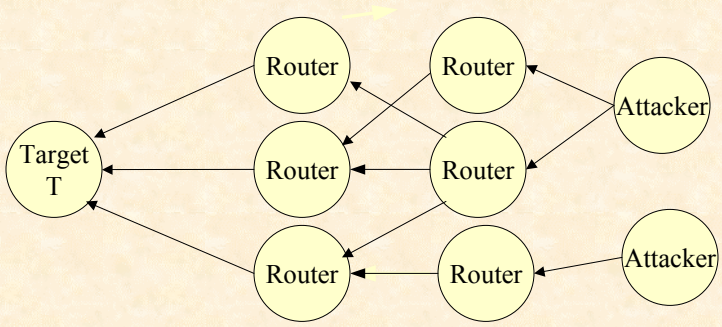
- Upon attack, target launches probes to instruct neighboring routers to recognize attack packets
- Routers trace attack packets to immediate predecessor routers from which the attack packets arrive and then launch probes to instruct those predecessors to recognize attack packets, and the process repeats …
- Routers at the edge of the network shut off attack traffic at the sources

Caveat

Attacker may attack the routers themselves. We need to ascertain *Resource Bound Security* on routers.

A Target Application

DDOS Attack and Resource Bound Security



Conclusion

- TINMAN framework provides full coverage for resource bound safety

  Source Code Analysis + Formal method + Runtime Monitoring

- Static validation reduces run-time checking overhead. Runtime monitoring checks resource assertions that cannot be verified statically
- Code providers are not to be trusted at all; TINMAN aims to enable code recipient to achieve resource bound safety relatively efficiently

TRUST BUT ISOLATE, CHECK AND MONITOR

Event-Based Real-Time Monitoring

A. Mok, G. Liu & C.G. Lee

Outline

- Introduction
- Event Model & Functions
- Simple Constraint & Timing Constraint
- Implicit Constraint
- Compilation & Monitoring Approach
- Timing Constraint on Time Intervals
- Timing Constraint with Confidence Threshold
- Summary

Introduction

- Many applications of timing constraints
  - ✓ Timing constraints exist in many real-life applications, e.g., air-traffic control, medical-life support, network management, stock market watch, etc.
- Need for monitoring timing constraints
  - ✓ Despite real-time scheduling effort, timing constraint violation may still occur due to system failures, design errors as well as implementation errors
- Need for detecting violations as early as possible
  - ✓ Catching timing constraint violation as early as possible is important in many real-time systems

Event Model

- Event
  - ✓ Represents state change of interest

    Examples :
    - StartTransation : Transaction started
    - RecvMsgFromNodeA : Received a message from Node A
  - ✓ May have multiple occurrences :

    (e,i) : ith instance of event e

    Examples :
    - (StartTransation, 1)
    - (RecvMsgFromNodeA, 2i+1)

RTL (Real Time Logic)

- RTL is a subset of Presburger Arithmetic plus an uninterpreted function
- Syntax:
  - ✓ @(e,i) = time of occurrence of ith instance of event e
  - ✓ Time has domain the set of non-negative integers
  - ✓ Example

    Let ↑TALK denote the event: Mok's talk starts

    Let ↓TALK denote the event: Mok's talk finishes

    Then @(↑TALK ,1) ≥ October 2, 2003,

    @(↓TALK ,1) ≤ July 13, 3000,

RTL (Real Time Logic)

- Semantics:
  - ✓ A computation is a sequence of sets of event names (instances) indexed by their common time of occurrence
  - ✓ A computation satisfies a timing property P is the time values of the event occurrences in the computation satisfy P.

Occurrence index / Event	1	2	3	4	5	6	…
↑TALK							
↓TALK							

Event Functions

- ✓ $@(e,i)$: occurrence time of the ith instance of event e
- ✓ $\#(e,t)$: index of the most recent instance of event e at time t

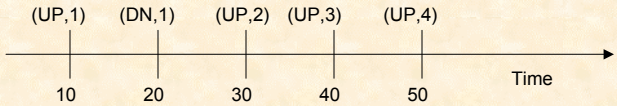
$$\#(e,t) = \begin{cases} 0 & \text{if } t < @(e,1) \\ \max(i)\ s.t.\ @(e,i) \le t & \text{otherwise} \end{cases}$$

- ✓ $@_r(e,t,i) = @(e,\#(e,t) + i)$
  - – the next i^{th} instance of e at t when $i > 0$
  - – the ith recent instance of e at t when $i \le 0$

Event Function Examples

@(UP,2) = 30

$@_r$(UP, @(DN,1), 1)
= @(UP, #(UP, @(DN,1)) +1)
= @(UP, #(UP, 20) +1)
= @(UP, 1+1) = 30



Simple Constraint

Form: $T_1 + D \geq T_2$

- ✓ T_1, T_2 : @ function, relative @ function, or 0
- ✓ D : an integer constant
- ✓ one variable may appear in @ functions' occurrence parameters

Examples

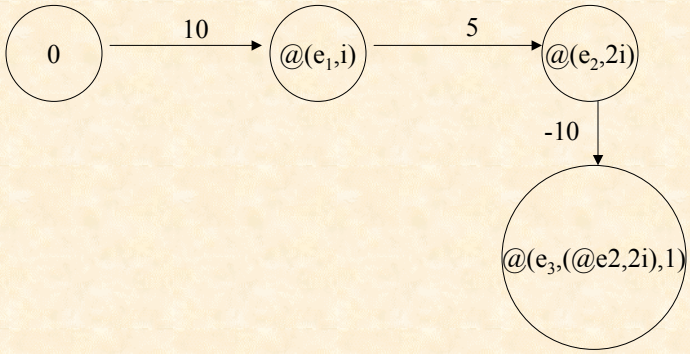
- ✓ Deadline constraint: $@(e_1,i) + 10 \geq @(e_2,i+2)$
- ✓ Delay constraint: $@(e_2,i) - 10 \geq @(e_3,2i)$

Timing Constraint

- Formulas of simple constraints in disjunctive normal forms
- Only one variable is allowed in one formula
- Example:

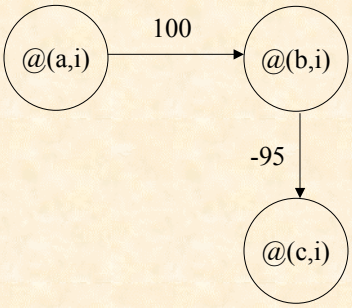
  $@(StartT,i) + 100 \geq @_r(CommitT,@(StartT,i),0)$
  $\vee\ @(StartT,i) + 100 \geq @_r(AbortT,@(StartT,i), 0)$

Constraint Graph



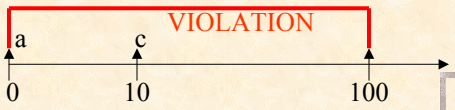
$@(e_1,i) + 5 \geq @(e_2,2i) \wedge 10 \geq @(e_1,i)$
$\wedge\ @(e_2,2i) - 10 \geq @_r(e_3, @(e_2,2i), 1)$

Detecting Violations without Implicit Constraints

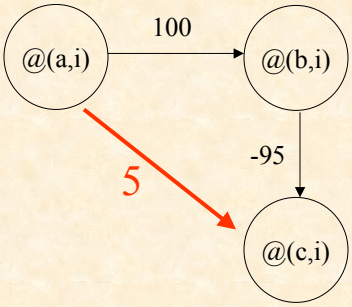


$@(a,i) + 100 \geq @(b,i)$ (1)

$@(b,i) - 95 \geq @(c,i)$ (2)



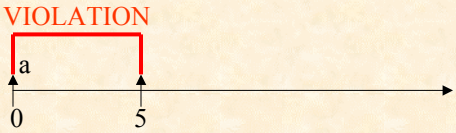
Detecting Violations with Implicit Constraints



$@(a,i) + 100 \geq @(b,i)$ (1)

$@(b,i) - 95 \geq @(c,i)$ (2)

→ $@(a,i) + 5 \geq @(c,i)$ (3)

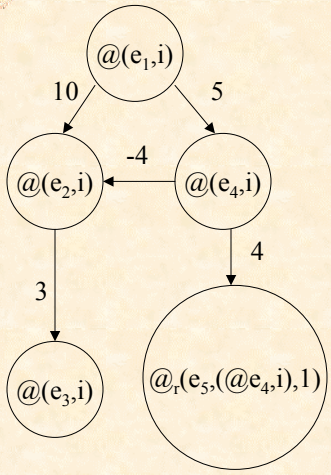


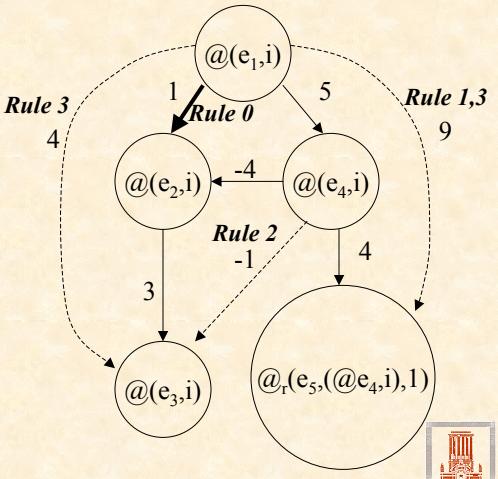
(3) is violated → (1) or (2) will be violated eventually

Separating Compilation & Monitoring

- Compilation : O(n^3)
  - ✓ Calculate all pairs shortest path
  - ✓ Detect negative cycles
  - ✓ Eliminate unnecessary paths
- Run-time Monitoring : O(n)
  - ✓ Initiate the compiled constraint graph
  - ✓ Update paths
  - ✓ Check related constraints

Removing Unnecessary Constraints





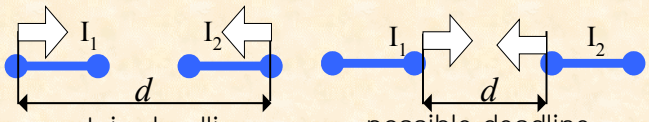
Removing Unnecessary Constraints

- *Rule 0* : Longer paths are unnecessary
- Path $u \xrightarrow{l} v \xrightarrow{m} w$ is unnecessary iff
  - ✓ *Rule 1* : *u* or *v* correspond to relative @ function or
  - ✓ *Rule 2* : $l \leq 0$ or
  - ✓ *Rule 3* : $m \geq 0$

Extension: Timing Constraints on Time Intervals

- Interval Timestamp
  - ✓ [*min_time*, *max_time*]
  - ✓ Actual occurrence can be anywhere between *min_time* and *max_time*
- π : maximum length of the timestamp
- Simple Timing Constraint : $I_1 + D \geq I_2 \;\; U$
  - ✓ I_1, I_2 : interval timestamp
  - ✓ D : delay ($D \geq 0$) or deadline ($D < 0$)
  - ✓ U : *certainly* or *possibly*

Constraints on Time Intervals

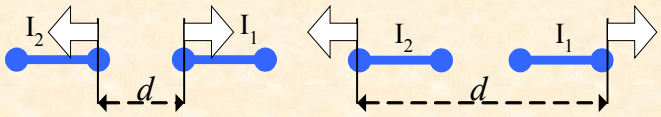


certain deadline

$I_1 + d \geq I_2$ *certainly*

possible deadline

$I_1 + d \geq I_2$ *possibly*



certain delay

$I_1 - d \geq I_2$ *certainly*

possible delay

$I_1 - d \geq I_2$ *possibly*

In all cases, $d \geq 0$

Deriving Implicit Constraints

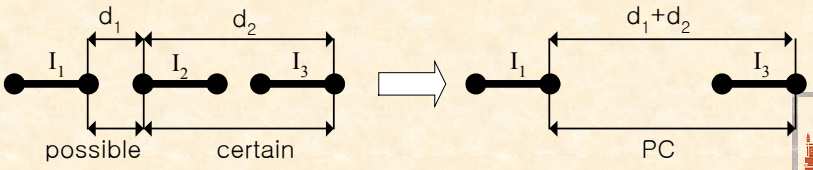
$I_1 + d_1 \geq I_2$ *possibly* (1)

$I_2 + d_2 \geq I_3$ *certainly* (2)

$\max(I_1) + d_1 \geq \min(I_2)$ (1')

$\min(I_2) + d_2 \geq \max(I_3)$ (2')

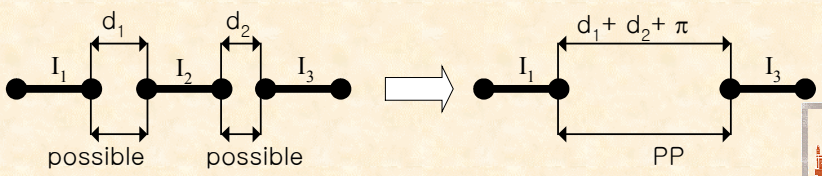
$\max(I_1) + d_1 + d_2 \geq \max(I_3)$ (1')+(2')



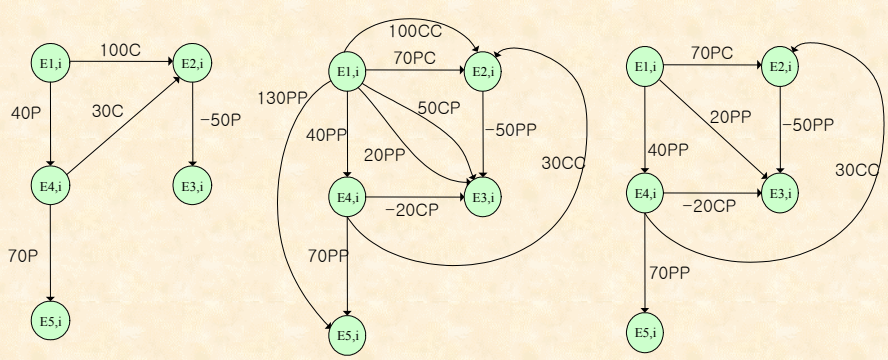
Deriving Implicit Constraints

$$I_1 + d_1 \geq I_2 \;\; possibly \qquad (1)$$
$$I_2 + d_2 \geq I_3 \;\; possibly \qquad (2)$$

$$\max(I_1) + d_1 \geq \min(I_2) \qquad (1')$$
$$\max(I_2) + d_2 \geq \min(I_3) \qquad (2')$$
$$\max(I_1) + d_1 + d_2 + \pi \geq \min(I_3) \qquad (1')+(2')$$



Example



(A) Original Constraint Graph

(B) All-Pairs Shortest Path

(π=20)

(C) Compiled Constraint Graph

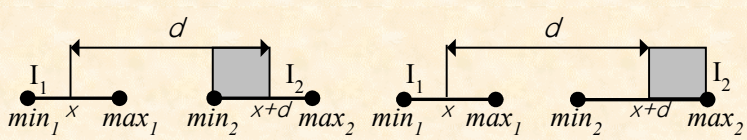
Extension: Timing Constraints with Confidence Thresholds

- Simple Timing Constraint: $I_1 + d \geq I_2$ with P
  - ✓ I_1, I_2: interval timestamp
  - ✓ D: delay ($D \geq 0$) or deadline ($D<0$)
  - ✓ P: confidence threshold
  - ✓ Timing Violation :

    happens when the specified confidence threshold (P) of the timing constraint cannot be maintained
- Example

  $@(e_1,i) + 100 \geq @(e_2,i)$ with 50%

Satisfaction Probability Calculation



(A) Deadline **(B) Delay**

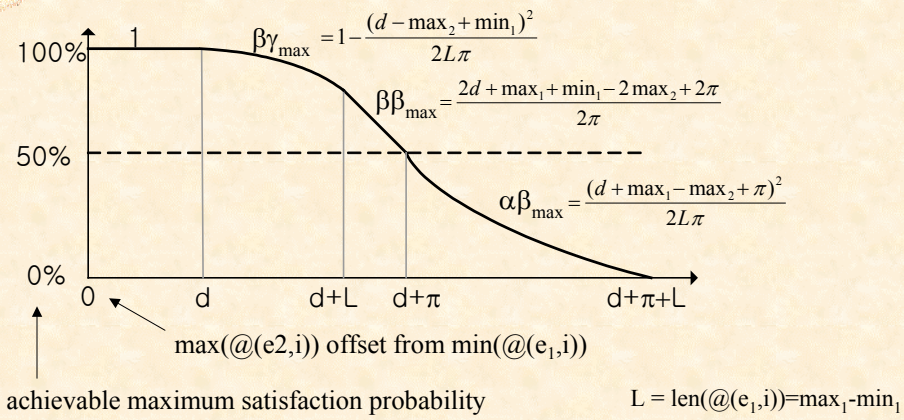
Satisfaction Probability of a deadline

$$= \frac{1}{len(I_1)len(I_2)} \int_{\min(I_1)}^{\max(I_1)} MIN(MAX(x+d-\min(I_2),0), len(I_2))\, dx$$

Satisfaction Probability of a delay

$$= \frac{1}{len(I_1)len(I_2)} \int_{\min(I_1)}^{\max(I_1)} MIN(MAX(\max(I_2)-x-d,0), len(I_2))\, dx$$

Monitoring Timing Constraint with Confidence Threshold



Timing Constraint :@(e_1,i) + d ≥ @(e_2,i) with P

Conclusion

- Event Based Monitoring
- Timing Constraint
- Implicit Constraint
- Pruning Unnecessary Constraints
- Extensions
  - ✓ Timing Constraints on Time Intervals
  - ✓ Timing Constraints with Confidence Thresholds

Discussion

Nicholas Halbwachs
Pascal Raymond

Verimag, CRNS
Grenoble, France

Synchronous Programming of Reactive Systems

— A Tutorial —

Nicolas Halbwachs
Verimag/CNRS

Pascal Raymond
Verimag/CNRS

Grenoble, France

Reactive Systems

Permanent reaction

to an environment that cannot wait

embedded systems

e.g., transportation, industrial control

Specific features

- deterministic
- concurrent (logical $\neq$ physical)
- safety critical

Logical concurrency

ex. A digital watch:

- time keeper
- alarm
- stopwatch
- display manager
- button handler

Design these modules separately, compose them concurrently.

Usual (asynchronous) languages for concurrency don't work

Example: Every 60 seconds, emit a signal MINUTE

An attempt in ADA style:

```
task A:    loop
               delay 60; B.MINUTE!
           end
```

Rendez-vous (symmetric communication) doesn't work.

Non-deterministic scheduling (asynchronous interleaving) doesn't work.

No broadcasting

How are reactive systems commonly implemented?

Simple implementation (event driven)

```
< Intialize Memory >
foreach input_event do
    < Compute Outputs >
    < Update Memory >
end
```


Even simpler implementation (periodic sampling)

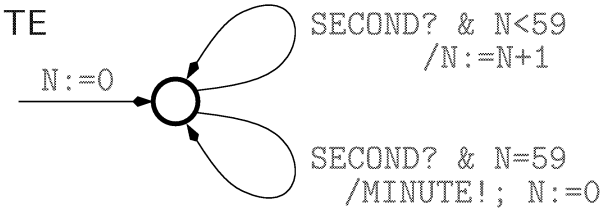
```
< Intialize Memory >
foreach period do
    < Read Inputs >
    < Compute Outputs >
    < Update Memory >
end
```


$\sim$ interpreted automaton

a loop iteration = a transition = a logical instant

Our example in Esterel:

```
every 60 SECOND do
    emit MINUTE
end
```



"Real-time" correctness condition

max transition time < min environment delay

Synchronous programming

= high level, structured, modular

description of interpreted automata

concurrency = synchronous product

```
[
    every 60 SECOND do emit MINUTE end
||
    every 60 MINUTE do emit HOUR end
]
```

Another point of view:

time, concurrency, and compositionality

$$\Delta(f(x))?$$

depends on implementation of f, of the target machine, and generally of x

Abstraction: $\Delta(f(x)) = \delta$

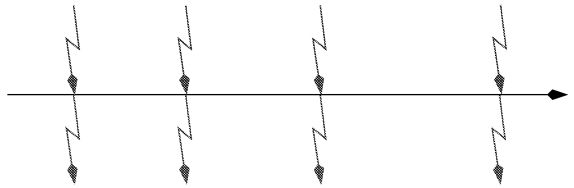
Compositionality: $f(x) = g(h(x))$

$$\Delta_f = \Delta_g + \Delta_h \qquad \delta = \delta + \delta$$

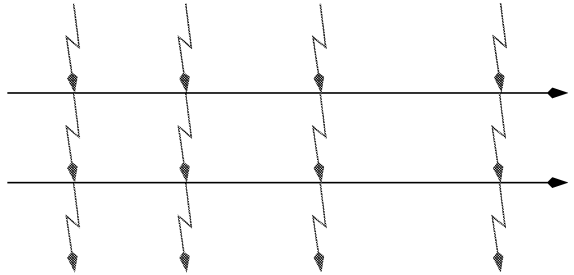
Two solutions:

$\delta = 0$ (synchrony), $\delta =?$ (asynchrony)

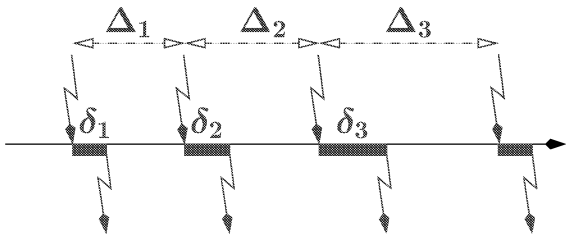
Abstract synchronous behavior: sequence of reactions to input events, to which all processes take part:



Composition of behaviors:



Concrete behavior



Valid abstraction as long as $\delta_i < \Delta_i$

What's new?

Classical in synchronous circuits

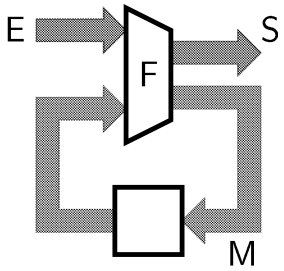
- synchronous communicating Mealy machines
- dynamic Boolean equations
- gate and latch networks

Classical in control engineering

data-flow synchronous formalisms

- differential or finite difference equations
- block-diagrams, analog networks

Connexion with synchronous circuits



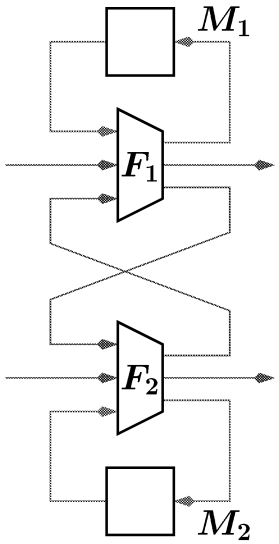
$$(S_n, M_n) = F(E_n, M_{n-1})$$

⟶ Data-flow languages

(Lustre/Scade, Signal/Sildex)

In Lustre: (S,M) = F(E, pre(M))

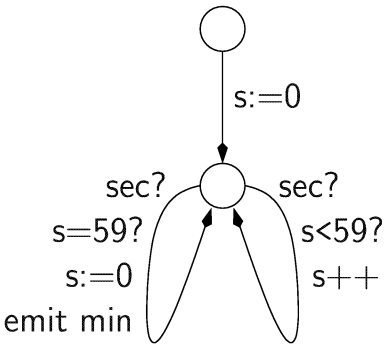
Parallel composition

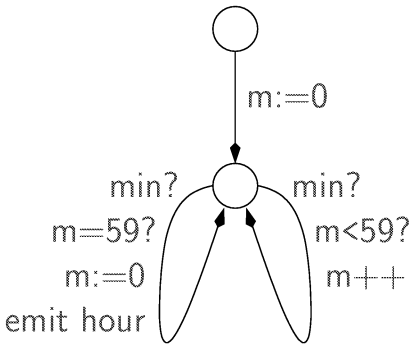


(S1,L1,M1) = F1(E1,L2,pre(M1))

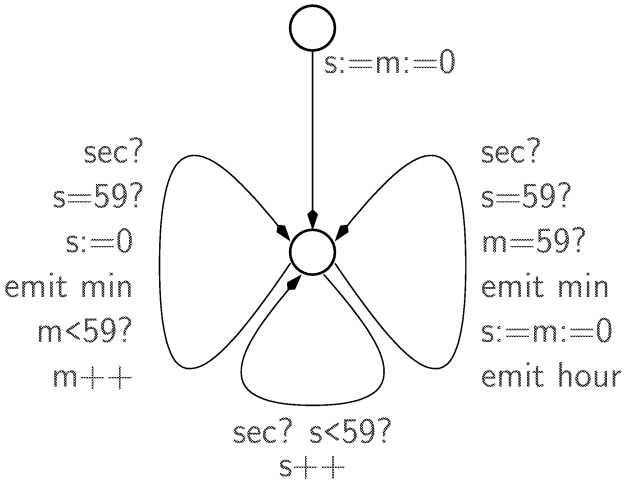
(S2,L2,M2) = F2(E2,L1,pre(M2))

Connexion with synchronous automata





Synchronous product of automata



⟶ Imperative Languages (Esterel, Synccharts)

In Esterel:

```
    every 60 sec do emit min end
  ||
    every 60 min do emit hour end
```


Synchronous Languages

Imperative

- StateCharts
- Esterel
- Argos, SyncChart

Declarative

- Lustre, Signal

Industrial use

Avionics:

Airbus, Honeywell, Eurocopter (Lustre)
Dassault (Esterel)
Snecma (Signal)

Nuclear plants:

Schneider-Electric, Electricité de France (Lustre)

CAD:

Cadence, Synopsys (Esterel)

Telecom:

Thomson, TI (Esterel)

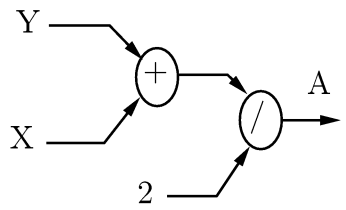
The Lustre Language

Pascal Raymond

Vérimag/CNRS, Grenoble

Data-flow approach

- Classical in automatics, circuits




```
node Average(X, Y : int)
returns (A : int);
let
   A = (X + Y) / 2 ;
tel
```

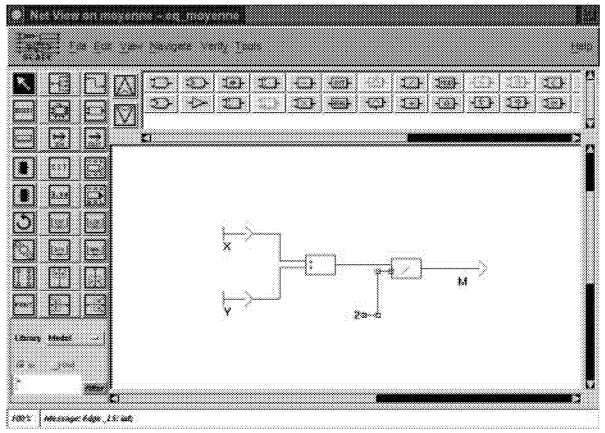
- Synchronous: discrete time $= \mathbb{N}$
  $\forall t \in \mathbb{N} \;\; A_t = (X_t + Y_t)/2$

Another version

```
node Average(X, Y : int)
returns (A : int);
var S : int;      ← local variable
let
   A = S / 2;      ← equations
   S = X + Y;      (order don't care)
tel
```

- Set of equation (no order)
- One definition for each output and local
- Data flows: variables are infinite sequences of values

Lustre (textual) and Scade (graphical)



Combinatory programs

- Basic types: bool, int, real
- Constants:
  $2 \equiv 2, 2, 2, \cdots$
  `true` $\equiv true, true, true, \cdots$
- Point wise operators:
  `X` $\equiv x_0, x_1, x_2, x_3 \ldots$ `Y` $\equiv y_0, y_1, y_2, y_3 \ldots$
  `X + Y` $\equiv x_0 + y_0,\ x_1 + y_1,\ x_2 + y_2,\ x_3 + y_3 \ldots$

Boolean example

```
node Nand(X,Y: bool) returns (Z: bool);
var U: bool;
let
   U = X and Y;
   Z = not U;
tel
```

Execution:

X	*true*	*true*	*false*	*true*	*true*	*false*	...
Y	*false*	*true*	*false*	*false*	*true*	*false*	...
U	*false*	*true*	*false*	*false*	*true*	*false*	...
Z	*true*	*false*	*true*	*true*	*false*	*true*	...

Example: `if` operator

```
node Max(A,B: real) returns (M: real);
let
   M = if (A >= B) then A else B;
tel
```

Warning: functional `if then else`, not statement

```
let
  if (A >= B) then M = A ;
  else M = B ;
tel
```


Memory

- Previous operator: `pre`

X	x_0	x_1	x_2	x_3	x_4	...
pre X	nil	x_0	x_1	x_2	x_3	...

i.e. $(\mathrm{pre}X)_0$ undefined and $\forall i \neq 0\ (\mathrm{pre}X)_i = X_{i-1}$

- Initialization: `->`

X	x_0	x_1	x_2	x_3	x_4	...
Y	y_0	y_1	y_2	y_3	y_4	...
X ->Y	x_0	y_1	y_2	y_3	y_4	...

i.e. $(X\text{->}Y)_0 = X_0$ and $\forall i \neq 0\ (X\text{->}Y)_i = Y_i$

Raising edge

```
node Edge (X : bool) returns (E : bool);
let
   E = false -> X and not pre X ;
tel
```

Min and max of a sequence

```
node MinMax(X : int)
returns (min, max :  int);   ← several outputs
let
   min = X -> if (X < pre min) then X else pre min;
   max = X -> if (X > pre max) then X else pre max;
tel
```

⇒ Recursive flows

Correct recursive definition

- The sequence can be built step by step
- Example: `alt = false -> pre alt`
  - $\texttt{alt}_0 = \texttt{false}_0 = \mathit{false}$
  - $\texttt{alt}_1 = (\texttt{not pre alt})_1 = \neg\, \texttt{alt}_0 = \mathit{true}$
  - $\texttt{alt}_2 = (\texttt{not pre alt})_2 = \neg\, \texttt{alt}_1 = \mathit{false}$
  - etc
- Counter-example: `X = 1/(2-X)`
  - Unique solution `X = 1`
  - Not computable: rejected
- Sufficient condition: no combinational loop

Correct recursive definition (cont'd)

- Problem:

  ```
  X = if C then Y else A;
  Y = if C then B else X;
  ```

  - Syntactic loop,
  - but not semantics loop:

    ```
    Y = X = if C then B else A;
    ```
- Choice in Lustre: syntactic loops are rejected
- Remarks:
  - general problem of synchronous approach
  - related to the ***causality*** problem

Exercises

Initializations

- Define a flow :

  $\texttt{P} \equiv \mathit{false}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}, \mathit{false}, \mathit{true}...$
- Define a flow :

  $\texttt{F} \equiv 1, 1, 2, 3, 5, 8, 13 \ldots$

Counters

- Write a node:

  `node Count(X : bool) returns (C : int);`

  such that $\texttt{C}_t$ = number of occurrences of `X`
- Write a node:

  `node CountR(X, reset :  bool) returns (C : int);`

  such that output is 0 when reset occurs

Modularity

Reuse

- Each user defined node becomes an available operator
- Instantiation in a functional style
- Example:

```
node Tempo(X : bool; delay :  int)
returns (Y : bool);
var cpt :  int;
let
   cpt = CountR(true -> not pre Y, not X);
   Y = (cpt >= delay);
tel
```

What is: `A = CountR(true, true -> (pre A = 4))`?

Node with several outputs

- Left-hand list:

```
node MinMaxAverage(X : int) return (A : int);
var min, max :  int;
let
   A = Average(min, max);
   min, max = MinMax(X);
tel
```

- Or even simpler:

```
A = Average(MinMax(X));
```


Clocks

Sampling: `when` operator

- Used to define flows which are "slower" than inputs

X	4	1	-3	0	2	7	8
C	*true*	*false*	*false*	*true*	*true*	*false*	*true*
X when C	4			0	2		8

- When `C` is false, `X when C` does not exist
- One can operate only on flows with same clock
  Example: "`X + (X when C)`" is forbidden

Projection: `current` operator

- Projects a sampled flow on the faster clock

X	4	1	-3	0	2	7	8
C	*true*	*false*	*false*	*true*	*true*	*false*	*true*
Y = X when C	4			0	2		8
Z = current(Y)	4	4	4	0	2	2	8

N.B. `current(X when C)` $\neq$ `X`

Initialization problem (classical error)

`X`	4	-3	0	2	7	8
`C`	*false*	*false*	*true*	*true*	*false*	*false*
`current(X when C)`	nil	nil	0	2	2	2

- Trick: sampling with initially true clocks

`C1 = true -> C`	*true*	*false*	*true*	*true*	*false*	*true*
`current(X when C1)`	4	4	0	2	2	8

- Another solution: enforce a "default" value

```
E = if C then current(X when C) else (dflt -> pre E);
```


Nodes and clocks

- Clock of a node instance = clock of its effective inputs
- Sampling inputs = enforce the whole instance to run slower
- In particular, sampling inputs $\neq$ sampling output:

`C`	*true*	*true*	*false*	*false*	*true*	*false*	*true*	*false*
`Count(true when C)`	1	2			3		4	
`Count(true) when C`	1	2			5		7	

Oversampling

- Sampling an already sampled flow
- `X when C` correct $\Leftrightarrow$ `X` and `C` are on the same clock
- `current` projects on the immediately faster clock

`X`	4	1	-3	0	2	7	8	13
`Y`	*true*	*false*	*true*	*true*	*true*	*false*	*false*	*true*
`C`	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*true*
`Z = X when C`	4	1		0	2		8	13
`H = Y when C`	*true*	*false*		*true*	*true*		*false*	*true*
`T = Z when H`	4			0	2			13
`current T`	4	4		0	2		2	13

Clock verification

- Rules to check clocks within a node:
  - The fastest clock is called the ***base clock***
  - Constants are on the ***base clock***
  - The default for inputs is to be on ***base clock***
  - Clock(`X when C`) = `C`
  - Clock(`current X`) = Clock(Clock(`X`))
  - Clock(`X op Y`) = C iff C = Clock(`X`) = Clock(`Y`)
- Moreover:
  - Some inputs may be sampled by other inputs
  - Output's clocks must be visible in the interface (i.e. either base, input or another output)

Static verification of clocks

- Similar to type-checking (no inference)
- variables must be declared with their clocks (as a consequence clocks are named)
- clock equivalence = same variable name

Multi-clock node: full example

```
node MultiClock(
   X, Y : int; C : bool;   ← base clock
   (Z : int) when C   ← input sampled on input
) returns (
   (S : int) when C   ← output sampled on input
);
var (H : bool) when C;
   (U : int) when H;
let
   H = (true when C) -> ((X + Y) when C) < Z;
   U = (Z when H) -> Average(Z when H, pre U);
   S = current(U);
tel
```

Introduction to Esterel

and its Semantics

Introductory example: a speedometer

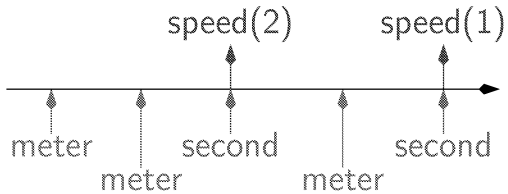
- receives signals Second and Meter
- each Second emit a signal Speed carrying the number of Meters received during the last Second


```
module Speedometer:
input Second, Meter ;
output Speed : integer in
    loop    var Distance := 0 : integer in
                abort
                    every Meter do Distance := Distance+1
                    end every
                when Second ;
                emit Speed(Distance)
            end var
    end loop
end module
```


An Esterel process communicates with its environment by means of (pure or valued) signals.

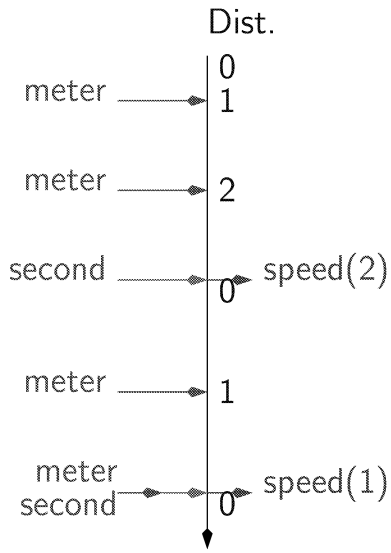
Its behavior is a sequence of reactions, each of which triggered by input signals

Ex.: a behavior of the speedometer



In a reaction, a signal is either present or absent (instantaneous broadcast)

```
module Speedometer:
input Second, Meter ;
output Speed : integer in
  loop
    var Distance := 0 : integer in
      abort
        every Meter do
          Distance := Distance+1
        end every
      when Second ;
      emit Speed(Distance)
    end var
  end loop
end module
```




```
module Speedometer:
input Second, Meter ;
output Speed : integer in
  loop
    var Distance := 0 : integer in
      abort
        every Meter do
          Distance := Distance+1
        end every
      when Second ;
      emit Speed(Distance)
    end var
  end loop
end module
```

loop Stat end

abort Stat when Occ

every Occ do Stat end

```
  =  await Occ ;
     loop
        abort
           Stat ; halt
        when Occ
     end
```

await Occ

= abort halt when Occ

Signals , Events, and occurrences

Event = set of present signals

A reaction consists of adding signals to an input event.

tick is a special signal which belongs to all events

If S is a valued signal, ?S refers to the value carried by its last occurrence

Occurrences:

In statements like await S, abort ... when S, the considered occurrence of S is in the strict future.

but you can write also await immediate S, ...

An occurrence may also consist of a number of signal occurrences:

every 60 SEC do emit MIN end

Kernel language

nothing, halt

v := exp

stat ; stat

if exp then stat else stat

loop stat end

trap id in stat

exit id

var var_decls in stat

signal signal_decls in stat

[stat || stat]

emit S , emit S(exp)

present S then stat1 else stat2

abort stat when S

Some derived statements

await S (= abort halt when S)

weak abort stat when S

(= trap T in [stat; exit T || await S; exit T] end)

pause (= await tick)

abort stat1 when S timeout stat2 end

```
(= trap T in
      abort stat1; exit T when S;
      stat2
   end                                  )
```

every S do stat

```
(=    await Occ ;
      loop
      abort
            Stat ; halt
         when Occ
      end
```


Example 1:

Parallel composition and instantaneous broadcast

```
input Second;
output Minute, Hour;
[
    every 60 Second do emit Minute end
||
    every 60 Minute do emit Hour end
]
```

Exercise: emit O as soon as both A and B have occurred.
Restart on any occurrence of R.

Structure nesting and priorities

Example 2: Mouse Click
receives pure signals click and hsec.

emits double whenever two clicks happen within d hsec, and single whenever a click is not followed by a second click within that delay.


```
module mouse: constant d: integer;
              input click, hsec;
              output single, double;
   loop    await click;
           abort   await click; emit double
           when d hsec
           timeout emit single end
   end
end
```

slightly wrong when simultaneous “click” and “d hsec”

Correct solution:

```
module mouse: constant d: integer;
              input click, hsec;
              output single, double;
   loop    await click;
           abort   await d hsec; emit single
           when click
           timeout emit double end
   end
end
```

```
module mouse:
    constant d: integer;
    input click, hsec;
    output single, double;
loop
    await click;
    abort
        await click; emit double
    when d hsec
    timeout emit single end
end
end
```

```
module mouse:
    constant d: integer;
    input click, hsec;
    output single, double;
loop
    await click;
    abort
        await d hsec; emit single
    when click
    timeout emit double end
end
end
```


Exercise: Reflex game

The player puts a coin in the machine
After a random delay, the machine switches on the go lamp
The player should press the stop button as soon as possible
Then the machine displays the time (in ms.) elapsed between go and stop. The go is switched off, the game_over lamp is switched on, and a new game can start
Exception cases:

- the player presses the stop before go (cheating! ring the bell and end the game)
- the player does not press stop within limit_time ms. after the go lamp is on (abandon, end the game)

Initially, only the game_over in on, and the machine displays 0.

Reflex game: declarations

```
module REFLEX_GAME :
constant limit_time : integer;
function RANDOM() : integer;
input MS, COIN, STOP;
output DISPLAY(integer),
       GO_ON, GO_OFF,
       GameOver_ON, GameOver_OFF,
       RingBell;
```


The story of Esterel

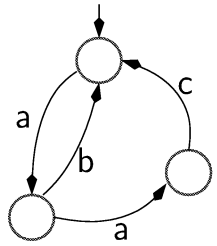
- developped since early 80th at CMA/ENSMP and Inria [G. Berry]
- now equipped with a graphical syntax: SyncCharts [Ch. Andre, 1996]
- commercialized by Esterel-Technologies
- several sources, many users
- see www.esterel.org

SyncCharts: a graphical language based on Esterel [André96]

Automata are very useful for describing control

The best way for describing an automaton is by drawing it

(not always easy to specify in Esterel)

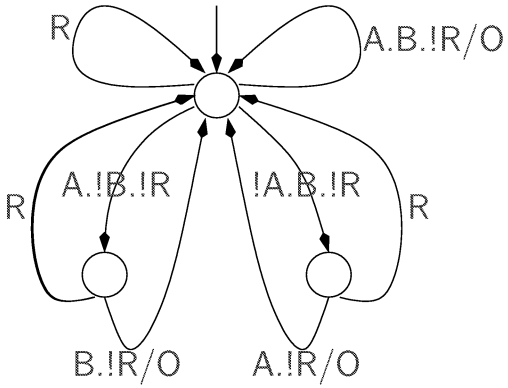


Graphical, automata-based language: Needs

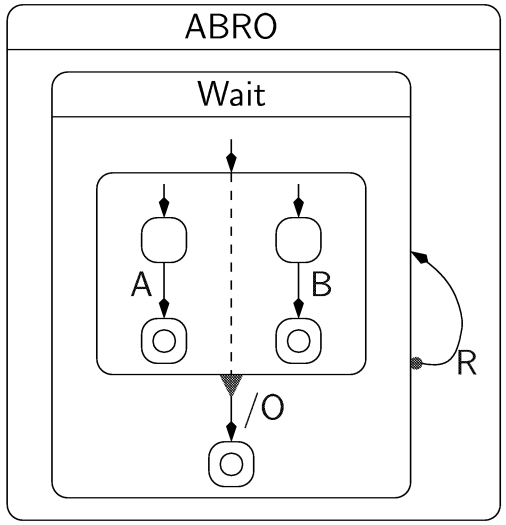
- Structure (cf. Statecharts: parallel and hierarchical composition)
- Avoid text (texts and drawings don't merge so well)
- Small number of graphical primitives
- (Of course!) Precise semantics

Example: ABRO as an automaton

emit O as soon as both A and B have occurred.
Restart on any occurrence of R.



ABRO in SyncCharts

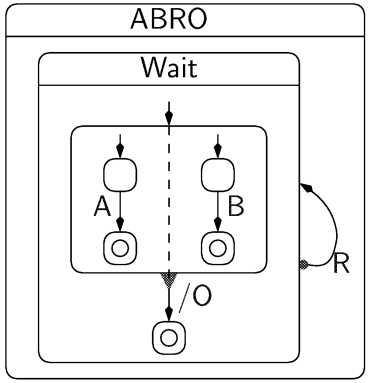


Advantages

Better structure

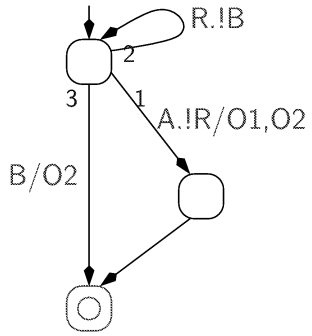
Each signal appears once

Waiting also for C doesn't
complexify too much
(linear increase)



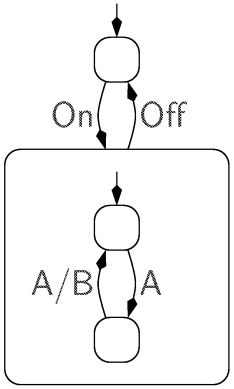
SyncCharts: Basic constructs (1)

Automata with
(input events)/(output signals)
on the transitions,
possibly final states,
possible priorities on transitions.



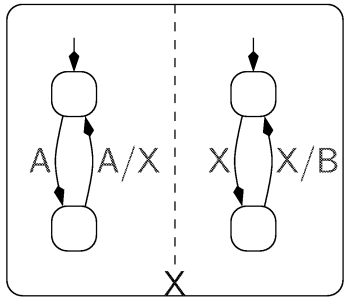
SyncCharts: Basic constructs (2)

Hierarchical composition



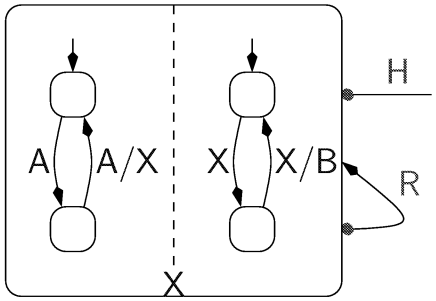
SyncCharts: Basic constructs (3)

Parallel composition, local signal



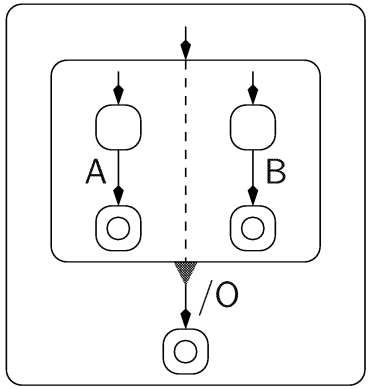
SyncCharts: Basic constructs (4)

Inhibition and inhibiting transitions

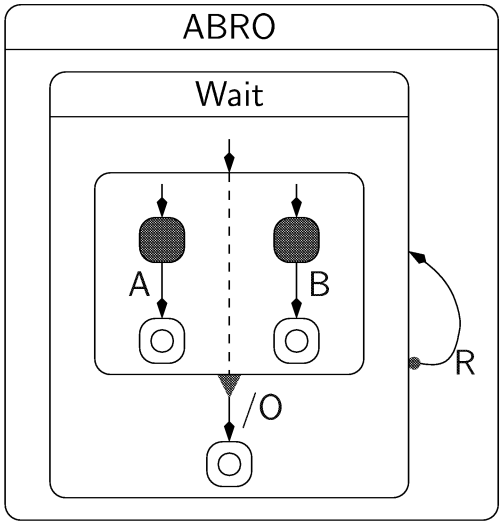


SyncCharts: Basic constructs (5)

Transitions on termination

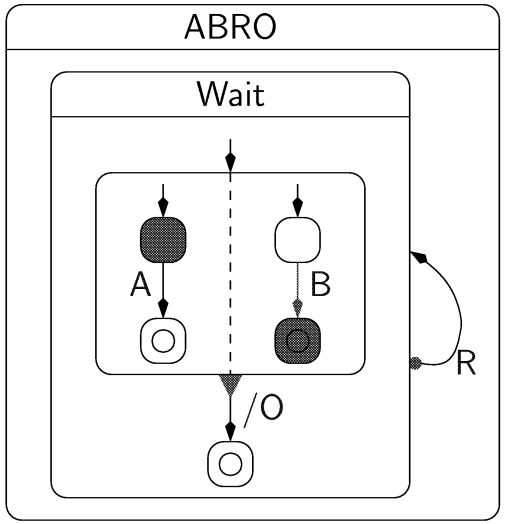


ABRO simulation



Initial

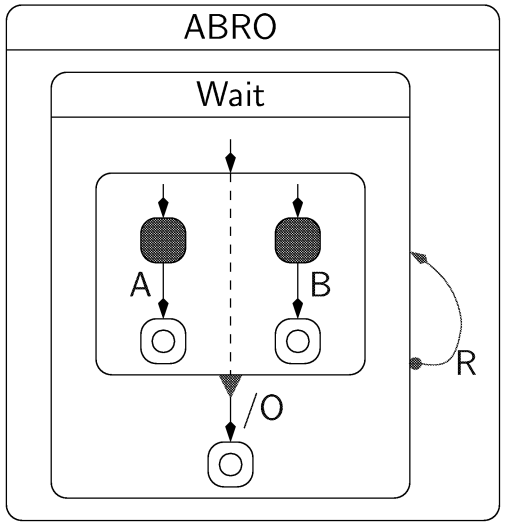
ABRO simulation



Present signal(s)

B

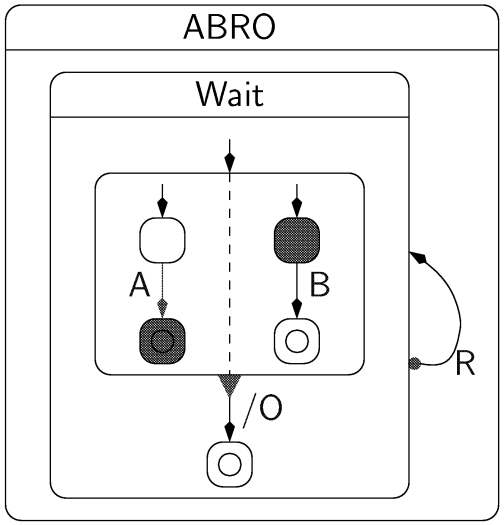
ABRO simulation



Present signal(s)

A, R

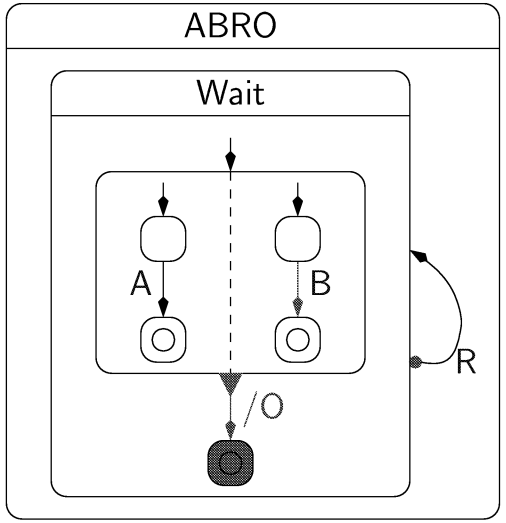
ABRO simulation



Present signal(s)

A

ABRO simulation



Present signal(s)

B

Advantages of SyncCharts over Esterel

Graphical (people sometimes like it ?!)

Easy automata description

Nesting of interrupts/preemption/exceptions more readable than with textual nesting

Compilation of Synchronous Languages

Static semantics (causality)

Code generation

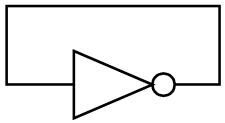
- Single loop for Lustre
- Automata
- From Esterel to Lustre

Causality analysis

Some programs don't make sense
(∼ combinational loops in circuits)

In Lustre

x = not x;



no behavior

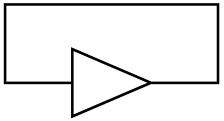
In Esterel

```
module P1: output x ;
    present x else emit x
    end present
end module
```


Other bad case:

In Lustre

x = x

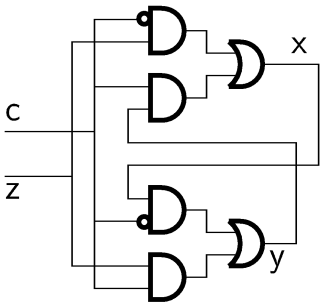


many behaviors
(nondetermism?)

In Esterel

```
module P1:
output x ;
    present x
        then emit x
    end present
end module
```


but some loops do make sense !



In Lustre

x = if c then y else z ;
y = if c then z else x ;

In Esterel

```
module P1:
input c, z ; output x, y ;
    present c
    [   present y then emit x end
    ||  present z then emit y end
    ]
    else
    [   present z then emit x end
    ||  present x then emit y end
    ]
    end present
end module
```

Source of all the problems with the semantics of Statecharts and Sequential Function Charts (Grafcet)

Where does the problem come from ?
The result of a reaction is a fixpoint of a function which is not necessarily increasing (because of the negation, or the reaction to absence)

Solutions :

- forbid loops (Lustre)
- forbid instantaneous reaction to absence (SL [Boussinot])
- one and only one behavior in classical logic
- one and only one behavior in constructive logic (Esterel V5)

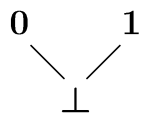
A strange Lustre program:

```
x = x;
y = x and not y;
```

only one behavior in classical logic
no behavior in constructive logic!

Computing in constructive logic:

- Use Scott's Boolean domain with



$$\mathbf{0} \wedge \bot = \bot \wedge \mathbf{0} = \mathbf{0}$$

$$\mathbf{1} \vee \bot = \bot \vee \mathbf{1} = \mathbf{1}$$

- or use dual rail encoding :

$$\boldsymbol{x_0 x_1} \quad \text{with} \quad \begin{cases} \boldsymbol{x_0 = 1} & \text{iff } \boldsymbol{x} \text{ surely } 0 \\ \boldsymbol{x_1 = 1} & \text{iff } \boldsymbol{x} \text{ surely } 1 \end{cases}$$

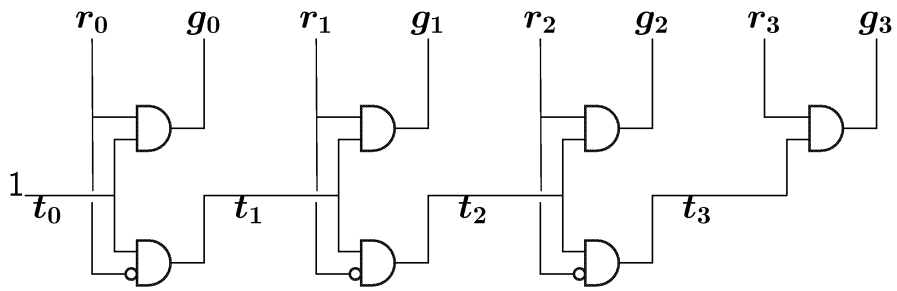
Example:

x = x;	$\perp$	$\perp$
y = x and not y;	$\perp$	$\perp$

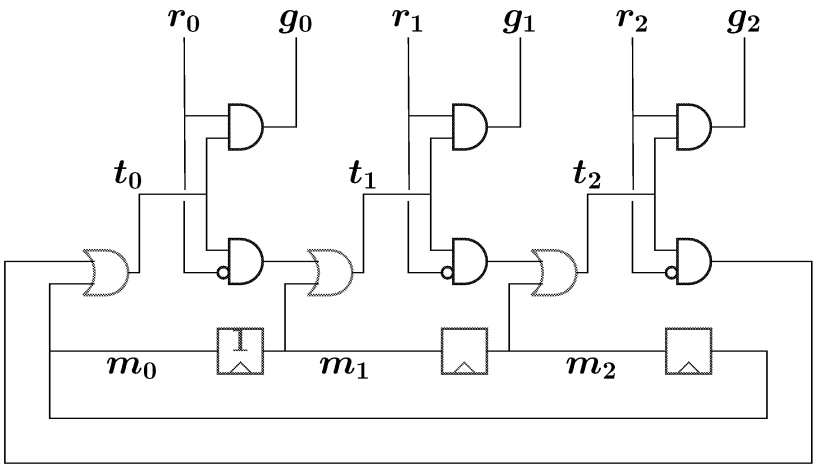
Example : McMillan/DeSimone's bus arbiter

n units connected to a bus. At each clock tick, some of them can ask for bus access for this tick. At most one may be granted, and the allocation should be fair.

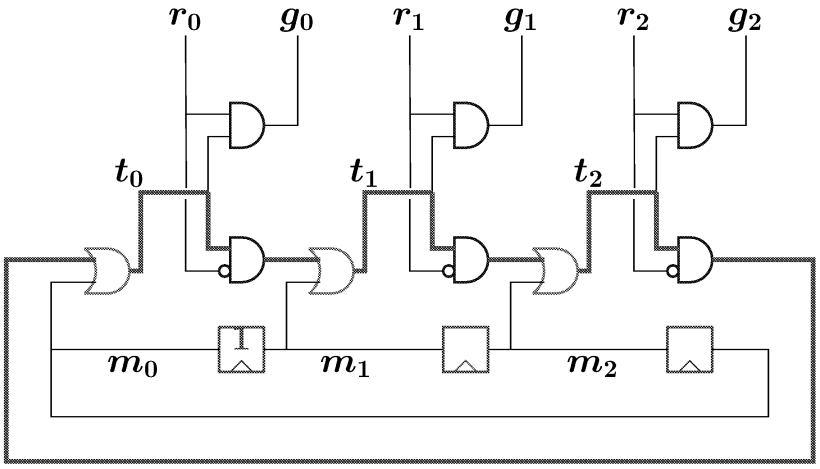
Basic idea: travelling token:



Fairness: change the master at each tick



Combinational loop!



$$t_0 = m_0 + t_2.\overline{r_2} \qquad t_1 = m_1 + t_0.\overline{r_0}$$
$$t_2 = m_2 + t_1.\overline{r_1}$$

but...

- if $m_0 = 1$, $t_0 = 1$,
  $$t_1 = m_1 + \overline{r_0}\,, \quad t_2 = m_2 + (m_1 + \overline{r_0}).\overline{r_1}$$
- if $m_1 = 1$, $t_1 = 1$,
  $$t_2 = m_2 + \overline{r_1}\,, \quad t_0 = m_0 + (m_2 + \overline{r_2}).\overline{r_0}$$
- if $m_2 = 1$, $t_2 = 1$,
  $$t_0 = m_0 + \overline{r_2}\,, \quad t_1 = m_1 + (m_0 + \overline{r_0}).\overline{r_2}$$

remains to show that $m_0 \vee m_1 \vee m_2$ always true

Other example:

constructive operational semantics of SyncCharts

To compute the reaction to an input event

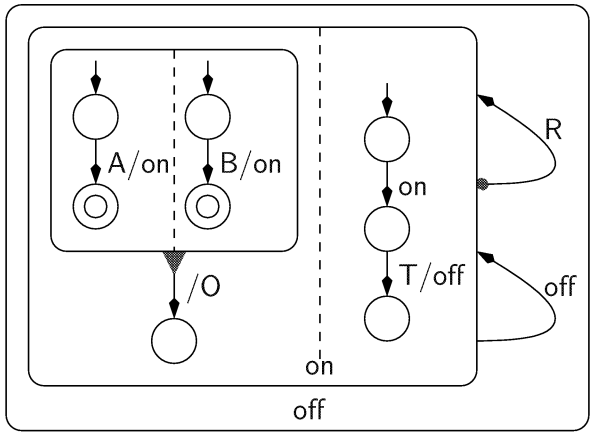
- give their value to all input signals
- give the value $\perp$ to all other signals
- evaluate the top level

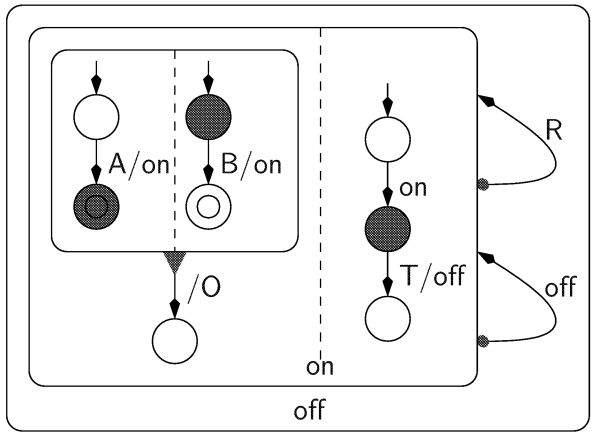
To evaluate a "box"

- if there are inhibiting transitions sourced in the box
  - if the current signal values validate a prioritary inhibiting transition sourced in the box, fire the transition (ignoring the content of the box)
  - if the guard of all prioritary inhibiting transitions evaluate to $\perp$, postpone the treatment
  - otherwise go inside the box (i.e., evaluate all parallel components inside)
- otherwise go inside the box

- after termination of the inside reaction, if there are weak transitions sourced in the box
  - if the current signal values validate a prioritary weak transition sourced in the box, fire the transition (ignoring the content of the box)
  - if the guard of all prioritary inhibiting transitions evaluate to $\perp$, postpone the treatment
  - otherwise terminate
- If the evaluation blocks, or if some output signals keep the value $\perp$, there is a causality problem.

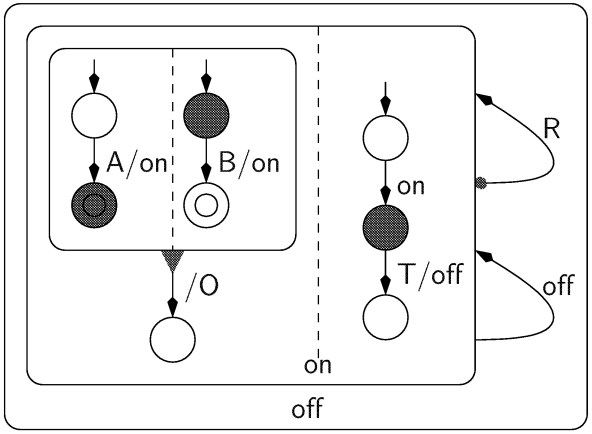
Example: Emit O whenever both A and B occur between two occurrences of T. Reset when R.





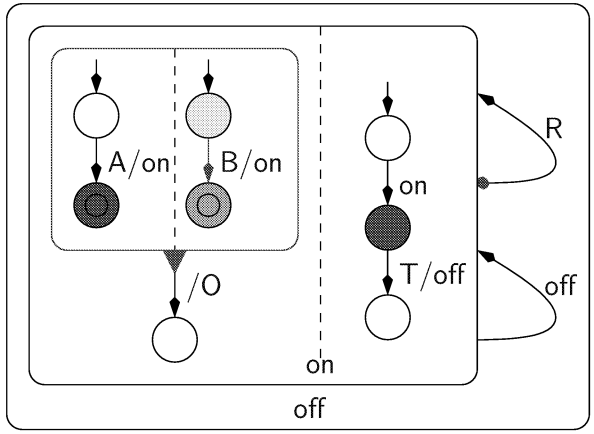
A=0
B=1
R=0
T=1

O=⊥
on=⊥
off=⊥



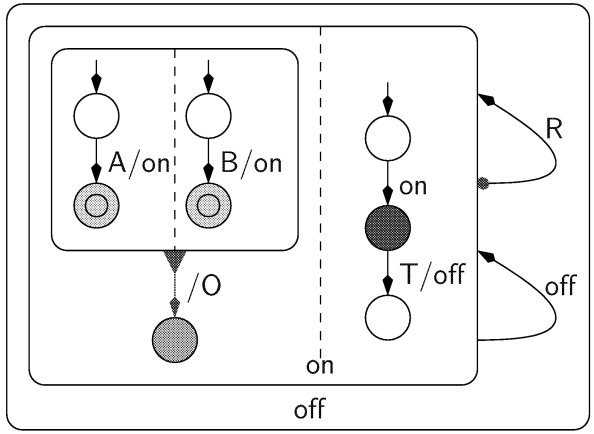
A=0
B=1
R=0
T=1

O=⊥
on=⊥
off=⊥



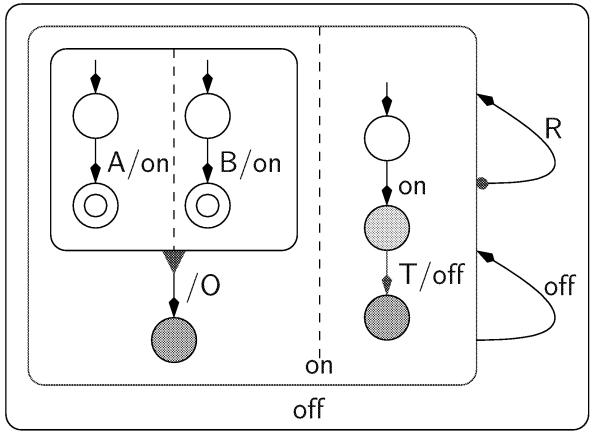
A=0
B=1
R=0
T=1

O=⊥
on=1
off=⊥



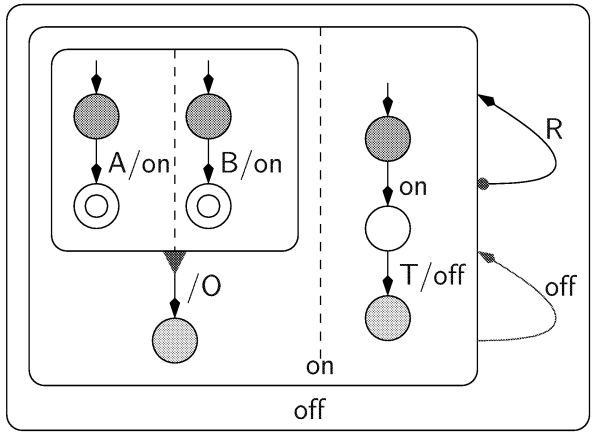
A=0
B=1
R=0
T=1

O=1
on=1
off=⊥



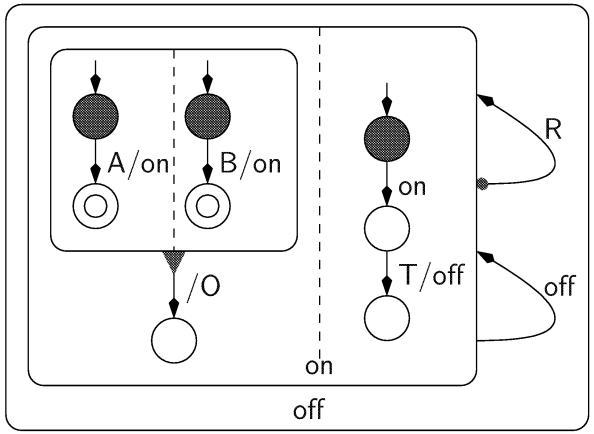
A=0
B=1
R=0
T=1

O=1
on=1
off=1



A=0
B=1
R=0
T=1

O=1
on=1
off=1



A=0
B=1
R=0
T=1

O=1
on=1
off=1

Compilation: sequential code generation

Single loop (implicit automaton)

obvious for data-flow programs:

- sort the variables according to their dependences
- choose a suitable set of memories

Example

```
x = 0 -> if edge then (pre(x) + y)
         else pre(x) ;
edge = c -> (c and not pre(c)) ;
```

```
_init := true ;
foreach step do
    read(c, y) ;
    if _init then
        { edge := c ; x := 0}
    else
        edge := c and not _pc ;
        if edge then x := x+y ;
    _init := false ; _pc := c ;
end for
```


Sorting variable computation, and choice of memories

$x > y$ iff x appears outside of any "pre" in the definition of y (x must be computed after y)

$>$ is a partial order, if there is no causality error

$x \succ y$ iff pre(y) appears in the definition of x (x should be computed before y, i.e., from the previous value of y)

Sorting variable computation, and choice of memories (2)

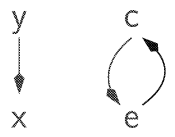
Consider the graph of the relation "$>$"$\cup$"$\succ$".

Remove $\succ$ edges to cut all loops (if any). The resulting graph is a partial order.

Whenever an edge $x \succ y$ is removed, a buffer py must be introduced.

Example

```
x = 0 -> if edge then (pre(x) + y)
         else pre(x) ;
edge = c -> (c and not pre(c)) ;
```



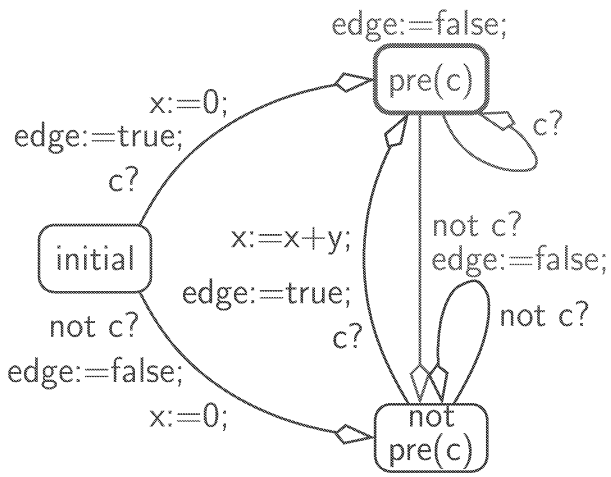

```
_init := true ;
foreach step do
    read(c, y) ;
    if _init then
        { edge := c ; x := 0}
    else
        edge := c and not _pc ;
        if edge then x := x+y ;
    _init := false ; _pc := c ;
end for
```


Explicit control automaton

- first way of compiling Esterel
- also applied to data-flow languages
- basis for verification methods
  (notion of control automaton)

An example in Lustre

```
x = 0 -> if edge then (pre(x) + y)
         else pre(x) ;
edge = c -> (c and not pre(c)) ;
```

An example in Esterel

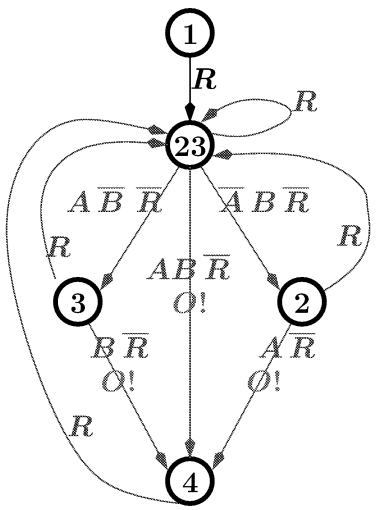
```
every R do
    [ await A || await B ] ;
    emit O ;
end every
```


In kernel language:

```
abort halt¹ when R;
loop
    abort
        [   abort halt² when A
        ||
            abort halt³ when B
        ];
        emit O; halt⁴
    when R
end
```



```
abort halt¹ when R;
loop
    abort
        [   abort halt² when A
        ||
            abort halt³ when B
        ];
        emit O; halt⁴
    when R
end
```



Explicit automaton

- very efficient code
- possible exponential growth of the code
  (less than for asynchronous languages)

Esterel is now compiled also into an implicit automaton (single loop)

- much more tricky!
- consequence of silicon compiling
- no explosion of the code size

Principles of the translation of (pure) Esterel into implicit automata

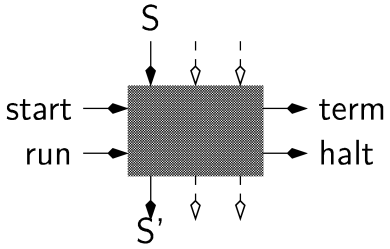
(or Esterel → Lustre)

Each statement is translated into a node with the same interface:

node N

(start, run, S,... : bool)

returns (term, halt, S',...)



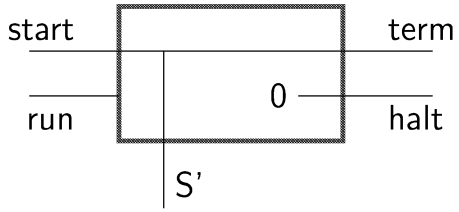
Examples:

emit S':

term = start

halt = false;

S' = start;



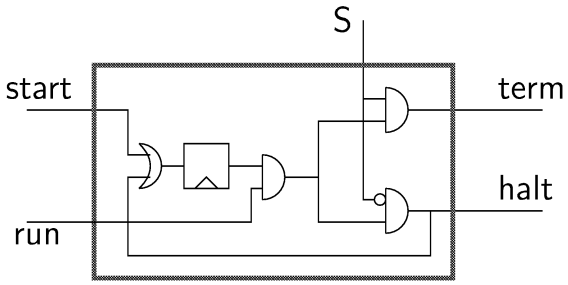
Examples:

await S:

term = run and w and S;

halt = run and w and not S;

w = false ->pre(start or halt);



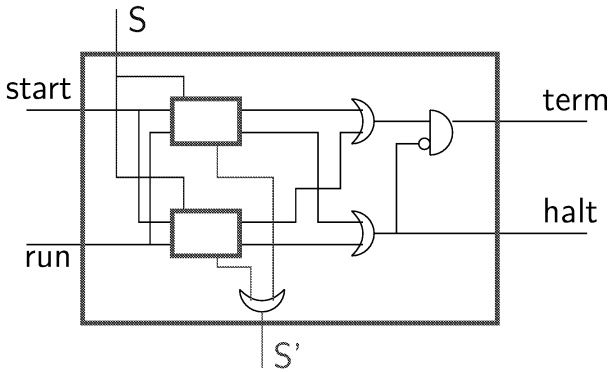
stat1 || stat2: (term1, halt1, S'1) = Stat1(start,run,...);

(term2, halt2, S'2) = Stat2(start,run,...);

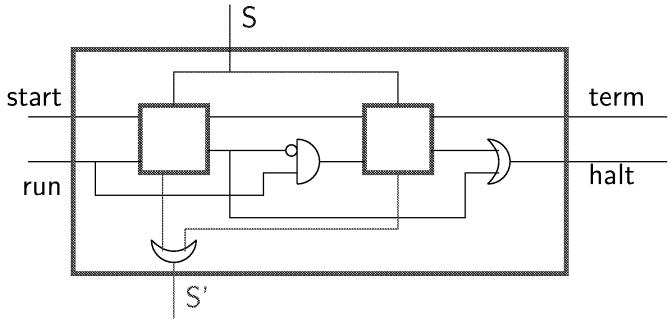
halt = halt1 or halt2;

term = (term1 or term2) and not halt;

S' = S'1 or S'2;

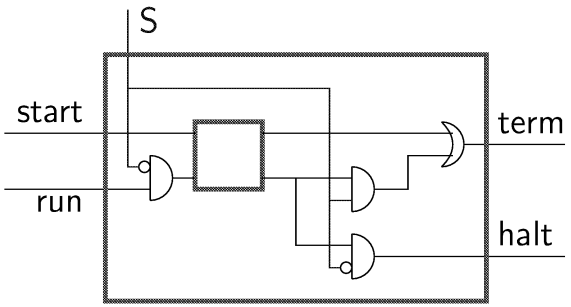


stat1; stat2: (term1, halt1, S'1) = Stat1(start,run,...);
(term2, halt2, S'2) = Stat2(term1,run and not halt1,...);
halt = halt1 or halt2;
term = term2;
S' = S'1 or S'2;

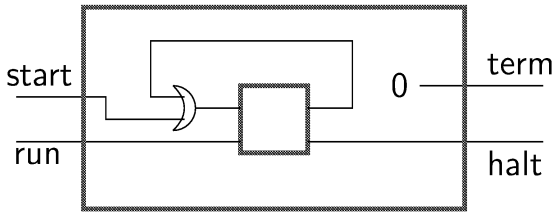


abort stat when S:

(term1, halt1, S',...) = Stat(start,run and not S,...);
halt = halt1 and not S;
term = term1 or (halt1 and S);



loop stat: (term1, halt1, S',...) = Stat(start or term1, run,...);
term = false;
halt = halt1;



A last compilation problem: "Reincarnation" in Esterel

```
loop
    signal S in
        [
            await T; emit S
        ||
            present S then emit O
        ]
    end
end
```

Solution1 : Code replication

```
loop
    signal S1 in [   await T; emit S1
                 || present S1 then emit O
                 ]
    end
    signal S2 in [   await T; emit S2
                 || present S2 then emit O
                 ]
    end
end
```


Solution 2 : Distinguish between

- the "surface" of the body, i.e., a piece of code corresponding to its first reaction,
- and its "depth", i.e., a piece of code corresponding to other reactions.

surf(await T) = pause
depth(await T) = await immediate T
surf(present S then emit O) = present S then emit O
depth(present S then emit O) = nothing


```
loop
    signal S1 in [   pause
                 || present S1 then emit O
                 ]
    end
    signal S2 in [   await immediate T; emit S2
                 || nothing
                 ]
    end
end
```

Verification of Synchronous Programs

Pascal Raymond

Vérimag/CNRS, Grenoble

Introduction

Functional verification

- Does the program compute the right outputs?
- Expected relation among time between inputs and outputs: temporal properties

Intuitive partition of temporal properties

- Safety: something (bad) never happens
- Liveness: something (good) eventually happens

Example: the beacon counter in a train

- Counts the difference between beacons and seconds
- Decides whether the train is late, early or ontime
- Hysteresis to avoid oscillations

```
node b(sec,bea:  bool) returns (ontime,late, early:  bool);
var diff int
let
   diff = (0 -> pre diff) + (if bea then 1 else 0) +
         (if sec then -1 else 0);
   early = (true -> pre ontime) and (diff > 3)
         or (false -> pre early) and (diff > 1);
   late = (true -> pre ontime) and (diff < -3)
         or (false -> pre late) and (diff < -1);
   ontime = not (early or late);
tel
```


Some properties

- It's impossible to be late and early
- It's impossible to directly pass from late to early
- It's impossible to remain late only one instant
- If the train stops, it will eventually get late

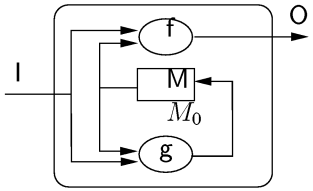
The 3 first ones are obviously safety, while the one is a typical liveness: it refers to unbounded future

Implicit state machines

Functionality of synchronous program

A synch. prog, is a function from infinite seq. of inputs to infinite seq. of outputs: $\mathcal{P}(I_0, I_1, I_2, \cdots) = O_0, O_1, O_2, \cdots$ defined via a well initialized internal memory

- Inputs I, outputs O
- Memory M, initial value M_0
- Output function: $O_t = f(I_t, M_t)$
- Transition function: $M_{t+1} = g(I_t, M_t)$



Finally, $\mathcal{P}(I_0, I_1, I_2, \cdots) = O_0, O_1, O_2, \cdots$ iff
$\exists M_0, M_1, M_2 \cdots$ s.t. $\forall t\ O_t = f(I_t, M_t)$ and $M_{t+1} = g(I_t, M_t)$

Common model for synchronous programs

- Obvious for Lustre (memory = `pre` operators)
- Less obvious, but still true, for Esterel/SyncCharts (cf. compilation)

Implicit vs explicit

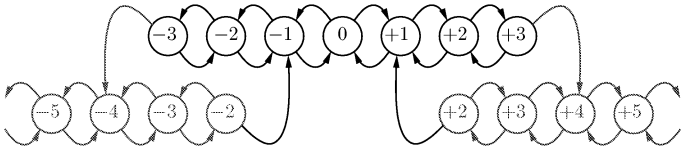
An ISM is equivalent to an explicit state/transition system:

- States are all possible values of M: $Q = \mathcal{D}(M)$
- Transition $q \xrightarrow{i/o} q'$ iff $q' = g(i, q)$ and $o = f(i, q)$
- In general: infinite state machine (numerical)

Example: beacon counter

- I = {sec, bea} O = {late, ontime, early}
- A memory for each "`-> pre`" expression,
  (e.g. Plate for "`false -> pre late`"):
  M = {Plate, Pontime, Pearly, Pdiff}
  with M_0 = (***false***, ***true***, ***false***, 0)
- Functions directly given by the Lustre equations

A small part of the explicit automaton:



Conservative Abstraction

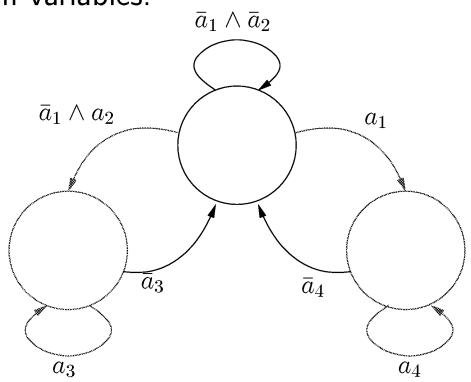
Model and verification

The explicit automaton is the set of behavior,
so exploring the automaton is checking the program
Problem: The automaton may be infinite, or at least enormous, it is impossible to explore it
Idea: work on a finite (not too big) abstraction of the program
N.B. the abstraction must conserve at least some properties (otherwise it's useless)

Example

Abstraction of numerical comparisons in the beacon counter, they become "free" boolean variables:

- a_1 for `diff > 3`
- a_2 for `diff > 1`
- a_3 for `diff < -3`
- a_4 for `diff < -1`



Conserved properties

- It's impossible to be late and early (safety)
- It's impossible to directly pass from late to early (safety)

Lost properties

- It's impossible to remain late only one instant (safety)
- If the train stops, it will eventually get late (liveness)

More serious: introduced property

- It's possible to remain late only one instant (liveness): true on the abstraction, false on the real program !

⇒ Important to precisely know what is preserved by the abstraction

Abstraction and safety

- Finite abstraction is a special case of over-approximation
- Anything which is impossible in the abstraction is impossible on the program
- The counterpart is (in general) false

⇒ safeties are preserved or lost, but never introduced

As a consequence, when checking a safety on the abstraction:

- the verification succeeds ⇒ property satisfied
- the verification fails ⇒ inconclusive (it may be a ***false negative***)

Expressing properties

Liveness requires complex formalisms (temporal logics)

Safety can be programmed ⇒ observers

Observer

- Observe the inputs and outputs of the program
- Outputs "ok" as long as the behavior meets the property (or, equivalently, outputs "ko" when the behavior violate the property)

Example (in Lustre)

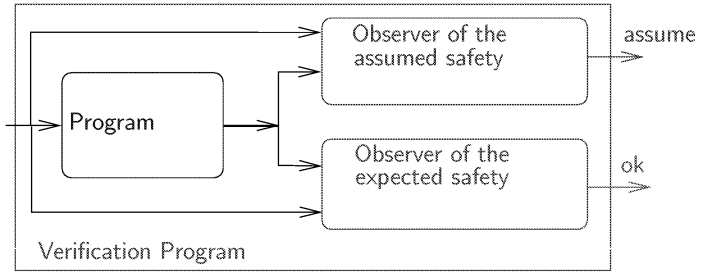
- It's impossible to be late and early:
  `ok = not(late and early);`
- It's impossible to directly pass from late to early:
  `ok = true -> (not early and pre late);`
- It's impossible to remain late only one instant:
  ```
  Plate = false -> pre late;
  PPlate = false -> pre Plate;
  ok = not(not late and Plate and not PPlate);
  ```


Assumptions

Convenient to split property into assumption/conclusion:
"if the train keeps the right speed, it remains on time"
property is simply `ok = ontime`, assumption can be:

- naive: `assume = (sec = bea);`
- more sophisticated, bea and sec alternate:
  ```
  SF = switch(sec and not bea, bea and not sec);
  BF = switch(bea and not sec, sec and not bea);
  assume = (SF => not sec) and (BF => not bea);
  ```
  with:
  ```
  node switch(on, off : bool) returns (s: bool);
  let s = false -> pre(if s then not off else on);
  tel
  ```


General scheme



- We suppose provided such a verification program
- Goal: if assume remains indefinitely true, then ok remains indefinitely true: ***(always*** `assume`***)*** $\Rightarrow$ ***(always*** `ok`***)***
- Note: it is NOT a "regular" safety, so in a first step, we approximate it by: ***always ((once not*** `assume`***) or*** `ok`***)*** (the problem will be explained later)

Proving properties

Abstracted verification program

Special case of Boolean synchronous program with 2 "outputs"

- Free variables V, state variables S
- Initial state(s):
  Init $: \mathbb{B}^{|S|} \rightarrow \mathbb{B}$
- Transition functions:
  $g_k : \mathbb{B}^{|S|} \times \mathbb{B}^{|V|} \rightarrow \mathbb{B}$ for $k = 1 \cdots |S|$
- Assumption:
  $H : \mathbb{B}^{|V|} \rightarrow \mathbb{B}$
- Property:
  $\phi : \mathbb{B}^{|V|} \rightarrow \mathbb{B}$

(N.B. we identify predicates and sets)

Associated explicit automaton

We note $Q = \mathbb{B}^{|S|}$ the state space

We use "pre" and "post" functions:

- for $q \in Q$, $\mathsf{post}_H(q) = \{q'/\exists v \;\; q \xrightarrow{v} q' \;\wedge\; H(q,v)\}$
- for $X \subseteq Q$, $\mathsf{Post}_H(X) = \bigcup_{q \in X} \mathsf{post}_H(q)$
- for $q \in Q$, $\mathsf{pre}_H(q) = \{q'/\exists v \;\; q' \xrightarrow{v} q \;\wedge\; H(q',v)\}$
- for $X \subseteq Q$, $\mathsf{Pre}_H(X) = \bigcup_{q \in X} \mathsf{pre}_H(q)$

Significant state sets

- Initial state(s): $\mathsf{Acc}_0 = \{q/\mathsf{Init}(q)\}$
- Error states: $\mathsf{Err} \;= \{q/\exists v \quad H(q,v) \wedge \neg\phi(q,v)\}$
- Accessible states: $\mathsf{Acc} = \mu X \cdot (X = \mathsf{Init} \cup \mathsf{Post}_H(X))$
- Bad states: $\mathsf{Bad} = \mu X \cdot (X = \mathsf{Err} \cup \mathsf{Pre}_H(X))$

Goal

Naive: prove that $\mathsf{Acc} \cap \mathsf{Bad} = \emptyset$

No need to compute both Acc and Bad:

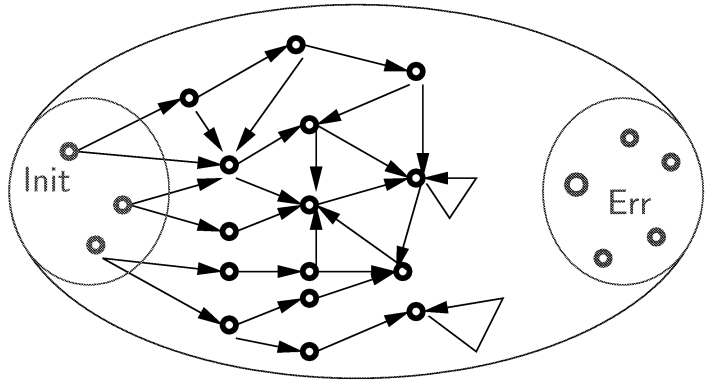
- prove that $\mathsf{Acc} \cap \mathsf{Bad}_0 = \emptyset$ (forward method)
- prove that $\mathsf{Bad} \cap \mathsf{Acc}_0 = \emptyset$ (backward method)

Remark: methods are non symmetric because of determinism

Enumerative (forward) algorithm

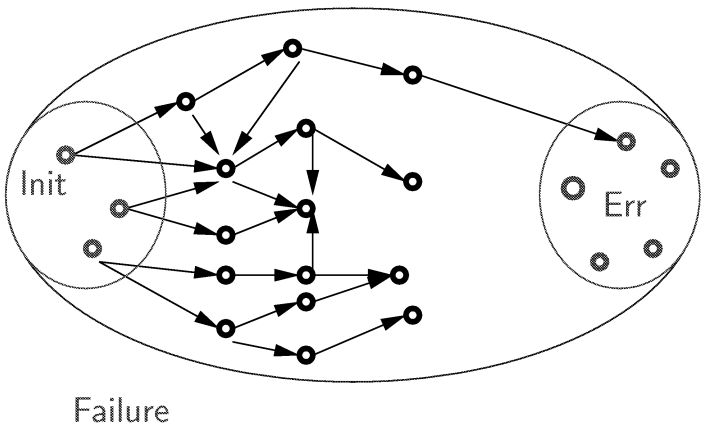
```
CurAcc := Init
Done := empty
while it exits q in CurAcc - Done do {
      (* q ∈ CurAcc \ Done *)
      for all q' in post_H(q) do {
          if q' in Bad0 then EXIT(failed)
          put q' in CurAcc
      }
      put q in Done
}
(* we have CurAcc = Done = Acc *)
EXIT(succeed)
```


Example of success (breadth first):



success

Example of failure (breadth first):



Notes on implementation

- depth first, breath first, or other
- compact encoding of states
- very costly:
  $|Acc|$ times the cost of post$_H$(q), with $|Acc| \sim 2^{|S|}$
- backward is even worse: pre$_H$(q) is more complex than post$_H$(q)
  (enumerative backward is never used in practice)

Big problem: computing pre$_H$(q)

- For a given q, find all v s.t. $H(q, v)$
- Typical decision problem (NP-complete)
- Naive method: try all $2^{|V|}$ possible values
- Need for non trivial, efficient decision procedure

$\Rightarrow$ Digression on efficient decision techniques

Decision techniques

Problem: let F be a formula on V, find all $v \in 2^{|V|}$ s.t. $F(v)$
Mainly two kind of solutions:

- Enumeration of the solutions, related to Sat-Solving, reference algo is Davis-Putnam
- Construction of the solution set, related to canonical form, reference method is Binary Decision Diagrams (BDD)

We study BDDs:

- Used with a certain success
- Address also the problem of state explosion
- Ad hoc algorithms: Symbolic Model Checking

Binary Decision Diagrams

Shannon decomposition

For any $f \in \mathbb{B}^n \to \mathbb{B}$:

$f(x, y, ..., z) = x.f(1, y, ..., z) + \bar{x}.f(0, y, ..., z)$

Let's define f_x and $f_{\bar{x}}$ in $\mathbb{B}^{n-1} \to \mathbb{B}$ by:

$f_x(y, ..., z) = f(1, y, ..., z)$

$f_{\bar{x}}(y, ..., z) = f(0, y, ..., z)$

For any f and any x, f_x and $f_{\bar{x}}$ are unique

Exercise: $f(x, y, z) = x.y + (y \oplus z)$ f_x, $f_{\bar{y}}$, f_z ?

$f_x = y + (y \oplus z)$

$f_{\bar{y}} = z$

$f_z = x.y + \bar{y} = x + \bar{y}$

Shannon tree

When applying recursively the S.D. on all variables, one obtains:

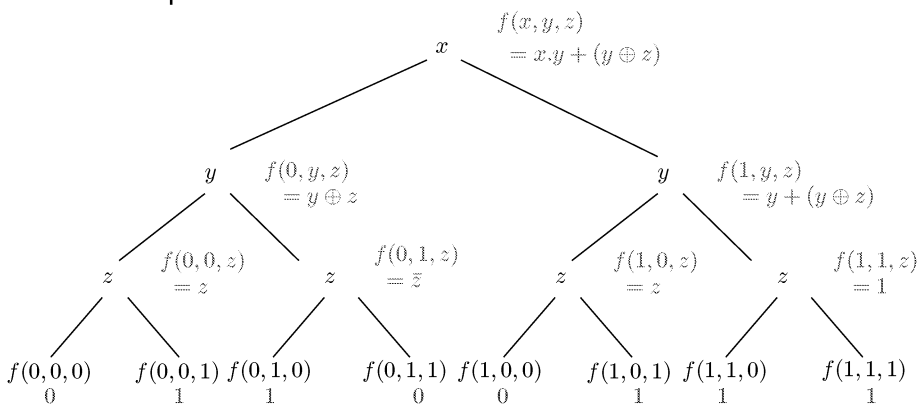
- 1 (the always-true function) or
- 0 (the always-false function)

Example, for $f = x.y + (y \oplus z)$:

- $f_{\bar{x}} = f(0, y, z) = y \oplus z$
- $f_{\bar{x}y} = f(0, 1, z) = \neg z$
- $f_{\bar{x}yz} = f(0, 1, 1) = 0$

Shannon tree: graphical representation of all the 2^n steps

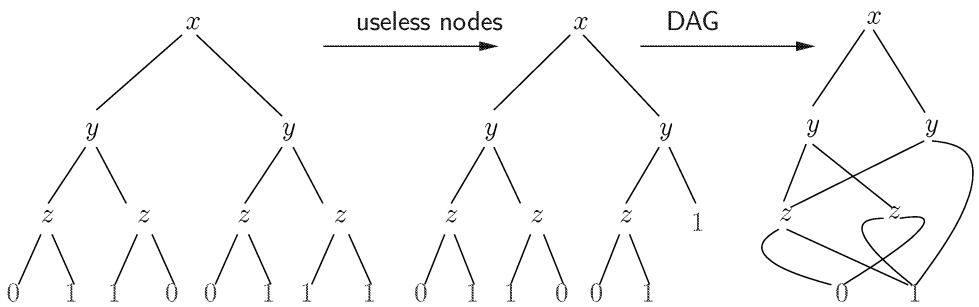
Example:



N.B. For a given variable ordering the tree is ***unique***

Binary Decision Diagram

- Concise representation of the Shannon tree
- No useless nodes (if x then g else g ⇔ g)
- Share common sub-graph (DAG)



N.B. For a given variable ordering the BDD is ***unique***

Formal definition

Set of variables V, with a total order $\preceq$

V extended with a max value:

$V^\star = V \cup \{\infty\}$ with $\forall x \in V \;\; x \prec \infty$

BDDs are defined, together with their "range" $rg : BDD \to V^\star$

- 1 is a BDD with $rg(1) = \infty$
- 0 is a BDD with $rg(0) = \infty$
- $\alpha = (x, l, h) \in V \times BDD \times BDD$ is a BDD with $rg(\alpha) = x$ iff $x \prec rg(h)$ et $x \prec rg(l)$

Implementation

Uniqueness of 0 and 1 "built-in"

Uniqueness of binary nodes guaranteed by hash-code

Creation of nodes made by $\mathcal{N}(x, \alpha, \beta)$, where α, β are BDDs

We note $\begin{array}{c} x \\ \alpha \quad \beta \end{array}$ for $\mathcal{N}(x, \alpha, \beta)$, and $\begin{array}{c} x \\ \alpha \quad \beta \end{array}$ for a BDD

- $\begin{array}{c} x \\ \alpha \quad \beta \end{array}$ = ERROR if $rg(\alpha) \prec x$ or $rg(\beta) \prec x$
- $\begin{array}{c} x \\ \alpha \quad \alpha \end{array} = \alpha$
- $\begin{array}{c} x \\ \alpha \quad \beta \end{array} = \begin{array}{c} x \\ \alpha \quad \beta \end{array}$ otherwise

Operations on BDDs

Negation

- $\neg 1 = 0$
- $\neg 0 = 1$
- $\neg \begin{array}{c} x \\ f_0 \quad f_1 \end{array} = \begin{array}{c} x \\ \neg f_0 \quad \neg f_1 \end{array}$

Binary operators

Property: any usual operator $\star$ (in $+$, $\cdot$, $\oplus$, $\Rightarrow$, $\Leftrightarrow$), distribute on Shannon decomposition:

$(x \cdot f_x + \bar{x} \cdot f_{\bar{x}}) \star (x \cdot g_x + \bar{x} \cdot g_{\bar{x}}) = x \cdot (f_x \star g_x) + \bar{x} \cdot (f_{\bar{x}} \star g_{\bar{x}})$

As a consequence, recursive rules are,

for $f = \begin{array}{c} x \\ f_0 \quad f_1 \end{array}$ et $g = \begin{array}{c} y \\ g_0 \quad g_1 \end{array}$:

- $f \star g = \begin{array}{c} x \\ f_0 \star g \quad f_1 \star g \end{array}$ if $x \prec y$ (balance)
- $f \star g = \begin{array}{c} y \\ f \star g_0 \quad f \star g_1 \end{array}$ if $y \prec x$ (balance)
- $f \star g = \begin{array}{c} x \\ f_0 \star g_0 \quad f_1 \star g_1 \end{array}$ if $x = y$

Terminal rules have priority, for instance:

- $(1 + \alpha) = (\alpha + 1) = 1$
  $(0 + \alpha) = (\alpha + 0) = \alpha$
- $(1 \cdot \alpha) = (\alpha \cdot 1) = \alpha$
  $(0 \cdot \alpha) = (\alpha \cdot 0) = 0$
- $\alpha \oplus \alpha = 0$
  $(0 \oplus \alpha) = (\alpha \oplus 0) = \alpha$
  $(1 \oplus \alpha) = (\alpha \oplus 1) = \neg\alpha$

Exercise: terminal rules for $\Rightarrow$?

Quantification

Exercise: implementation of $\exists v \;\; \alpha$?

Notes on complexity

- Cost of $\neg\alpha$: is linear w.r.t the size of α
- Cost of $\alpha \star \beta$: is in size α times size β
- Algebraic formula to BDD: exponential (worst case)
- Variable ordering is very important:
  $(x_1 \oplus x_2) \cdot (x_3 \oplus x_4) \cdot \cdots \cdot (x_{2n-1} \oplus x_{2n})$
  size in $O(n)$ for $x_1 \prec x_2 \prec x_3 \prec \cdots \prec x_{2n}$
  size in $O(2^n)$ for $x_1 \prec x_3 \prec \cdots \prec x_{2n-1} \prec x_2 \prec x_4 \prec \cdots \prec x_{2n}$

Lots of variants/implementations
$\Rightarrow$ a fundamental variant: Signed BDD

Signed BDD

Cost of negation

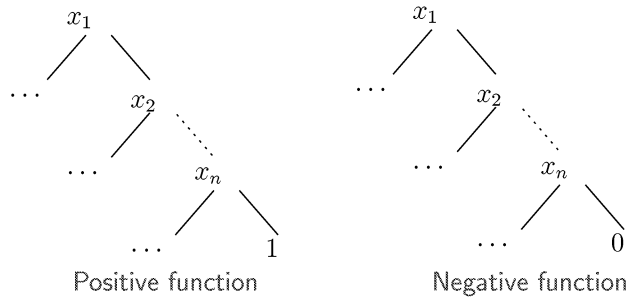
- BDDs for f and $\neg f$ are very similar: same structure, only leafs differ
- They don't share any node (costly in space)
- Computing $\neg$ is linear (costly in time)

Sharing structure

- Concretely represent only one of f or $\neg f$
- Define the other as the negation
- Problem: how to keep it canonical ?

Definition of positive functions

$f \in \mathbb{B}^n \to \mathbb{B}$ is positive iff $f(1, 1, \cdots, 1) = 1$



Idea: Nodes are reserved for positive functions, negative ones are defined by adding a ***sign*** flag

Definition of SBDD

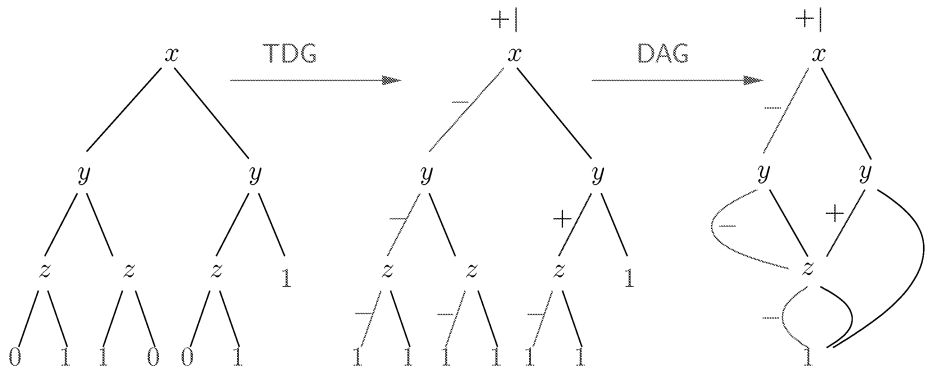
SBDD and FPOS are defined recursively:

- A SBDD is a couple (sign, positive func): $(s, f) \in \{+, -\} \times FPOS$
- The leaf 1 is a FPOS
- A tuple in $V \times SBDD \times FPOS$ is a FPOS, with the same range constraints than for classic BDD

Examples:

- $(+, 1)$ is "always true" $(-, 1)$ is "always false"
- $(+, \begin{array}{c} x \\ / \backslash \\ (-,1) \quad 1 \end{array})$ is x $(-, \begin{array}{c} x \\ / \backslash \\ (-,1) \quad 1 \end{array})$ is $\neg x$

Full example: $x \cdot y + (y \oplus z)$



Notes on complexity

- Negation is free
- Always better than "classical" BDD (space and time)

Using a BDD library

- Even if it is not explicit, they are always SBDD
- Variable ordering is hidden (dynamic reordering)
- high level Boolean functions are provided (true-bdd, false-bdd, idy-bdd(v), and-bdd(f,g) etc)
- Some other ad hoc procedures (depending on Shannon decomposition)

Symbolic algorithms

Encoding sets by formulas

- Enumerative algo $\Rightarrow$ complexity is related to number of states/transitions
- Idea: encoding sets (states, transitions) by Boolean formula (BDD)
- Example: $S = \{x, y, z, t\}$, states such that $x + y \cdot \neg t$:
  - 10 concrete states
  - small formula (3 BDD nodes)

Forward symbolic algorithm ...

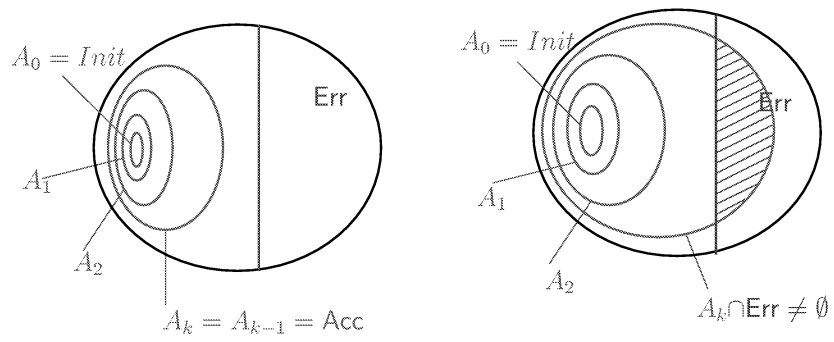
- operates on a verification program $(S, V, \mathsf{Init}, G, \phi, H)$, (we note $Q = 2^{|S|}$ the state space),
- manipulates sets of states (formulas on S) and transitions (formulas on $S \times V$),
- uses set (i.e. logical) operators ($\cup$, $\cap$, $\setminus$ etc),
- uses image computing: $\mathsf{Post}_H : 2^Q \to 2^Q$
  $\mathsf{Post}_H(X) = \{q'/\exists q \in X, v \in 2^V \ H(q, v) \wedge q \xrightarrow{v} q'\}$
  (implementation is presented later)

Main lines of the algo.

Manipulates a BDD A = states reachable in less than n transitions

- Initially: $A :=$ Init
- Repeat:
  - if $A \wedge \text{Err} \neq 0$ then EXIT(failed)
  - else let $A' := A \vee \text{Post}_H(A)$
    if $A' = A$ then EXIT(succeed)
    else $A := A'$, and continue

When the proof succeeds, we have $A = A' = \text{Acc}$



Proof succeeds Proof fails

Naive implementation of Post_H(X)

Using only logical operators, one build a (huge) formula over:

- source state variables $s_1, s_2, \cdots s_n$ (or s)
- free variables $v_1, v_2, \cdots v_m$ (or v)
- target state variables $s'_1, s'_2, \cdots s'_n$ (or s')

$$\exists s, v(\ X(s) \wedge H(s,v) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(s,v)\ )$$

$\rightarrow s$ is a source state
$\rightarrow (s,v)$ satisfies the assumption
$\rightarrow$ each s'_i is the image of g_i
$\rightarrow$ elimination of all s_i and all v_j

Result: the formula $N(s')$ characterizing the target states

Efficient implementation of Post_H(X)

Problem: naive method merges s_i and s'_j in BDD
Idea: using the fact that we have transition ***functions***
How: Define Post_H(X) by induction on transition functions
In order to simplify, we note:

- l for (s,v)
- $Y(l)$ for $X(l) \wedge H(l)$
  (Remark: $Y \neq 0$, otherwise it's trivial $\text{Post}_H(0) = 0$)
- $Img[g_1 ... g_n](Y)$ the expected formula over s'

Formally, we have: $Img[g_1 ... g_n](Y) = \exists l\ Y(l) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(l)$

Let us study the Shannon decomposition of this formula ...

Decomposition according s'_1:

- $s'_1 = 1$ gives $I_1 = \exists l \;\; (Y \wedge g_1)(l) \wedge (\wedge_{i=2}^n s'_i = g_i(l))$
- $s'_1 = 0$ gives $I_0 = \exists l \;\; (Y \wedge \neg g_1)(l) \wedge (\wedge_{i=2}^n s'_i = g_i(l))$

We consider 3 cases:

- $Y \wedge g_1$ is identically false (i.e. $Y \wedge \neg g_1 = Y$):
  $I_1 = 0$
  $I_0 = (\exists l \;\; Y(l) \wedge \wedge_{i=2}^n s'_i = g_i(l)) = Img[g_2 ... g_n](Y)$
- $Y \wedge \neg g_1$ is identically false (i.e. $Y \wedge g_1 = Y$):
  $I_1 = (\exists l \;\; Y(l) \wedge \wedge_{i=2}^n s'_i = g_i(l)) = Img[g_2 ... g_n](Y)$
  $I_0 = 0$
- otherwise:
  $I_1 = \exists l \;\; (Y \wedge g_1)(l) \wedge (\wedge_{i=2}^n s'_i = g_i(l)) = Img[g_2 ... g_n)](Y \wedge g_1)$
  $I_0 = \exists l \;\; (Y \wedge \neg g_1)(l) \wedge (\wedge_{i=2}^n s'_i = g_i(l)) = Img[g_2 ... g_n)](Y \wedge \neg g_1)$

Conclusion: recursive definition of Img, where s'_i variables are never merged with the other

- $Img[](Y) = 1$
- $Img[g_i ... g_n](Y) =$

if $Y \Rightarrow g_i$ then $\quad \begin{array}{c} s'_i \\ 0 \quad\quad Img[g_{i+1} ... g_n](Y) \end{array}$

if $Y \Rightarrow \neg g_i$ then $\quad \begin{array}{c} s'_i \\ Img[g_{i+1} ... g_n](Y) \quad\quad 0 \end{array}$

else $\quad \begin{array}{c} s'_i \\ Img[g_{i+1} ... g_n](Y \wedge \neg g_i) \quad\quad Img[g_{i+1} ... g_n](Y \wedge g_i) \end{array}$

Optimization of image computing

"Knowing that" operators

intuitively, $h = f$ knowing that g is

- equivalent to f if g is true ($f.g \Rightarrow h \Rightarrow f + \bar{g}$)
- such that $h = 1$ if $g \Rightarrow f$
- such that $h = 0$ if $g \Rightarrow \neg f$
- as *simple* as possible otherwise

Remarks:

- Depend on a particular representation (not strictly logical)
- There are many of such operators
- Some of them have ***interesting*** extra properties

Constrain operator $f \downarrow g$, is defined for $g \neq 0$ by:

- $f \downarrow 1 = f$
- $0 \downarrow g = 0$
- $1 \downarrow g = 1$
- $\begin{array}{c} x \\ f_0 \quad f_1 \end{array} \downarrow \begin{array}{c} x \\ 0 \quad g_1 \end{array} = f_1 \downarrow g_1$
- $\begin{array}{c} x \\ f_0 \quad f_1 \end{array} \downarrow \begin{array}{c} x \\ g_0 \quad 0 \end{array} = f_0 \downarrow g_0$
- otherwise, classical "balance" rules

Extra properties of constrain

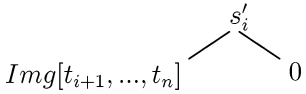
- distributes on negation:
  $(\neg f) \downarrow g \equiv \neg(f \downarrow g)$
- substitutes to $\wedge$ under $\exists$ quantifier:
  $\exists x \;\; (f \wedge g)(x) \equiv \exists x \;\; (f \downarrow g)(x)$
- in particular:
  $\exists l \;\; Y(l) \wedge \bigwedge_{i=1}^{n} s'_i = g_i(l) \;\; \equiv \;\; \exists l \;\; \bigwedge_{i=1}^{n} (s'_i = (g_i \downarrow Y)(l))$

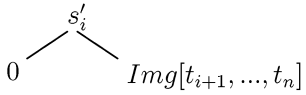
Constrain and image computing
$Img[g_1 ... g_n](Y) = Img[(g_1 \downarrow Y) ... (g_n \downarrow Y)](1)$
$\Rightarrow$ second argument useless, only compute universal images

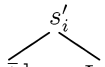
Optimized image computing

- Compute all $t_i = g_i \downarrow (X \downarrow H)$
- Then $Img[t_1, ..., t_n]$ with:

$Img[] = 1$

$Img[0, t_{i+1}, ..., t_n] =$ 

$Img[1, t_{i+1}, ..., t_n] =$ 

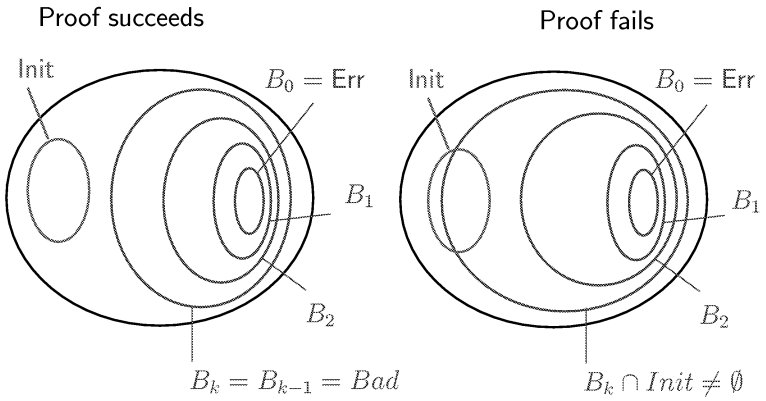
$Img[t_i, t_{i+1}, ..., t_n] = Img[t_{i+1} \downarrow \bar{t}_i, ..., t_n \downarrow \bar{t}_i]$ 

$Img[t_{i+1} \downarrow t_i, ..., t_n \downarrow t_i]$

Backward symbolic algorithm

- Very similar to forward
- Uses reverse image computing $\text{Pre}_H : 2^Q \rightarrow 2^Q$
  $\text{Pre}_H(X) = \{q \;/\; \exists\, q' \in X, v \in 2^V \;\; H(q, v) \wedge q \xrightarrow{v} q'\}$
- Uses $B =$ ***states leading to Err in less than*** n ***transitions***
- Initially: $B :=$ Err
- Repeat:
  - if $B \wedge$ Init $\neq 0$ then EXIT(failed)
  - else let $B' := B \vee \text{Pre}_H(B)$
    if $B' = B$ then EXIT(succeed)
    else $B := B'$, and continue

When the proof succeeds, we have $B = B' =$ Bad



Implementation of $\text{Pre}_H(X)$

No need to merge s_i and s'_i in BDD

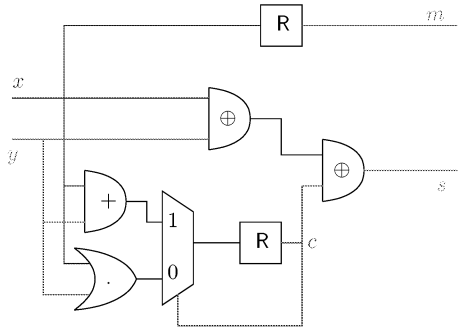
Similar to function composition

- $\text{Pre}_H(X) = \exists v \;\; H(s,v) \wedge Revim[X](s,v)$

with:

- $Revim[0] = 0$
- $Revim[1] = 1$
- $Revim[\begin{array}{c} s'_i \\ / \quad \backslash \\ X_0 \quad X_1 \end{array}] = g_i(s,v) \cdot Revim[X_1] + \neg g_i(s,v) \cdot Revim[X_0]$

Example of forward proof



Serial adder :

- inputs x, y
- sum s, carry c

Register :

- m encodes $2x$

time →

c	0	0	1	0	
x	0	1	0		(2)
y	1	1	0		(3)
s	1	0	1		(5)
m	0	0	1		(4)

Expected property: if $x = y$ then $s = m$

Model

- 2 inputs $V = \{x, y\}$
- 2 memories $S = \{c, m\}$
  with $c_{init} = 0$, and $c' = c.(x+y) + \bar{c}.x.y$
  and $m_{init} = 0$, and $m' = x$
- $H \equiv (x = y)$
- $\phi \equiv (m = (c \oplus x \oplus y))$

Error states:

- $\text{Err} \equiv (\exists x, y \;\; H \wedge \neg\Phi) \equiv (c \oplus m)$

Step 0

Initial state characterized by $\text{Acc}_0 \equiv (\bar{c} \cdot \bar{m})$,

intersection with Err empty, continue...

Step 1

$\Rightarrow$ Image computing for $\text{Acc}_0 \wedge H = (\bar{c} \cdot \bar{m}) \cdot (x = y)$

we have $g_c \downarrow (\bar{c} \cdot \bar{m}) \cdot (x = y) = x$

we have $g_m \downarrow (\bar{c} \cdot \bar{m}) \cdot (x = y) = x$

so $Img[x, x] = \begin{array}{c} c' \\ / \quad \backslash \\ Img[x \downarrow \bar{x}] \quad Img[x \downarrow x] \end{array}$

$= \begin{array}{c} c' \\ / \quad \backslash \\ Img[0] \quad Img[1] \end{array} = \begin{array}{c} c' \\ / \quad \backslash \\ m' \quad m' \\ / \backslash \quad / \backslash \\ 1 \;\; 0 \;\; 0 \;\; 1 \end{array} \equiv (c' = m')$

New reachable states: $\text{Acc}_1 \equiv \text{Acc}_0 + (c = m) \equiv (c = m)$
intersection with Err empty, continue...
Step 2
$\Rightarrow$ Image computing for $\text{Acc}_1 \wedge H = (c = m) \cdot (x = y)$
we have $g_c \downarrow (c = m) \cdot (x = y) = x$
we have $g_m \downarrow (c = m) \cdot (x = y) = x$
So $Img[x, x] \equiv \ (c' = m')$
$\text{Acc}_2 = \text{Acc}_1$, stop: proof succeeds

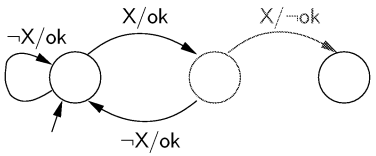
Extendable Assumptions

Example

- Raising edge program:
  `E = X and not (false -> pre X);`
- Assuming that: ***"X cannot occur twice in a row"***
  we have: ***"E is equal to X"***
- Property observer is simply: `ok = (E = X);`
- Assumption observer is (for instance):
  `assume = not(X and pre X);`
  $\Rightarrow$ verification succeeds

What about: `assume = not(pre X and pre pre X);` ?

As a matter of fact, infinite sequences such that
`not(pre X and pre pre X)` remains true
are exactly the same than those such that:
`not(X and pre X)` remains true !
But, with this assumption, the proof fails:



This is indeed a ***false negative***, due to the approximation we have made: ***always ((once not*** `assume`***) or*** `ok`***)***
instead of: ***(always*** `assume`***)*** $\Rightarrow$ ***(always*** `ok`***)***
Intuitively: the red transition should not be considered since it leads to a ***sink*** state

Extendable Assumption

- An assumption is extendable if, whatever is the state value, there exist some input value satisfying the assumption
- Example: `not(X and pre X)` is extendable, while `not(pre X and pre pre X)` is not
- Formally: H extendable iff $\forall q \exists v H(q, v)$
- If H is extendable:
  $always((once \neg H) or \phi) \equiv (always H) \Rightarrow (always \phi)$
- If H is not extendable:
  $always((once \neg H) or \phi) \Rightarrow (always H) \Rightarrow (always \phi)$
  (so the approximation was at least conservative)

Non-extendable assumptions

They raise problems:

- they may introduce ***false negative***
- they may be ***inconsistent***:
  it is impossible, starting in the initial state, to satisfy indefinitely the assumption

It's a big problem $\Rightarrow$ anything you try to proove is true if the assumption is inconsistent: it's certainly a design error!
A good proof algorithm may exit with:

- inconsistent assumption
- success (true property)
- failure (inconclusive)

What to do with non-extendable assumption ?

- Detect and reject them?
  too strong: they are sometime very convenient
- Detect and treat them?

A priori treatment

- let $\mathsf{PreT}(X) = \{(q,v) \mid \exists (q',v')\ X(q',v') \wedge q \xrightarrow{v} q'\}$
- let $\mathsf{Some}(H) = \nu X.(X = H \wedge \mathsf{PreT}(X))$
- $\mathsf{Some}(H)$ is the greatest extendable assumption in H

Solution: first build $H' = \mathsf{Some}(H)$, then perform a classical proof with H' as assumption
Exo: What does it mean if $\mathsf{Some}(H) = \emptyset$?
more generally if $\mathsf{Init} \cap \mathsf{Some}(H) = \emptyset$?

A posteriori treatment

For forward proof, the "extendable" notion is too strong: it does not matter if H is non-extendable as long as reachable states are not concerned.
Solution:

- Compute the whole set of reachable states
- At the end, remove all sinks from Acc(i.e. $\mathsf{Acc}' = \mathsf{Some}(\mathsf{Acc})$)
  $\mathsf{Some}(\mathsf{Acc}) = \emptyset$: inconsistent assumption
  $\mathsf{Some}(\mathsf{Acc}) \cap \mathsf{Err} = \emptyset$: success
  $\mathsf{Some}(\mathsf{Acc}) \cap \mathsf{Err} \neq \emptyset$: failure

Exo: What about backward proof ?

Paul Petersson
Wang Yi

Uppsala Universitet

Modeling and Verification:
an introductory course

- History, motivation and basic concepts
- Transition systems and Temporal logics
- Basic model checking algorithms
- Timed systems,Timing constraints, DBM's
- Unification of Scheduling and Verification
- Tools (e.g. UPPAAL, and TIMES)

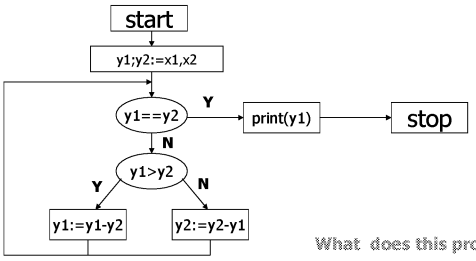
Modeling and Verification

Lecture 1

a brief introduction

History: how the dream started 35 years ago

- Program verification, [Floyd 1967, Hoare 1969]
  - Hoare Logic: {P} program {Q} to develop bug-free programs



What does this program?

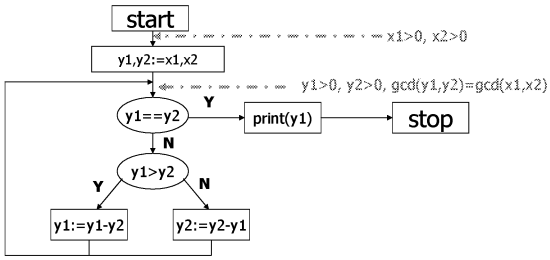
What does this program do?

(Partial correctness)

- It computes the greatest common divisor (gcd) of x1 and x2 and you can prove it [Floyd 67]:
  - Initially
    - x1>0, x2>0
  - At each iteration of the loop:
    - y1>0, y2>0, gcd(x1,x2) = gcd(y1,y2)
  - When done
    - y1=gcd(x1,x2)

What this program does?

(Partial correctness)



One more example

(Total correctness)

```
Function foo(n :integer): integer
begin
if n==1 then 1
        else if even(n) then foo(n/2)
                        else foo(3*n+1)
end
```

Does this program terminate for any n?

Reality: 10 years later **(1980)**

- The majority of programs are never proven correct! what went wrong?
  - Difficult to find and prove invariants:
    - partial correctness
  - Difficult to prove termination:
    - total correctness
  - Difficult to write complete specifications:
    - what I really want?
- What to do?
  - Start another research program! In 20 years, the problems will be solved, hopefully

History: Model checking for reative systems invented in the early 80s **[Pnueli 77, Clarke et al 81,Sifakis et al 81]**

- Temporal logics, Model $\models \phi$
  - nonterminination, control-intensive, less data
  - Finite state systems [ABP ca 140 states, 1984]
  - (Infinite state systems, a hot topic right now)
- BDD-based symbolic technique [Bryant 86]
  - SMV 1990 Clarke et al, state-space 10^{20}
- Many followers e.g SPIN, COSPAN ... were developed ...

History: Model checking for real time systems, started in the early 80s **[Alur&Dill 1990, UPPAAL et al]**

- Timed automata, timed process algebras [Alur&Dill 1990, I was there!]
- DBM and constraints [Bellman 58, Dill 89]
- Kronos, Hytech, 1993-1995
- TAB, 1993, UPPAAL 1995 (**TIMES 2002)**

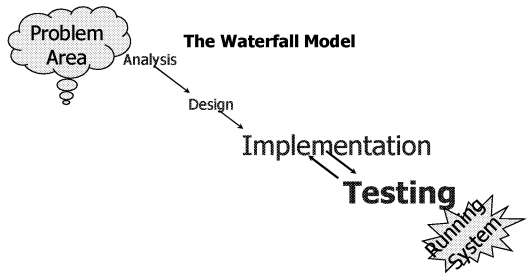
Reality 35 years later **(2003)**

- Many extensions and improvements have been proposed, various tools exist: commercial and non-commercial
- Good complete specifications are still hard to obtain
- However this is not a real problem !

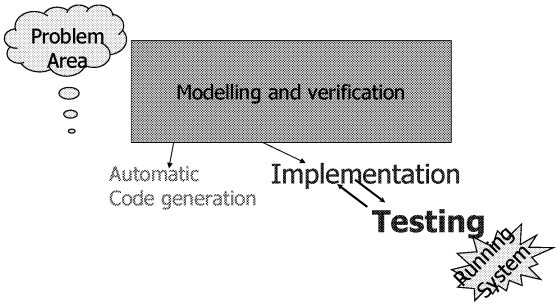
Reality 35 years later **(2003)**

- Checking simple (e.g. Is a state reachable?) or generic (e.g. Is a program deadlock free?) properties is already extremely useful!
- The goal is no longer seen as proving that a system is completely, absolutely and undoutedly correct (bug-free)
- The objective is to have tools that can help a developer find errors and gain confidence in her design. That is achievable
- Now widely used in hardware design, protocol design, embedded systems...

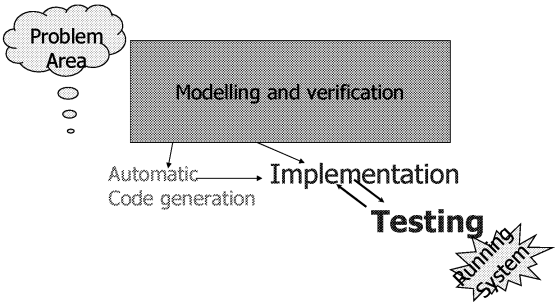
Traditional software development



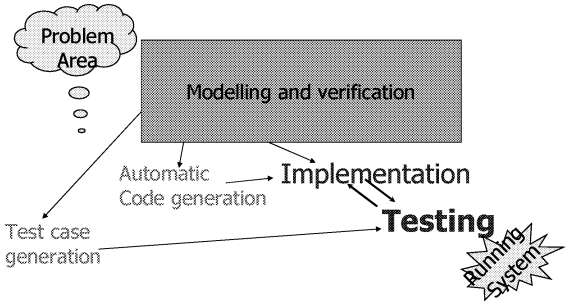
Software development: the future



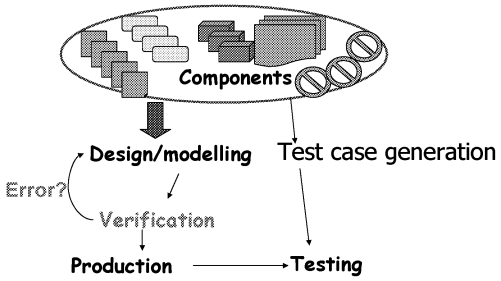
Software development: the future



Software development



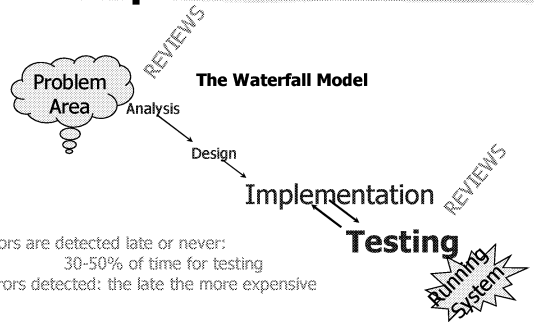
Software Development: the Future



Model Checking

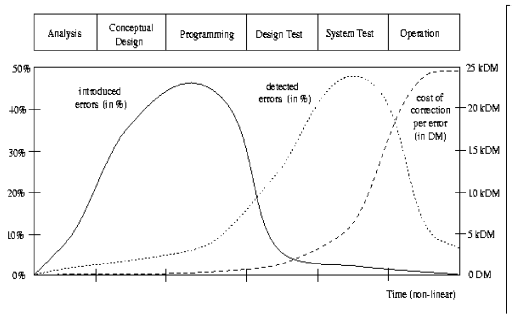
in a Nutshell

Traditional software development



- Errors are detected late or never: 30-50% of time for testing
- Errors detected: the late the more expensive

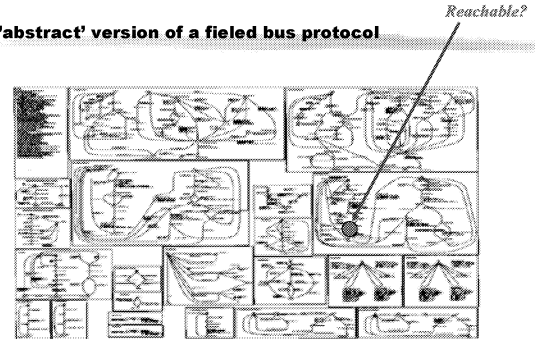
Introducing, Detecting and Correcting errors



Finding errors as early as possible!

HOW?

An 'abstract' version of a fieled bus protocol



Example: Petersson's algorithm

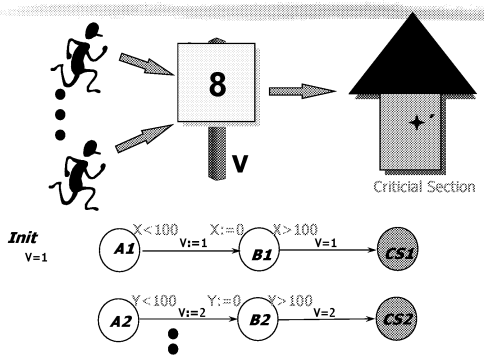
turn, flag1, flag2: shared variable

- Process 1
- Loop
- flag1:=1; turn:=2
- While (flag2 and turn=2) wait
- **CS1**
- flag1:=0
- End loop

- Process 2
- Loop
- flag2:=1; turn:=1
- While (flag1 and turn=1) wait
- **CS2**
- flag2:=0
- End loop

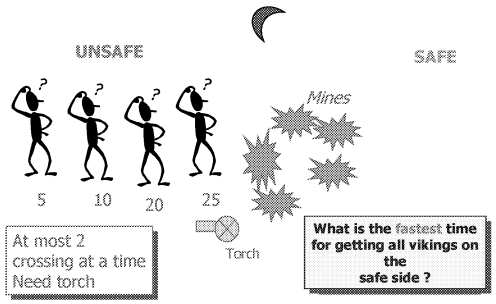
Question: no more than one process run in CS?

Example: Fischer's Protocol



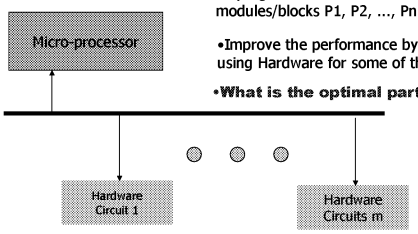
Example: the Vikings Problem

Real time scheduling



Performance Estimation in H/S Co-Design

(joint work with Ji Feng)



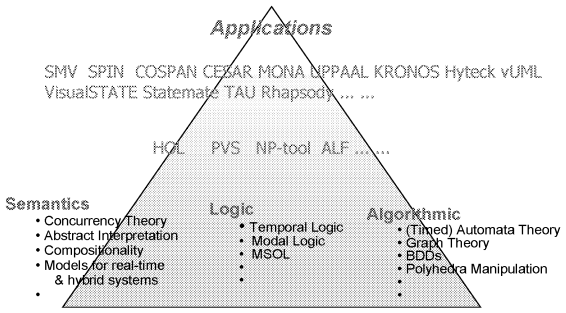
- A program P which contains modules/blocks P1, P2, ..., Pn
- Improve the performance by 50% using Hardware for some of the Pi's
- **What is the optimal partition ?**

This is an optimal reachability problem !

How do we know they all work?

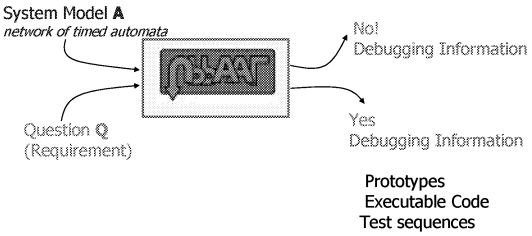
- Alternating bit protocol
- Sliding window protocol
- Leader selection algorithm
- Start-up synchronization protocols
- TTCAN, TTP
- ... ...

Tools for modelling and verification



UPPAAL = UPP$_{sala}$ + AAL$_{borg}$

A tool set for modelling and verification of real-time systems developed jointly by Uppsala and Aalborg University

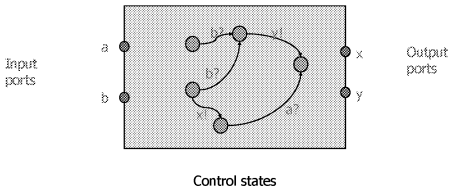


MODELLING

How to construct Model ?

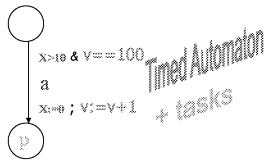
Modelling = programming+abstraction

Program as State Machine!



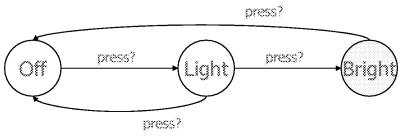
How?

Construction of Model: Timing and Data



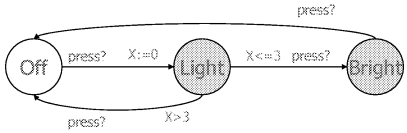
- Events
  - synchronization
  - interrupts
- Timing constraints
  - specifying event arrivals
  - e.g. Periodic and sporadic
- Dat variables
  - Guards
  - assignments
- Tasks
  - Computing time
  - Deadline, priority etc

Intelligent Light Control



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

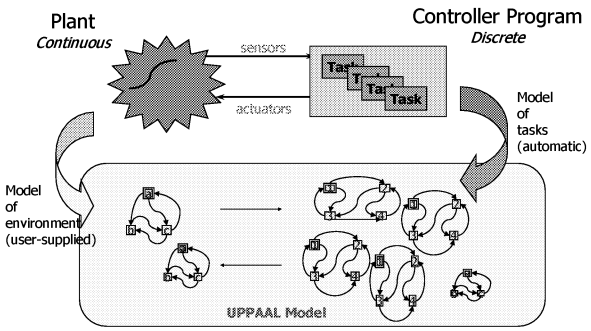
Intelligent Light Control (with timer)



Solution: Add real-valued clock x

How?

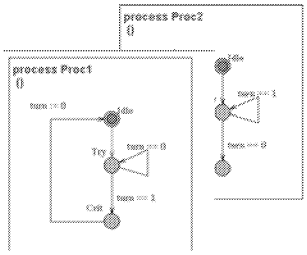
Construction of Models: Concurrency

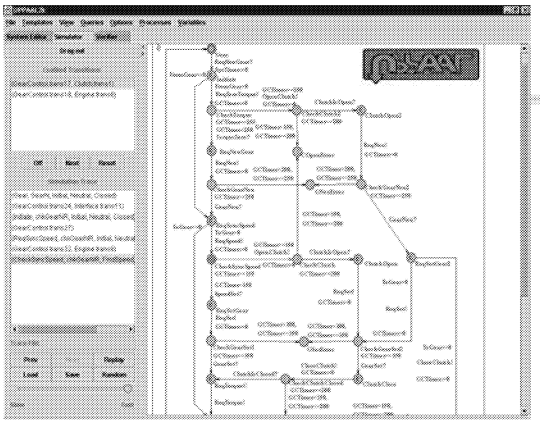


Modelling in UPPAAL: Example

```
P1 :: while True do
      T1 : wait(turn=1)
      C1 : CS1; turn:=0
      endwhile
||
P2 :: while True do
      T2 : wait(turn=0)
      C2 : CS2; turn:=1
      endwhile
```

Mutual Exclusion Program





SPECIFICATION

How to ask questions: Specs ?

Specification=Requirement, Lamport 1977

- Safety
  - Something (bad) will not happen
- Liveness
  - Something (good) must happen
- Realizability (Schedulability)

Specification: Examples

- AG not (CS1 and CS2)
  - never CS1 and CS2
  - Safety property
- AG (a $\rightarrow_{<=10}$ b)
  - if a then b within 10
  - Bounded liveness property
- EF p.test
  - Useful for debugging
- EF false
  - Generate the whole state space
  - Report deadlocks etc.
- AF rich and EG happy (liveness)
- AG (try => AF critical-section) (liveness)

VERIFICATION

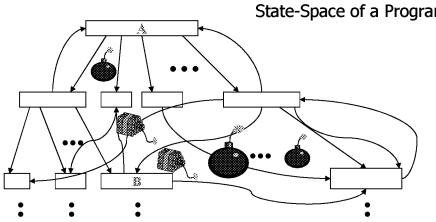
Model meets Specs ?

Verification

- Semantics of a system
  = all states + state transitions
  (all possible executions)
- Verification
  = state space exploration + examination

Verificatioin = Searching

State-Space of a Program



(1) Is it possible to fire the bombs?
(2) Is it possible to go from A to B within 10 sec?

Approaches to Verification

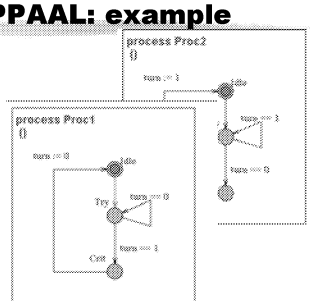
- Manual: Proof systems, paper and pen
  - Find invariants (difficult !)
  - Induction: Assume nth-state OK, check (n+1)th OK
  - Boring ☹ (more fun with programming)
- Semi-automatic: Theorem proving
  - Use theorem provers to prove the induction step
  - e.g. PVS, HOL, ALF
  - Require too much expertise ☹
- Automatic: Model-Checking ☺
  - State-Space Exploration and Examination
  - e.g. SPIN, SMV, UPPAAL

Two basic verification algorithms

- Reachability analysis
  - Checking safety properties
- Loop detection
  - Checking liveness properties

Modelling in UPPAAL: example

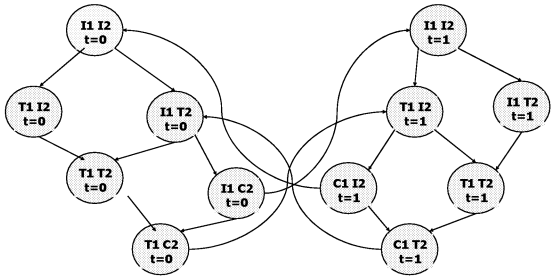
```
P1 :: while True do
      T1 : wait(turn=1);
      C1 : CS1; turn:=0;
      endwhile
||
P2 :: while True do
      T2 : wait(turn=0);
      C2 : CS2; turn:=1;
      endwhile
```



Mutual Exclusion Program

Is it possible that P1 and P2 reach C1 and C2 simultaneously?

Verification: example

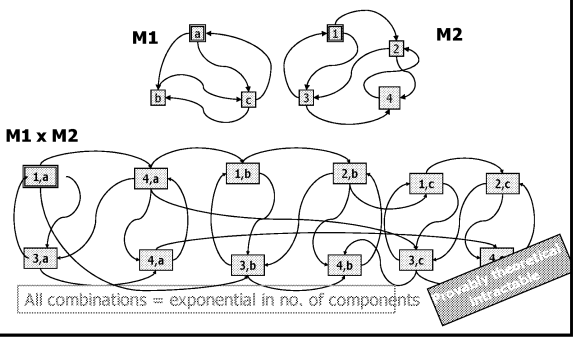


(C1 C2) is never reachable!

Why testing not good enough ?

- **Testing/simulation** of designs/implementations may not reveal error (e.g., no errors revealed after 2 days)
- **Formal verification** (=exhaustive testing) of design provides 100% coverage (e.g., error revealed within 5 min).
- **TOOL** support.

Problem with verification: 'State Explosion'



EXAMPLE

10 components and each with10 states

# of control states = 10,000,000,000 =10 G
Each state needs 4*(10 x 10) = 400 B

Worst case memory usage >> 4000 GB

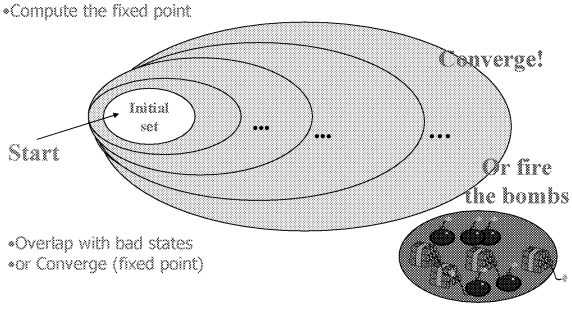
Solutions

- Theorem provers
- Manual proofs
- Symbolic Techniques e.g. BDD [Bryant 86]
- Abstraction techniques [Cosot, Kurshan]
- Approximation methods [Holzman, Wang-Toi ...]
- On-the-fly verification [Holzman]
- Partial order reduction [Wolper et al]
- Compositional verification [too many]
- Combining theorem provers and model checkers
- ... ...

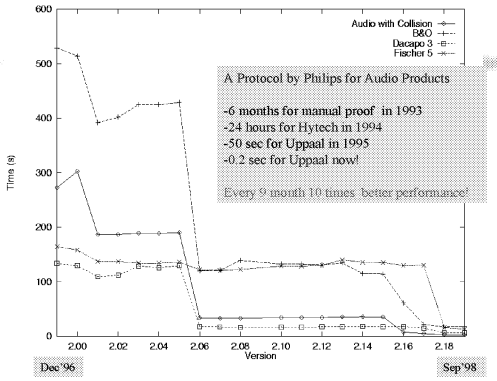
Symbolic Techniques:

Compute Sets of States instead of one-by-one

- Use formulas to represent sets of states
- Compute the fixed point



- Overlap with bad states
- or Converge (fixed point)



The dream goes on ... ...

- *Automatic Verification, a useful and applicable technique as compiler theory!*

End of INTRODUCTION

Lecture 2

Model-Checking Finite-State Systems

Finite Automata, CTL, LTL and Model Checking

Finite state automata

- Finite graphs with labels on edges/nodes
  - a set of nodes (states)
  - a set of edges (transitions)
  - a set of labels (alphabet)

Complete Systems and Kripke Structure

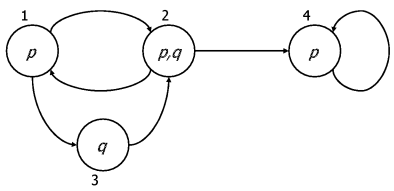
- From now on, we shall consider only Complete systems, that is, automata with labels on nodes.
  - There is no essential difference between models with labels on nodes or transitions
- This is the so called Kripke Structure, that is, automata with propositions labeled on states

CTL Models = *Kripke Structures*

A CTL-model is a triple $\mathcal{M} = (S, R, Label)$ where

- S is a non-empty set of states,
- $R \subseteq S \times S$ is a total relation on S, which relates to $s \in S$ its possible successor states,
- $Label : S \longrightarrow 2^{AP}$, assigns to each state $s \in S$ the atomic propositions $Label(s)$ that are valid in s.

Example



CTL: Computation Tree Logics

defined on Computation Trees of Kripke structures

Computation Tree Logic, CTL

Clarke & Emerson 1980

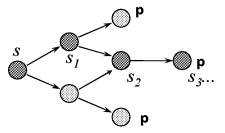
Syntax

$$\phi \ ::= \ p \mid \neg\phi \mid \phi \vee \phi \mid \mathsf{EX}\,\phi \mid \mathsf{E}\,[\phi\,\mathsf{U}\,\phi] \mid \mathsf{A}\,[\phi\,\mathsf{U}\,\phi].$$

- EX (pronounced "for some path next")
- E (pronounced "for some path")
- A (pronounced "for all paths") and
- U (pronounced "until").

Path

Definition 20. (Path)
A *path* is an infinite sequence of states $s_0\, s_1\, s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geqslant 0$.



The set of path starting in s

$P_{\mathcal{M}}(s)$

Formal Semantics

(satisfaction relation $\models$)

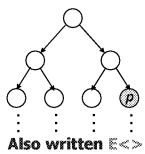
$s \models p$	iff $p \in Label(s)$
$s \models \neg\phi$	iff $\neg(s \models \phi)$
$s \models \phi \vee \psi$	iff $(s \models \phi) \vee (s \models \psi)$
$s \models \mathsf{EX}\,\phi$	iff $\exists\sigma \in P_{\mathcal{M}}(s).\, \sigma[1] \models \phi$
$s \models \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$	iff $\exists\sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\, \sigma[j] \models \psi \ \wedge\ (\forall 0 \leqslant k < j.\, \sigma[k] \models \phi))$
$s \models \mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$	iff $\forall\sigma \in P_{\mathcal{M}}(s).(\exists j \geqslant 0.\, \sigma[j] \models \psi \ \wedge\ (\forall 0 \leqslant k < j.\, \sigma[k] \models \phi))$.

CTL, Derived Operators

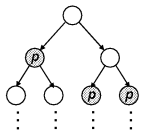
$$\mathsf{EF}\,\phi \ \equiv\ \mathsf{E}\,[\mathrm{true}\,\mathsf{U}\,\phi] \qquad \textit{possible}$$

$$\mathsf{AF}\,\phi \ \equiv\ \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,\phi]. \qquad \textit{inevitable}$$

EF p



AF p



Also written E<> p
This is used in UPPAAL !

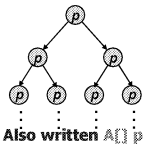
CTL, Derived Operators

$$\mathsf{EG}\,\phi \ \equiv\ \neg\,\mathsf{AF}\,\neg\,\phi \qquad \textit{potentially always}$$

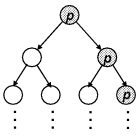
$$\mathsf{AG}\,\phi \ \equiv\ \neg\,\mathsf{EF}\,\neg\,\phi \qquad \textit{always}$$

$$\mathsf{AX}\,\phi \ \equiv\ \neg\,\mathsf{EX}\,\neg\,\phi.$$

AG p



EG p



Also written A[] p
This is used in UPPAAL !

There are too many operators! But

We need to remember only the following:

(neXt time)
(Future, some time)
(Global)
(Until)

The most useful are EF, AG, EG and AF:
EF and AG for safety properties
EG and AF for liveness

Theorem

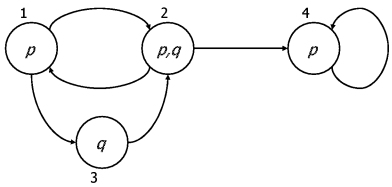
All operators are derivable from

- EX f
- EG f
- E[f U g]

and boolean connectives

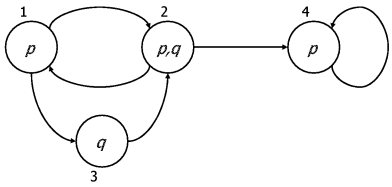
$$A[f \text{ U } g] \equiv \neg E[\neg g \text{ U} (\neg f \wedge \neg g)] \wedge \neg EG \neg g$$

Example



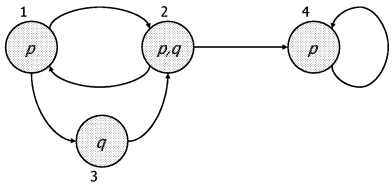
Example

EX p



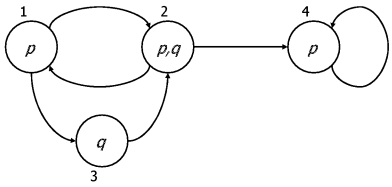
Example

EX p



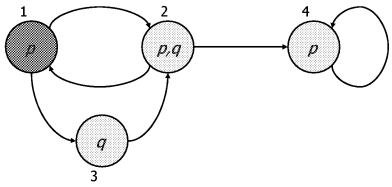
Example

AX p



Example

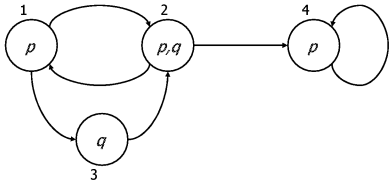
AX p



Note: state 1 doesn't satisfy AX p

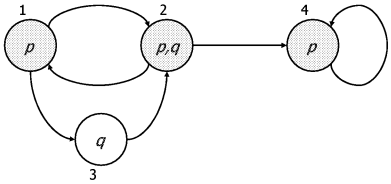
Example

EG p



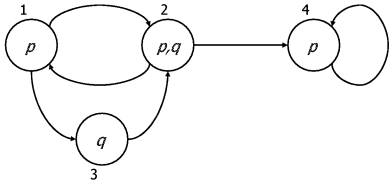
Example

EG p



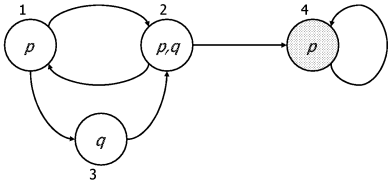
Example

AG p



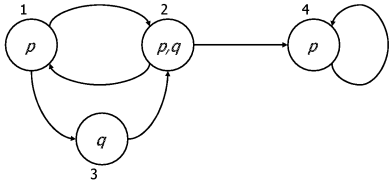
Example

AG p



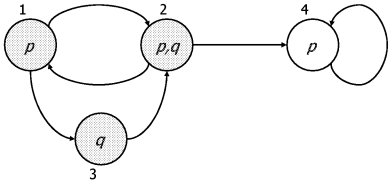
Example

A[p **U** q **]**



Example

A[p **U** q **]**



Properties of MUTEX example ?

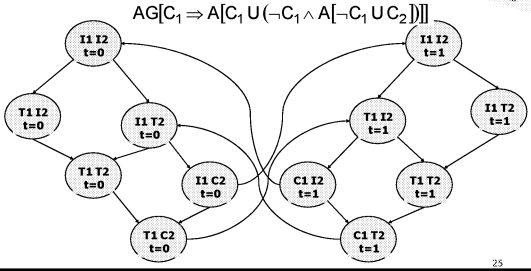
$AG\,\neg(C_1 \wedge C_2)$

$AG[\,T_1 \Rightarrow AF(C_1)]$

$EG[\neg C_1]$

$AG[C_1 \Rightarrow A[C_1\,U\,(\neg C_1 \wedge A[\neg C_1\,U\,C_2])]]$

HOW to DECIDE IN GENERAL



CTL Model Checking Algorithms

Fixpoint Characterizations

$$\mathsf{EF}\,p \quad \equiv \quad p \vee \mathsf{EXEF}\,p$$

or let A **be the set of states satisfying** EF p **then**

$$\mathsf{A} \quad \equiv \quad p \vee \mathsf{EXA}$$

in fact A **is the smallest one of sets satisfying the equations (the least fixpoint)**

Fixed points of monotonic functions

- Let τ be a function $\mathcal{S} \to \mathcal{S}$
- Say τ is *monotonic* when

$$x \subseteq y \;\text{ implies }\; \tau(x) \subseteq \tau(y)$$

- Fixed point of τ is y such that

$$\tau(y) = y$$

- If τ monotonic, then it has
  - least fixed point $\mu y.\ \tau(y)$
  - greatest fixed point $\nu y.\ \tau(y)$

Iteratively computing fixed points

- Suppose $\mathcal{S}$ is finite
  - The least fixed point $\mu y.\ \tau(y)$ is the limit of

$$\text{false} \;\subseteq\; \tau(\text{false}) \;\subseteq\; \tau(\tau(\text{false})) \;\subseteq \Lambda$$

  - The greatest fixed point $\nu y.\ \tau(y)$ is the limit of

$$\text{true} \;\supseteq\; \tau(\text{true}) \;\supseteq\; \tau(\tau(\text{true})) \;\supseteq \Lambda$$

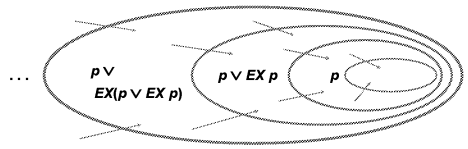
Note, since S is finite, convergence is finite

Example: *EF p*

- *EF p* is characterized by

$$EF\;\; p = \mu y.\,(p \vee EX\;\; y)$$

- Thus, it is the limit of the increasing series...

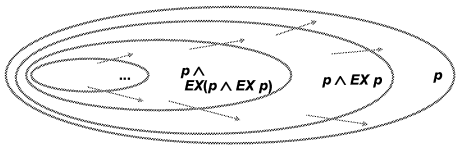


Example: *EG p*

- *EG p* is characterized by

$$EG\ p = \nu y.\ (p \wedge EX\ y)$$

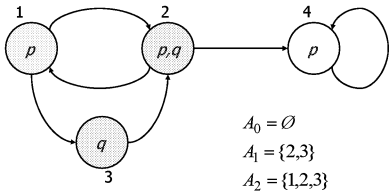
- Thus, it is the limit of the decreasing series...



Example, continued

EF *q*

$$EF\ q = \mu y.\ (q \vee EX\ y)$$



$A_0 = \emptyset$
$A_1 = \{2,3\}$
$A_2 = \{1,2,3\}$
$A_3 = \{1,2,3\}$

Remaining operators

$$\begin{aligned} AF\ p &= \mu y.(p \vee AX\ y) \\ AG\ p &= \nu y.(p \wedge AX\ y) \\ E(pUq) &= \mu y.(q \vee (p \wedge EX\ y)) \\ A(pUq) &= \mu y.(q \vee (p \wedge AX\ y)) \end{aligned}$$

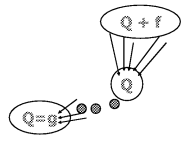
Labeling Methods [Clarke et al 81]

- Check all sub-formulas of $\mathbb{F}$
- For each sub-formula $\mathbb{f}$ of $\mathbb{F}$, label all nodes where $\mathbb{f}$ is true
- Check the composed formulas

function $Sat(\phi : Formula)$: **set of** $State$;
(* precondition: true *)
begin
  if $\phi = \text{true} \longrightarrow$ **return** S
  $[]\ \phi = \text{false} \longrightarrow$ **return** $\emptyset$
  $[]\ \phi \in AP \longrightarrow$ **return** $\{\, s \mid \phi \in Label(s) \,\}$
  $[]\ \phi = \neg\phi_1 \longrightarrow$ **return** $S - Sat(\phi_1)$
  $[]\ \phi = \phi_1 \vee \phi_2 \longrightarrow$ **return** $(Sat(\phi_1) \cup Sat(\phi_2))$
  $[]\ \phi = \mathsf{EX}\,\phi_1 \longrightarrow$ **return** $\{\, s \in S \mid (s, s') \in R \ \wedge\ s' \in Sat(\phi_1) \,\}$
  $[]\ \phi = \mathsf{E}\,[\phi_1\ \mathsf{U}\ \phi_2] \longrightarrow$ **return** $Sat_{EU}(\phi_1, \phi_2)$
  $[]\ \phi = \mathsf{A}\,[\phi_1\ \mathsf{U}\ \phi_2] \longrightarrow$ **return** $Sat_{AU}(\phi_1, \phi_2)$
  fi
(* postcondition: $Sat(\phi) = \{\, s \mid \mathcal{M}, s \models \phi \,\}$ *)
end

Algorithm ideas for checking $\mathbb{E}(\mathbb{f}\ \mathbb{U}\ \mathbb{g})$

- Mark all nodes where $\mathbb{f}$ is true and all nodes where $\mathbb{g}$ is true
- Start from all nodes where $\mathbb{g}$ is true and
- Perform backwards reachability analysis
- Each step backwards, store all nodes in Q where $\mathbb{f}$ is true
- Repeat the above step, until it converges
- Q contains all nodes satisfying $\mathbb{E}(\mathbb{f}\ \mathbb{U}\ \mathbb{g})$

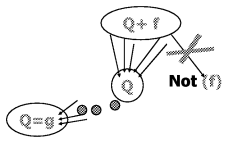


```
function Sat_EU (φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
        Q, Q' := Sat(ψ), ∅;
        do Q ≠ Q' ⟶
            Q' := Q;
            Q := Q ∪ ({ s | ∃ s' ∈ Q. (s, s') ∈ R } ∩ Sat(φ))
        od;
        return Q
(* postcondition: Sat_EU(φ, ψ) = { s ∈ S | M, s ⊨ E[φ U ψ] } *)
end
```

Table 3.4: Labelling procedure for $\mathsf{E}[\phi \,\mathsf{U}\, \psi]$

Algorithm ideas for checking A(f U g)

- Similar to the case for A(f U g)
  - But each step backwards, store all nodes in Q where (f or g) is true, and the stored nodes do not lead to a node where (f or g) is false
  - Repeat the above step, until it converges
  - Q contains all nodes satisfying A(f U g)




```
function Sat_AU (φ, ψ : Formula) : set of State;
(* precondition: true *)
begin var Q, Q' : set of State;
        Q, Q' := Sat(ψ), ∅;
        do Q ≠ Q' ⟶
            Q' := Q;
            Q := Q ∪ ({s | ∀s'.(s,s') ∈ R ⇒ s' ∈ Q} ∩ Sat(φ))
        od;
        return Q
(* postcondition: Sat_AU(φ, ψ) = { s ∈ S | M, s ⊨ A[φ U ψ] } *)
end
```

Table 3.5: Labelling procedure for $\mathsf{A}[\phi \,\mathsf{U}\, \psi]$

Complexity

The worst-case time complexity of checking whether system-model sys satisfies the CTL-formula ϕ is $\mathcal{O}(|\, S_{sys} \,|^2 \times |\, \phi \,|)$

However S_{sys} may be EXPONENTIAL in number of parallel components!
--
FIXPOINT COMPUTATIONS may be carried out using
ROBDD's
(Reduced Ordered Binary Decision Diagrams)
Bryant, 86

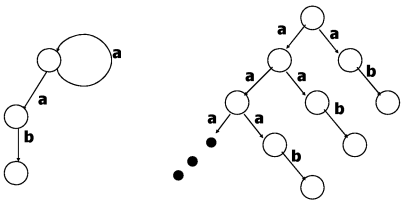
Something more about Finite State Automata and Temporal Logics

(Continuation of Lecture 2)

Branching time semantics

- Computation tree of an automaton is the unfolding of the automaton

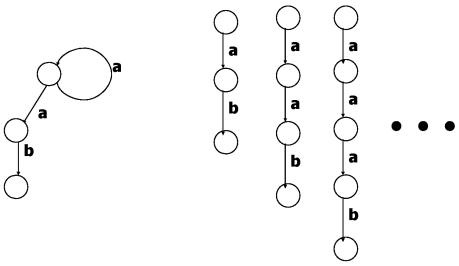
Example (Branching Time)



Linear Time Semantics

- Sequences of transitions (or states)
  - set of possible excecutions of a system
- Suite best for closed systems

Example (Linear Time)



Equivalences and Preorders

- A equivalent to B if the tree of A is identical to the tree of B (Too strong!)
- A is simulated by B if every transition of A is simulated by a transition of B (simulation [Milner78])
- A and B are bisimular if there is a symmetrical simulation between A's and B's states (bisimulation [Milner80])
- A and B are testing equivalent if they can pass the same set of tests (may and must testing [Nicola and Hennessy 84])
- A and B trace-equivalent if they provide the same set of sequences of transitions (trace equivalence [Hoare76])

LTL: Linear Time Logics

defined on infinite traces of Kripke structures with accepting conditions

Models: Infinite Sequences (ω-language accepted by automata)

- Automata with accepting conditions
  - Buchi, Muller automata
- Infininte accepted sequences of transitions as semantics of automata

LTL: Syntax

- P
- not F
- F1 and F2
- **O** F (next time)
- F1 **U** F2 (Until)

LTL: semantics

- assume an automaton M
  - a sequence of M: t=s(0)→s(1)→s(2)→...→s(i) ... ...
  - The set of sequences of M is Comp(M)

- s(i) sat p if p is a label of s(i)
- s(i) sat not F if not (s(i) sat F)
- s(i) sat F1 and F2 if s(i) sat F1 and s(i) sat F2
- s(i) sat **O** F if s(i+1) sat F
- s(i) sat F1 **U** F2 if s(k) sat F2 for some k=>i and s(j) sat F1 for all j such that i<=j<k

LTL: semantics (contn.)

- assume an automaton M
  - a sequence of M: t=s(0)→s(1)→s(2)→...→s(i) ... ...
  - The set of sequences of M is Comp(M)

- t sat F iff s(0) sat F
- M sat F iff t sat F for all sequences t of Comp(M)

Derived Operators

- <>F denotes (true **U** F)
- [ ]F denotes not (<> not F)
- F1 **W** F2 denotes (F1 **U** F2) or [ ]F1 (weak Until-operator)

Model Checking LTL [Wolper et al 1986]

- Given an automata M and a formula F, to check M sat F
  - Construct the formula automaton: A(¬ F)
  - Construct the product automaton M || A(¬ F) (on-the-fly)
- If M || A(¬ F) is empty then M sat F otherwise NO
- Time-Complexity = $|M|*2^{O(|F|)}$

The same idea can be used for CTL model checking using Tree-automata

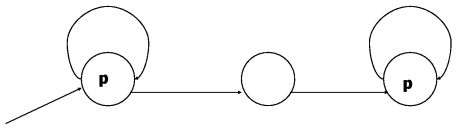
Comparing CTL and LTL

- <> P (LTL) similar AF p (CTL)
  [] p (LTL) similar AG p (CTL)

However,

- **LTL cannot express possibilities properties: EF p**
- **CTL cannot express <>[] p**

Comparing CTL and LTL (contn.)



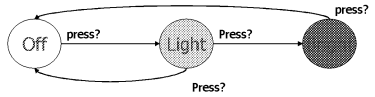
Satisfies <>[] p
but it does not satisfy AF AG p

END
with Finite State Systems

Timed Automata

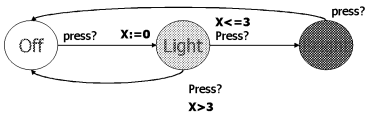
Paul Pettersson
Dept. of Information Technology

Timed Automata
Intelligent Light Control



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

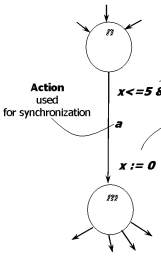
Timed Automata
Intelligent Light Control



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

Timed Automata

(Alur & Dill 1990)



Clocks: ***x, y***

Guard
Boolean combination of comp with integer bounds

Reset
Action perfumed on clocks

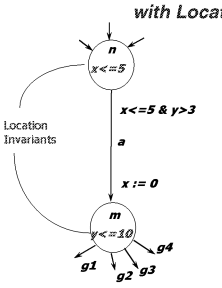
State
(***location*** , ***x***=v , ***y***=u) where v,u are in **R**

Transitions

(***n*** , ***x***=2.4 , ***y***=3.1415) $\xrightarrow{a}$ (***m*** , ***x***=0 , ***y***=3.1415)

(***n*** , ***x***=2.4 , ***y***=3.1415) $\xrightarrow{e(1.1)}$ (***n*** , ***x***=3.5 , ***y***=4.2415)

Timed Automata
with Location Invariants



Clocks: ***x, y***

Transitions

(***n*** , ***x***=2.4 , ***y***=3.1415) $\not\xrightarrow{e(3.2)}$

(***n*** , ***x***=2.4 , ***y***=3.1415) $\xrightarrow{e(1.1)}$ (***n*** , ***x***=3.5 , ***y***=4.2415)

Invariants ensure progress!!

Clock Constraints

For set C of clocks with $x, y \in C$, the set of *clock constraints* over C, $\Psi(C)$, is defined by

$$\alpha ::= x \prec c \mid x - y \prec c \mid \neg\alpha \mid (\alpha \wedge \alpha)$$

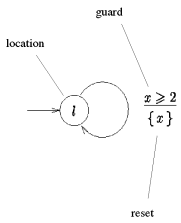
where $c \in \mathbb{N}$ and $\prec \in \{<, \leqslant\}$.

Timed (Safety) Automata

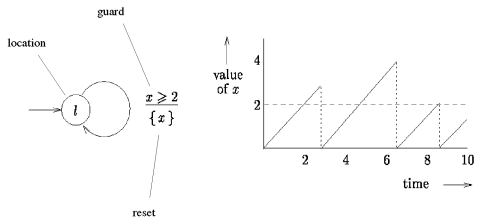
A *timed automaton* $\mathcal{A}$ is a tuple $(L, l_0, E, Label, C, clocks, guard, inv)$ with

- L, a non-empty, finite set of locations with initial location $l_0 \in L$
- $E \subseteq L \times L$, a set of edges
- $Label : L \longrightarrow 2^{AP}$, a function that assigns to each location $l \in L$ a set $Label(l)$ of atomic propositions
- C, a finite set of clocks
- $clocks : E \longrightarrow 2^C$, a function that assigns to each edge $e \in E$ a set of clocks $clocks(e)$
- $guard : E \longrightarrow \Psi(C)$, a function that labels each edge $e \in E$ with a clock constraint $guard(e)$ over C, and
- $inv : L \longrightarrow \Psi(C)$, a function that assigns to each location an *invariant*.

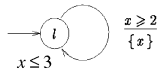
Timed Automata: Example



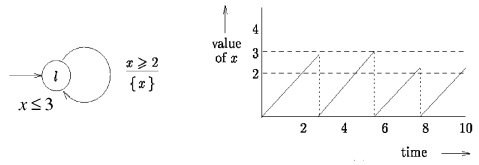
Timed Automata: Example



Timed Automata: Example

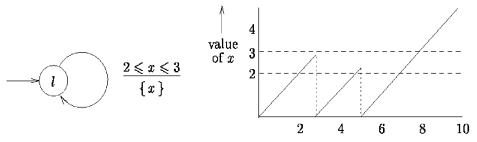


Timed Automata: Example

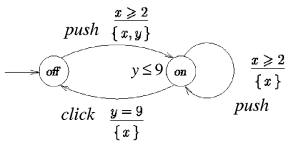


Timed Automata: Example

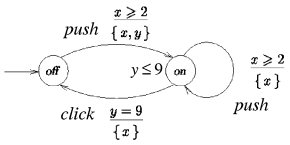
Timed Automata: Example



Light Switch

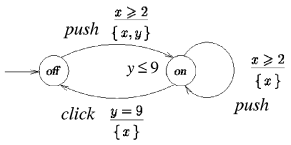


Light Switch



- Switch may be turned on whenever at least 2 time units has elapsed since last "turn off"

Light Switch

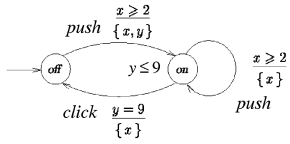


- Switch may be turned on whenever at least 2 time units has elapsed since last "turn off"
- Light automatically switches off after 9 time units.

Semantics

- *clock valuations*: $V(C) \quad v: C \to R_{\geq 0}$
- *state*: (l, v) *where* $l \in L$ *and* $v \in V(C)$
- Semantics of timed automata is a *labeled transition system* $(S, \to)$
  where $S = \{ (l, v) \mid v \in V(C) \text{ and } l \in L \}$
- *action transition* $(l, v) \xrightarrow{a} (l', v')$ *iff* $l \xrightarrow{g\ a\ r} l'$
  $g(v)$ *and* $v' = v[r]$ *and* $Inv(l')(v')$
- *delay transition* $(l, v) \xrightarrow{d} (l, v + d)$ *iff*
  $Inv(l)(v + d')$ *whenever* $d' \leq d \in R_{\geq 0}$

Semantics: Example



$(off, x = y = 0) \xrightarrow{3.5} (off, x = y = 3.5) \xrightarrow{push}$

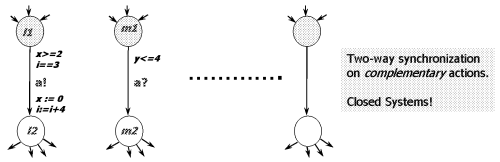
$(on, x = y = 0) \xrightarrow{\pi} (on, x = y = \pi) \xrightarrow{push}$

$(on, x = 0, y = \pi) \xrightarrow{3} (on, x = 3, y = \pi + 3) \xrightarrow{9-(\pi+3)}$

$(on, x = 9 - (\pi + 3), y = 9) \xrightarrow{click} (off, x = 0, y = 9) \ldots$

Networks of Timed Automata

+ Integer Variables + arrays ….



Two-way synchronization on *complementary* actions.

Closed Systems!

Example transitions

(l1, m1,……, x=2, y=3.5, i=3,…..) $\xrightarrow{\text{tau}}$ *(l2,m2,……,x=0, y=3.5, i=7,…..)*

Timed Automata in UPPAAL

- next file…

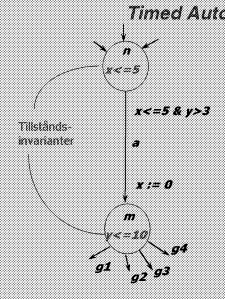
Tidsautomater i UPPAAL

Paul Pettersson
mailto:paupet@docs.uu.se
http://www.docs.uu.se/~paupet/

(Henzinger et al., 1992)

Tidsautomater med Invarianter

Timed Automata + Invariants



Klockor: *x, y*

Transitioner

(*n* , *x*=2.4 , *y*=3.1415) $\xrightarrow{\cancel{e(2.2)}}$

(*n* , *x*=2.4 , *y*=3.1415) $\xrightarrow{e(1.1)}$ (*n* , *x*=3.5 , *y*=4.2415)

En invariant används för att tvinga systemet att utföra en övergång (dvs lämna tillståndet) innan invarianten blir falsk!!

Tidsautomater i UPPAAL

- Tidsautomater med Invarianter
  + urgent actions
  + urgent locations
  + committed locations
  + data variabler
  + arrayer av data variabler (heltal med bundna domäner)
  + villkor och tilldelningar med data-variabler
  + templates med lokala klockor, data variabler, och konstanter

Deklarationer i UPPAAL

- Syntaxen som används i UPPAAL är C-liknande.
- **Klockor**:
  - **Syntax:**

```
clock x1, …, xn ;
```

  - **Exempel:**
  - `clock x, y;` **Deklarerar två klockor: x och y.**

Deklarationer i UPPAAL (forts.)

- **Datavariabler**
  - **Syntax:**

```
int n1, … ;
int[l,u] n1, … ;
int n1[m], … ;
```

Heltal med "default" domän.
Heltal med domän från "l" till "u".
Heltalsarray n1[0] till n1[m-1].

  - **Exempel;**
  - `int a, b;`
  - `int[0,1] a, b[5];`

Deklarationer i UPPAAL (forts.)

- Händelser (eller Kanaler):
  - **Syntax:**

```
chan a, … ;
urgent chan b, … ;
```

Vanliga kanaler.
Urgent kanaler (se senare bild)

  - **Exempel:**
  - `chan a, b;`
  - `urgent chan c;`

Deklarationer i UPPAAL (forts.)

- Konstanter
  - **Syntax:**

```
const c1 n1, …, ck nk;
```

  - **Exempel:**
  - `const true 1, false 0;`

Tidsautomater i UPPAAL

Klocktilldelningar

$x := n$

Variabeltilldelningar

$i := Expr$

$Expr ::= i \mid i[Expr] \mid$

$n \mid -Expr \mid$

$Expr + Expr \mid$

$Expr - Expr \mid$

$Expr * Expr \mid$

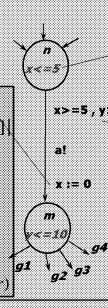
$Expr / Expr \mid$

$(g_d ? Expr : Expr)$

Tillståndsinvarianter

$inv ::= x < n \mid x <= n \mid inv, inv$

klocka naturligt tal "and"



$g ::= g_c \mid g_d \mid g, g$

$g_c ::= x \otimes n \mid x \otimes y + n$ — klockvillkor

$g_d ::= Expr\ op\ Expr$ — variabelvillkor

$\otimes \in \{<, <=, ==, >=, >\}$

$op \in \{<, <=, ==, >=, >, !=\}$

Tidsautomater i UPPAAL

Klocktilldelningar

$x := n$

Variabeltilldelningar

$i := Expr$

$Expr ::= i \mid i[Expr] \mid$

$n \mid -Expr \mid$

$Expr + Expr \mid$

$Expr - Expr \mid$

$Expr * Expr \mid$

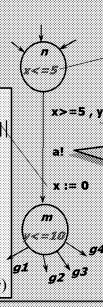
$Expr / Expr \mid$

$(g_d ? Expr : Expr)$

Tillståndsinvarianter

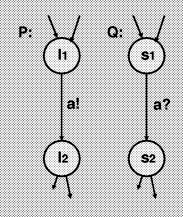
$inv ::= x < n \mid x <= n \mid inv, inv$

klocka naturligt tal "and"



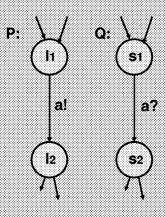
Actions:
- "a" är action namn
- a! eller a?
- exakt en per övergång

Urgent Action: Exempel 1



- Antag att de två övergångarna i automaterna P och Q skall tas så fort som möjligt,
- dvs så snart båda automaterna är redo (är i l_1 och s_1 samtidigt).
- Hur modellera med invarianter om det är okänt (eller varierar) vem som först når l_1 resp. s_1?

Urgent Action: Exempel 1



- Antag att de två övergångarna i automaterna P och Q skall tas så fort som möjligt,
- dvs så snart båda automaterna är redo (är i l_1 och s_1 samtidigt).
- Hur modellera med invarianter om det är okänt (eller varierar) vem som först når l_1 resp. s_1?
- **Lösning**: deklarera action "a" som *urgent*.

Urgent Action

```
urgent chan hurry;
```

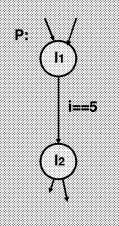
Informell Semantik:

- Inget delay kan ske om övergång med urgent action kan utföras.

Begränsningar:

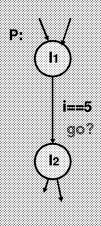
- Inget klockvillkor får förekomma på övergång med urgent action.
- Invarianter och variabelvillkor kan användas som vanligt.

Urgent Action: Exempel 2

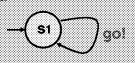


- Antag att i är en datavariabel.
- Vi vill att P skall ta övergången från l1 till l2 så snart i==5.

Urgent Action: Exempel 2



- Antag att i är en datavariabel.
- Vi vill att P skall ta övergången från l1 till l2 så snart i==5.
- **Lösning**: P kan tvingas att ta övergången om man lägger till en automat:



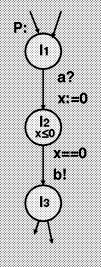
där "go" är en urgent kanal, och "go?" läggs till övergången l1→l2 i P.

Urgent Tillstånd: Exempel

- Antag att vi vill modellera ett enkelt medium M,

tar emot meddelanden på kanal a och sänder omedelbart vidare på kanal b.

- P modellerar mediet m h a klocka x.

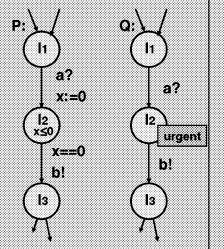


Urgent Tillstånd: Exempel

- Antag att vi vill modellera ett enkelt medium M,

tar emot meddelanden på kanal a och sänder omedelbart vidare på kanal b.

- P modellerar mediet m h a klocka x.
- Q modellerar mediet m h a ***urgent tillstånd***.
- P och Q har samma beteende.



Urgent Tillstånd

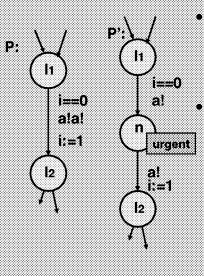
Klicka på "Urgent" i State Editorn.

Informell Sematik:

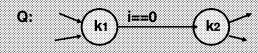
- Inget delay kan ske i urgent tillstånd.

Obs: användandet av urgent tillstånd kan reducera antalet klockor i modellen och därmed den tid och utrymme som krävs för att analysera modellen.

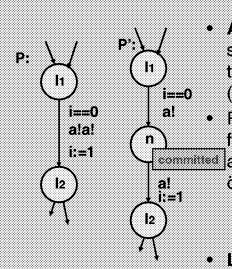
Committed Tillstånd: Exempel 1



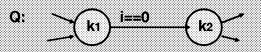
- **Antag:** vi vill modeller en process (P) som sänder ut ett meddelande (a) till två mottagande processer samtidigt (när i==0).
- P' sänder a två gånger utan fördröjning, men i tillståndet "n" kan andra processer, t ex Q utföra övergångar (vilket är fel):



Committed Tillstånd: Exempel 1



- **Antag:** vi vill modeller en process (P) som sänder ut ett meddelande (a) till två mottagande processer samtidigt (när i==0).
- P' sänder a två gånger utan fördröjning, men i tillståndet "n" kan andra processer, t ex Q utföra övergångar (vilket är fel):



- **Lösning:** markera n "committed" i processen P' (istället för "urgent").

Committed Locations/Tillstånd

Klicka på "Committed" i State Editor.

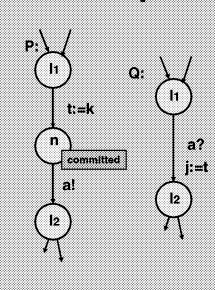
Informell Semantik:

- Inget delay kan ske i committed tillstånd..
- Nästa övergång måste ske i en automat som befinner sig i committed tillstånd.

Obs: användandet av urgent tillstånd kan reducera antalet klockor i modellen och därmed den tid och utrymme som krävs för att analysera modellen.

Committed Tillstånd: Exempel 2

- **Antag:** vi vill skicka heltalet "k" från automat P till variabeln "j" i Q.
- Värdet överförs via den globala heltalsvariabeln "t".
- Tillståndet "n" är committed för att säkerställa att ingen annan automat ändrar värdet på "t" innan tilldelningen "j:=t".



Utökningar i UPPAAL 3.4 (beta)

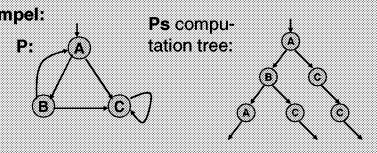
- New operators (not clocks):
  - Logical:
    - && (logical and), || (logical or), ! (logical negation),
  - Bitwise:
    - ^ (xor), & (bitwise and), | (bitwise or),
  - Bit shift:
    - << (left), >> (right)
  - Numerical:
    - % (modulo), ? (max)
  - Assignments:
    - +=, -=, *=, /=, ^=, <<=, >>=, :=
  - Prefix and postfix:
    - ++ (increment), -- (decrement)

Utökningar i UPPAAL 3.4 (forts.)

- Multi dimensional arrays
  e.g. int b[4][2];
- Array initialiser:
  e.g. int b[4] := { 1, 2, 3, 4 };
- Arrays of channels, clocks, constants.
  e.g.
  - chan a[3];
  - clock c[3];
  - const k[3] { 1, 2, 3 };
- Broadcast channels.
  e.g. broadcast chan a;

UPPAALs Specifikationsspråk

- Krav beskrivs som logiska formler.
- En delmängd av logiken TCTL (Timed Computation Tree Logic) används i UPPAAL.
- Formlerna uttrycks i det "computation tree" som en automat genererar.
- **Exempel:**



Kvatifiering i TCTL

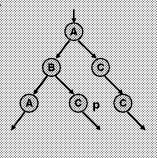
- E - existerar väg genom trädet ("**E**" i UPPAAL).
- A - för alla vägar genom trädet ("**A**" i UPPAAL).
- G - alla tillstånd i en väg ("**[]**" i UPPAAL).
- F - något tillstånd i en väg ("**<>**" i UPPAAL).

I UPPAAL kan följande kombinationer användas:

- **A[], A<>, E<>,** och **E[].**

E<>p – "p Reachable"

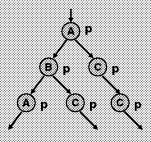
- **E<> p** – ett tillstånd där p är sant kan nås (eng. "reachable").



- P är sann i minst ett tillsånd i trädet (som kan nås från initialtillståndet).

A[]p – "Invariantly p"

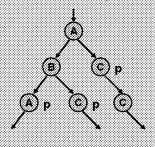
- **A[] p** – p är invariant, dvs håller alltid (eng. "invariantly").



- P är sann i alla nåbara tillstånd i trädet.

A<>p – "Inevitable p"

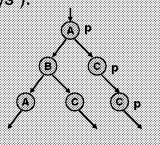
- **A<> p** – p är ofrånkomlig, dvs automaten är garanterad att nå ett tillstånd där p är sann (eng. "inevitably").



- P är sann någon gång i alla vägar genom trädet.

E[] p – "Potentially Allways p"

- **E[] p** – p kan vara sann hela tiden (eng. "potentially allways").



- Det finns en väg genom trädet där p håller i alla tillstånd.

UPPAAL Lokala Egenskaper

- **A[]p, A<>p, E<>p, E[]p** – **p** är en lokal egenskap
- **Syntax:**

processtillstånd, variabelvillkor, klockvillkor

```
p::= a.l | gd | gc | p and p |
      p or p | not p | p imply p |
      ( p )
```

processnamn

UPPAAL Demo: Two Small Examples

Paul Pettersson
Dept. of Information Technology
Uppsala University

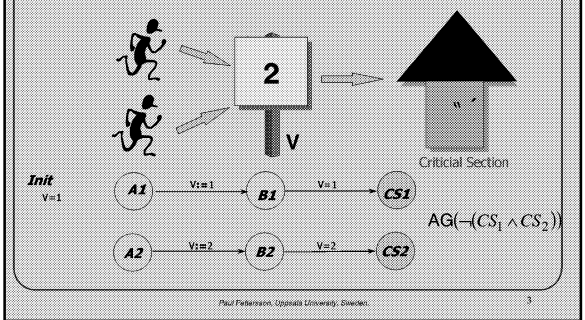
http://www.docs.uu.se/~paupet/
mailto:paupet@docs.uu.se

Example 1: Fischer's Mutual Exclusion Protocol

From lecture 1!

Fischer's Protocol

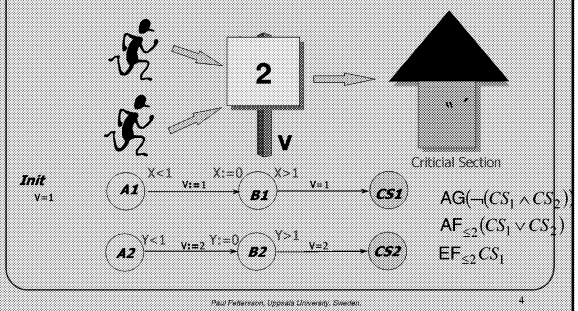
A simple MUTEX Algorithm



$AG(\neg(CS_1 \wedge CS_2))$

Fischer's Protocol

A simple MUTEX Algorithm



$AG(\neg(CS_1 \wedge CS_2))$

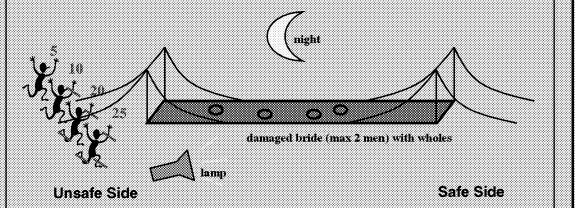
$AF_{\leq 2}(CS_1 \vee CS_2)$

$EF_{\leq 2} CS_1$

Example 2: The Bridge Problem

Find schedule for four men to cross bridge before hard deadline.

Example: Bridge Problem

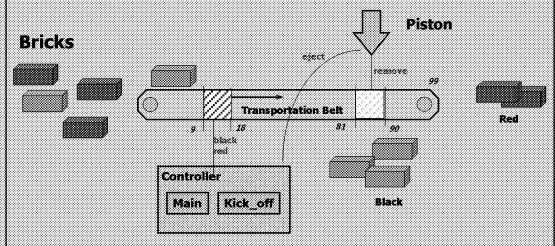


If possible find schedule for all four men to reach safe side in 60 min.

Example 3: Brick Sorter

Real-Time Controller sorting **black** bricks from red bricks.

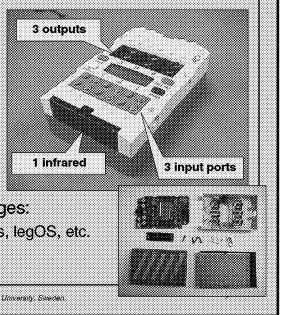
Example: Brick Sorter



Problem: Design **Controller** that kicks all (and only) black bricks off the belt.

LEGO Mindstorms/RCX

- Processor: Hitachi H8 with
  - 32K RAM,
  - 16K ROM.
- Sensors: temperature, light, rotation, pressure, etc.
- Actuatorer: motor, lamp, etc.
- Infrared Communication
- 6 x 1.5V Power Pack
- Several Programmering Languages:
  - Not Quite C, Mindstroms, Robotics, legOS, etc.



NQC Program

```
int active;
int DELAY;
int LIGHT_LEVEL;
```

```
task main{
 DELAY=23;
 LIGHT_LEVEL=48;
 active=0;
 Sensor(IN_1, IN_LIGHT);
 Fwd(OUT_A,1);
 Display(1);

 start kick_off;

 while(true){
   wait(IN_1<=LIGHT_LEVEL);
   ClearTimer(1);
   active=1;
   PlaySound(1);
   wait(IN_1>LIGHT_LEVEL);
  }
}
```

```
task kick_off{
  while(true){
    wait(Timer(1)>DELAY && active==1);
    active=0;
    Rev(OUT_C,1);
    Sleep(8);
    Fwd(OUT_C,1);
    Sleep(12);
    Off(OUT_C);
  }
}
```

Model Checking Timed Systems using Clock Constraints (part 1)

Reachability Analysis and Constraint solving

Problem: reachability analysis

- Give an automaton and a location n, or a local property F
- Question: does it exist an execution of the automaton, that leads to n (or a state where F holds)?
- This is the so called reachability problem.

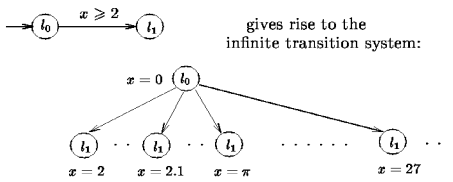
Formalizing requirements

- Reachability properties: E<> Q
  - E<> P.stop
  - E<> (y>200)
- Invariant properties: A[] Q (not E<> not Q)
  - A[] not (P1.CS and P2.cs)
  - A[] (i < 100)
  - A[] (x>10 imply i>100)
    - After 10:00AM, i should be larger than 100
- Bounded Liveness Properties: F1→$_{<=10}$ F2
  - A[](f1 and x>10 imply f2)

Other Verification Problems

- Timed Language Inclusion ☹
- Untimed Language Inclusion ☺
- (Un)Timed Bisimulation ☺
- Reachability Analysis ☺
- Optimal Reachability (synthesis problem) ☺
  - If a location is reachable, what is the minimal delay before reaching the location?

Infinite State Space!

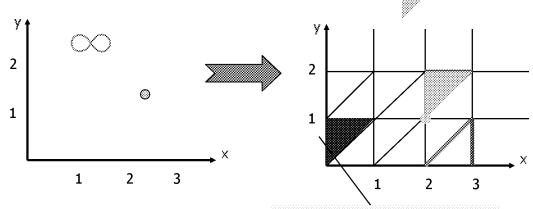


However, the reachability problem is decidable ☺ Alur&Dill 1991

Region: From infinite to finite

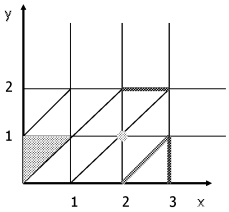
Concrete State
(n, x=2.2, y=1.5)

Symbolic state (region)
(n,)



An equivalence class (i.e. a *region*)
There are only *finite* many such!!

Region equivalence (Intuition)



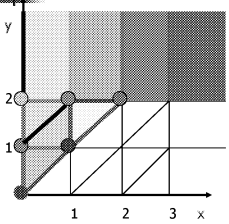
u ≅ v iff u and v satisfy exactly the same set of constraints in the form of
xi ~ m and xi-xj ~ n
where ~ is in {<,>,≤,≥}
and m,n < MAX

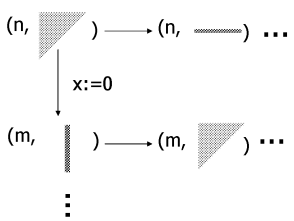
This is not quite correct; we need to consider the MAX more carefully

Region equivalence: Definition *[Alur and Dill 1990]*

- u,v are clock assignments
- u≈v if (1) and (2) hold
  - (1) for each clock x,
    if u(x)<=Cx then
    - ⌊u(x)⌋=⌊v(x)⌋ (the same integer part)
  - (2) for each pair clocks x, y
    if u(x)<=Cx and u(y)<=Cy then
    - {u(x)}=0 iff {v(x)}=0
    - {u(x)}<= {u(y)} iff {v(x)}<= {v(y)}

Regions
Finite partitioning of state space



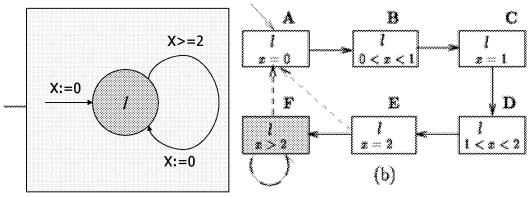


OBS: there are only Finite many regions

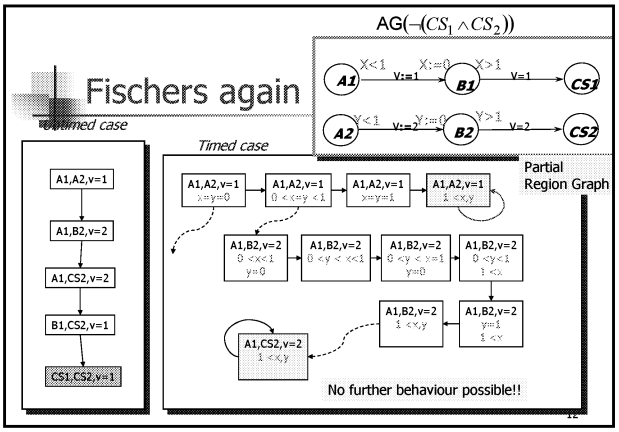
An Important Theorem for Region Equivalence

- u≈v implies
  - u+d≈v+d for all d=>0
  - u(x:=0) ≈ v(x:=0)
- that is, 'region equivalence' is preserved by addition and reset
- in fact, it is also preserved by subtraction if clock values are 'bounded'

Region graph of a simple timed automata



Fischers again



Problems with Region Partitioning

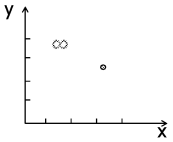
- Too many 'regions'
- Sensitive to the maximal constants
  - e.g. x>1000000
- The number of regions is highly exponential in the number of clocks and the maximal constants (used to compare with clocks)

The more efficient solution *[1989 Dill … 1994]*

Symbolic Reachability Using Clock Constraints

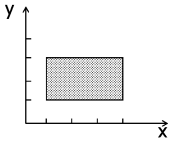
Zones: From infinite to finite

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

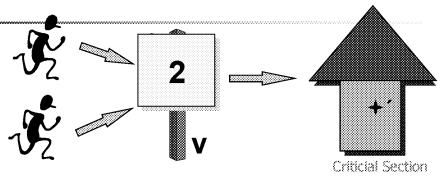


Symbolic state (zone)
(n, $1 \le x \le 4, 1 \le y \le 3$)

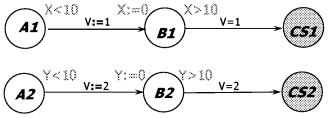
Zone:
conjunction of
x-y~n, x~n



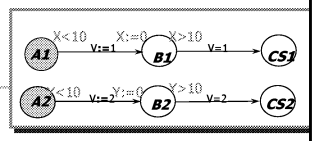
Fischer's Protocol
analysis using zones



Initially
V=1



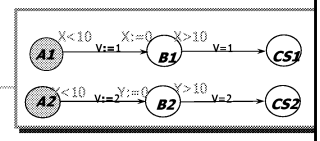
Fischers cont.



Untimed case

A1,A2,v=1 → A1,B2,v=2 → A1,CS2,v=2 → B1,CS2,v=1 → CS1,CS2,v=1

Fischers cont.

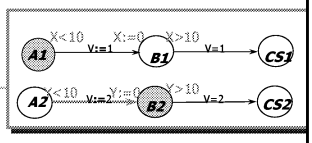


Untimed case

A1,A2,v=1 → A1,B2,v=2 → A1,CS2,v=2 → B1,CS2,v=1 → CS1,CS2,v=1

Taking time into account

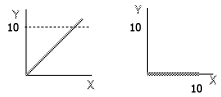
Fischers cont.



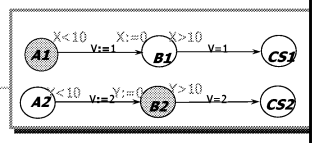
Untimed case

A1,A2,v=1 → A1,B2,v=2 → A1,CS2,v=2 → B1,CS2,v=1 → CS1,CS2,v=1

Taking time into account



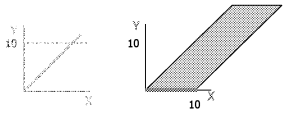
Fischers cont.



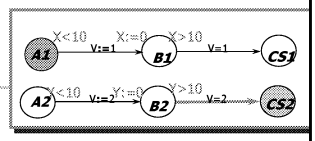
Untimed case

A1,A2,v=1 → A1,B2,v=2 → A1,CS2,v=2 → B1,CS2,v=1 → CS1,CS2,v=1

Taking time into account



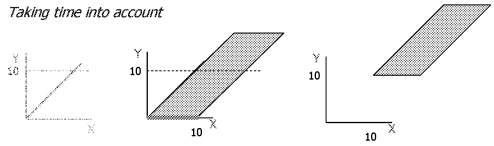
Fischers cont.



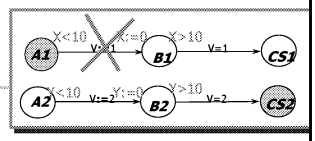
Untimed case

A1,A2,v=1 → A1,B2,v=2 → A1,CS2,v=2 → B1,CS2,v=1 → CS1,CS2,v=1

Taking time into account



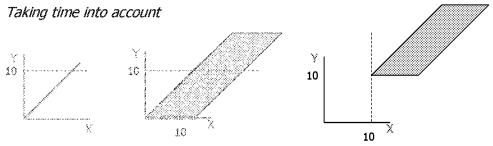
Fischers cont.



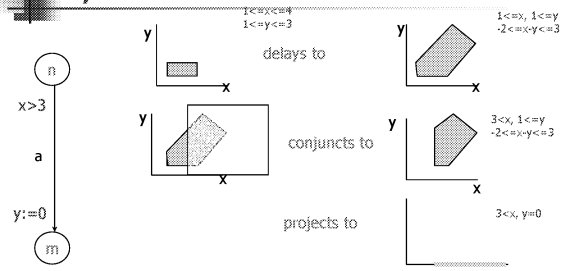
Untimed case

A1,A2,v=1 → A1,B2,v=2 → A1,CS2,v=2 → B1,CS2,v=1 → CS1,CS2,v=1

Taking time into account



Symbolic Transitions



Thus (n,1<=x<=4,1<=y<=3) =a=> (m,3<x, y=0)

Zones = Conjuctive constraints

- A zone Z is a conjunctive formula:
  g_1 & g_2 & ... & g_n
  where g_i is a clock constraint:
  $x_i \sim b_i$ 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 $\mathbb{Z}$ be a zone (a set of constraints)
- Let $[\mathbb{Z}]$={u | u is a solution of Z}
  - The semantics

(We shall simply write $\mathbb{Z}$ instead $[\mathbb{Z}]$)

Operations on Zones

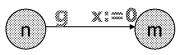
- Strongest post-condition (Delay): SP(Z) or Z↑
  - $[Z\uparrow]$ = {u+d| d ∈ R, u∈[Z]}
- Weakest pre-condition: WP(Z) or Z↓ (the dual of Z↑)
  - $[Z\downarrow]$ = {u| u+d∈[Z] for some d∈R}
- Reset: {x}Z or Z(x:=0)
  - [{x}Z] = {u[0/x] | u ∈[Z]}
- Conjunction
  - [Z&g]= [Z]∩[g]

An important 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]

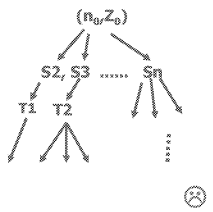
One-step reachability: Si→Sj

- Delay: (n,Z) → (n,Z') where Z'= $Z\uparrow \wedge$ inv(n)
- Action: (n,Z) → (m,Z') where Z'= {x}(Z ∧g)

  if 

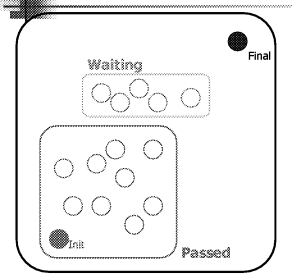
- Successors(n,Z)={(m,Z') | (n,Z) →→(m,Z'), Z'≠Ø}
  - Sometime we write: (n,Z)→(m,Z') if (m,Z') is a successor of (n,Z)

Now, we have a search problem



Forward Rechability

Init -> Final ?



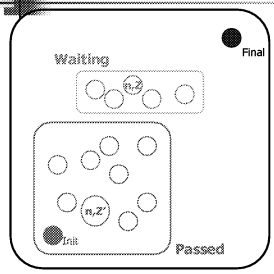
INITIAL **Passed** := Ø;
Waiting := {(n0,Z0)}

REPEAT
- pick (n,Z) in **Waiting**
- if for some Z' ⊒ Z
  (n,Z') in **Passed** then STOP
- else (explore) add
  successors(n,Z) to **Waiting**;
  Add (n,Z) to **Passed**

UNTIL **Waiting** = Ø
or
Final is in **Waiting**

Forward Rechability

Init -> Final ?



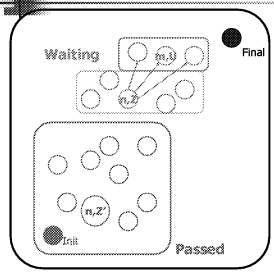
INITIAL Passed := Ø;
Waiting := {(n0,Z0)}

REPEAT
- pick (n,Z) in Waiting
- **if** for some Z' ⊇ Z
  (n,Z') in Passed **then STOP**

UNTIL Waiting = Ø
or
Final is in Waiting

Forward Rechability

Init -> Final ?



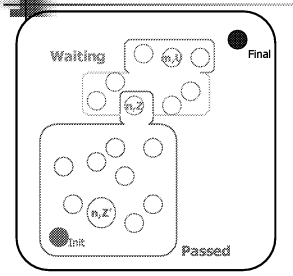
INITIAL Passed := Ø;
Waiting := {(n0,Z0)}

REPEAT
- pick (n,Z) in Waiting
- **if** for some Z' ⊇ Z
  (n,Z') in Passed **then STOP**
- **else** /explore/ add
  successors(n,Z) to Waiting;

UNTIL Waiting = Ø
or
Final is in Waiting

Forward Rechability

Init -> Final ?



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

REPEAT
- pick (n,Z) in Waiting
- **if** for some Z' ⊇ Z
  (n,Z') in Passed **then STOP**
- **else** /explore/ add
  successors(n,Z) to Waiting;
  Add (n,Z) to Passed

UNTIL Waiting = Ø
or
Final is in Waiting

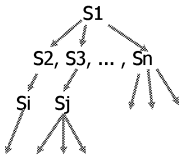
Two more operations on Zones

- Inclusion checking: $Z_1 \subseteq Z_2$
  - solution sets
- Emptiness checking: $Z = \emptyset$
  - no solution

All Operations on Zones

(needed for reachability analysis)

- Transformation
  - Conjunction
  - Post condition (delay)
  - Reset
- Consistency Checking
  - Inclusion
  - Emptiness



EFFICIENT IMPLEMENTATION

Canonical Datastructures for Zones

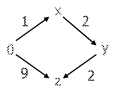
Difference Bounded Matrices Bellman 1958, Dill 1989

Inclusion

Z1

x<=1
y-x<=2
z-y<=2
z<=9

Graph

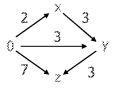


? ⊆ ?

Z2

x<=2
y-x<=3
y<=3
z-y<=3
z<=7

Graph



Canonical Dastructures for Zones

Bellman 1958, Dill 1989

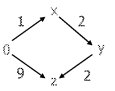
Difference Bounded Matrices

Inclusion

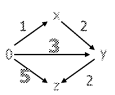
Z1

x<=1
y-x<=2
z-y<=2
z<=9

Graph



Shortest Path Closure



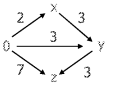
? ⊆ ?

Z1 ⊆ Z2 !

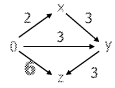
Z2

x<=2
y-x<=3
y<=3
z-y<=3
z<=7

Graph



Shortest Path Closure



Canonical Datastructures for Zones

Difference Bounded Matrices Bellman 1958, Dill 1989

Emptiness

Z

x<=1
y>=5
y-x<=3

Graph

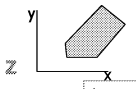
Negative Cycle
iff
empty solution set

Connect

Canonical Datastructures for Zones

Difference Bounded Matrices

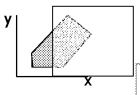
Conjunction



Z

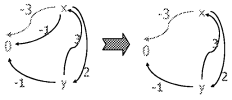
1<=x, 1<=y
-2<=x-y<=3

Z∧g



1<=x, 1<=y
-2<=x-y<=3
3<=x

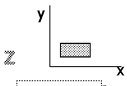
Add new edge for g



Canonical Dastructures for Zones

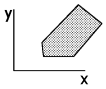
Difference Bounded Matrices

Delay



Z

1<= x <=4
1<= y <=3



Z ↑

1<=x, 1<=y
-2<=x-y<=3

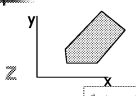
Shortest Path Closure

Remove upper bounds on clocks

Canonical Datastructures for Zones

Difference Bounded Matrices

Reset



Z

1<=x, 1<=y
-2<=x-y<=3

{y}Z

y=0, 1<=x

Remove all bounds involving y and set y to 0

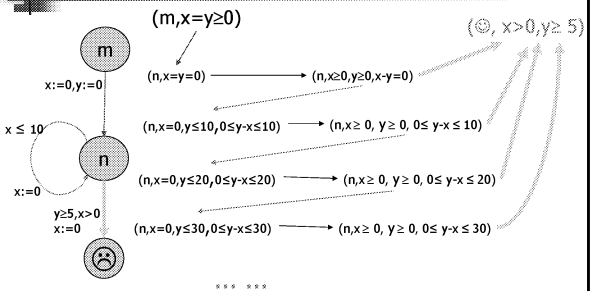
COMPLEXITY

- Computing the shortest path closure, the cannonical form of a zone: $O(n^3)$ [Dijkstra's alg.]
- Run-time complexity, mostly in $O(n)$ (when we keep all zones in cannonical form)

How about termination?

We need the normalization operation according to the maximal constant

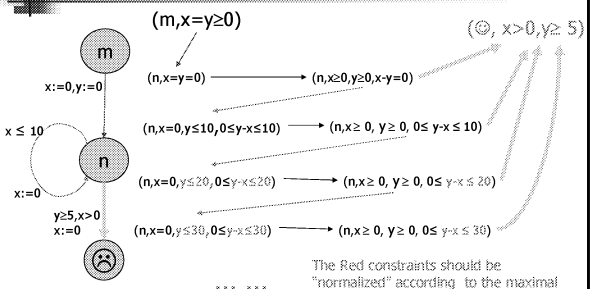
Example: is ☹ reachable?



The search process may never terminate

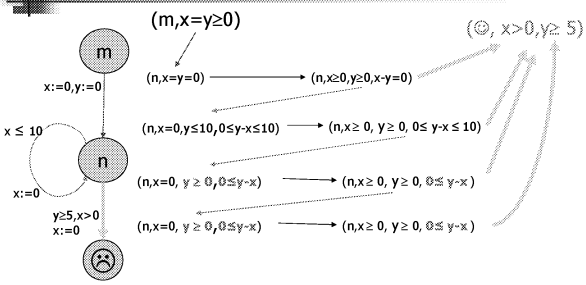
- The new zones created (due to the red edge/transitions) is getting larger and larger … …
- Note that in this example, Breadth-first may terminate the search with result: yes (i.e. ☺ is reachable)
- But in general, there is no guarantee for termination even using Breadth-first e.g.
  - Consider the same example and check the reachability of (☺,x>y) (it is NOT reachable; then the search will try to construct the whole state space)
- We need a solution !! (using the Maximal constant)

Example: is ☹ reachable?

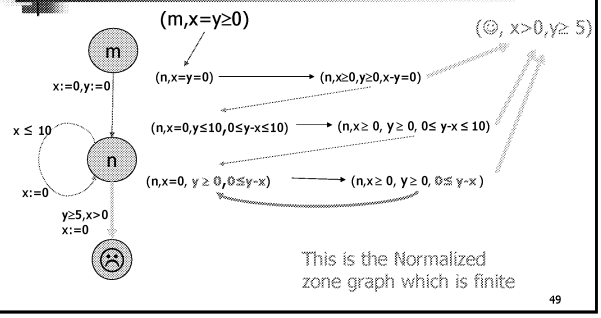


The Red constraints should be "normalized" according to the maximal constant 10

Example: is ☹ reachable?



Example: is ☹ reachable?



This is the Normalized zone graph which is finite

The normalization Operation

(For automata containing no constraints on clock differences, Only)

- First compute the shortest path closure of a zone
- Remove all constraints in the form:
  Xi <(≤) m or Xi-Xj <(≤) n
  where m, n>MAX
- Replace all constraints in the form:
  Xi >(≥) m or Xi-Xj >(≥) n
  where m, n>MAX

with Xi > MAX or Xi-Xj > MAX

The number of "Normalized Zones" is bounded

By the number of regions !

NOW, YOU CAN GO HOME
to MAKE YOUR OWN
MODEL CHECKER

Model Checking Timed Systems using Clock Constraints (part 2)

Reduction and Optimization

Optimizations/Reductions

(UPPAAL Specific)

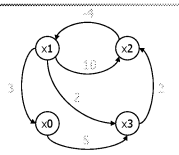
- Minimal Constraints
- Committed Locations
- Global (loop) Reductions
- (In)active clock reduction
- Approximation
  - Bitstate Hashing (under-approximation)
  - Convex-Hull (over approximation)
- Re-Use of state space

Improved Datastructures

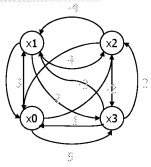
Compact Datastructure for Zones

RTSS 1997

x1-x2<=-4
x2-x1<=10
x3-x1<=2
x2-x3<=2
x0-x1<=3
x3-x0<=5



Shortest Path Closure O(n^3)

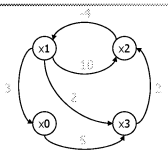


Improved Datastructures

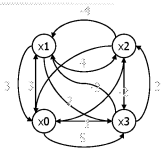
Compact Datastructure for Zones

RTSS 1997

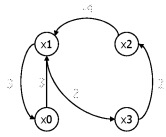
x1-x2<=-4
x2-x1<=10
x3-x1<=2
x2-x3<=2
x0-x1<=3
x3-x0<=5



Shortest Path Closure O(n^3)



Shortest Path Reduction O(n^3)

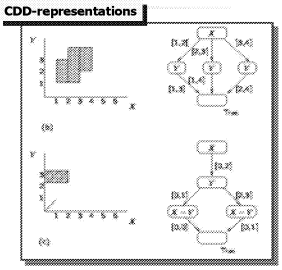


Canonical wrt =
Space worst O(n^2)
practice O(n)

Other Symbolic Datastructures

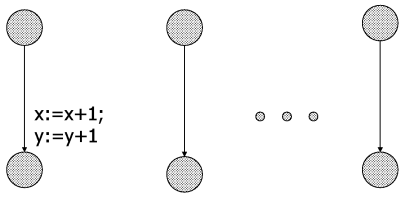
- Regions Alur, Dill
- NDD's Maler et. al.
- CDD's UPPAAL/CAV99
- DDD's Møller, Lichtenberg
- .....

CDD-representations



Committed Locations

(example: atomic sequence in a network)

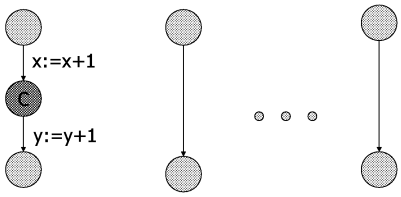


If the sequence becomes too long, you can split it ...

Committed Locations

(example: atomic sequence in a network)

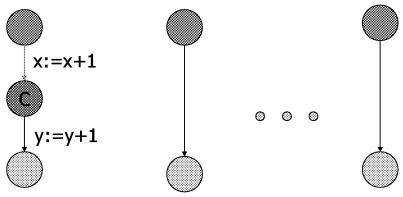
Semantics: the time spent on C-location should be zero !



Committed Locations

(example: atomic sequence in a network)

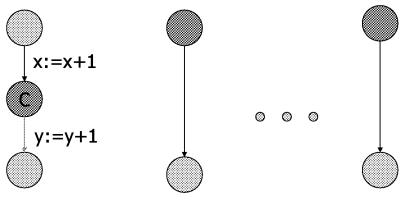
Semantics: the time spent on C-location should be zero !



Committed Locations

(example: atomic sequence in a network)

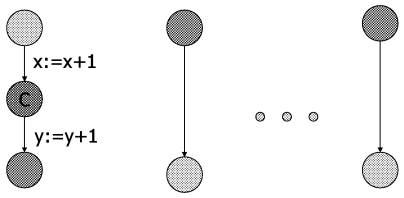
Semantics: the time spent on C-location should be zero !



Now, only the committed (red) transition can be taken!

Committed Locations

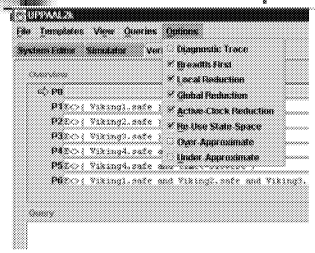
(example: atomic sequence in a network)



Committed Locations

- A trick of modeling (e.g. to model multi-way synchronization using handshaking)
- More importantly, it is a simple and efficient mechanism for state-space reduction!
  - In fact, it is a simple form of 'partial order reduction'
- It is used to avoid intermediate states, interleavings:
  - Committed states are not stored in the passed list
  - Interleavings of any state with a committed location will not be explored

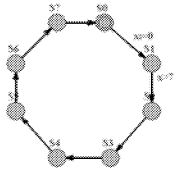
Verification Options



- Diagnostic Trace
- Breadth-First
- Depth-First
- Local Reduction
- Active-Clock Reduction
- Global Reduction
- Re-Use State-Space
- Over-Approximation
- Under-Approximation

Case Studies

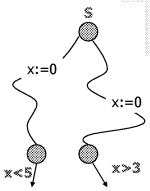
Inactive (passive) Clock Reduction



x is only *active* in location S1

Definition

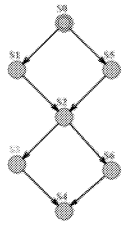
x is *inactive* at S if on all path from S, x is always reset before being tested.



Case Studies

Global Reduction

(When to store symbolic state)

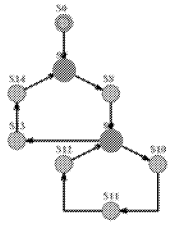


However,
Passed list useful for efficiency

Case Studies

No Cycles: **Passed** list not needed for *termination*

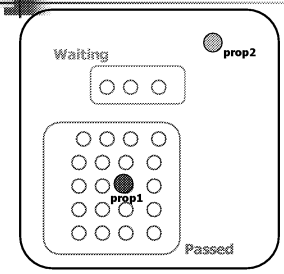
Global Reduction
(When to store symbolic state)



Cycles:
Only symbolic states involving loop-entry points need to be saved on **Passed** list

Case Studies

Reuse of State Space

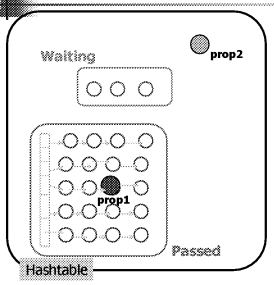



```
A[]  prop1

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

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

Which order to search?

Reuse of State Space

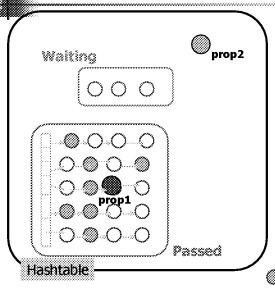



```
A[]  prop1

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

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

Which order to search?

Reuse of State Space



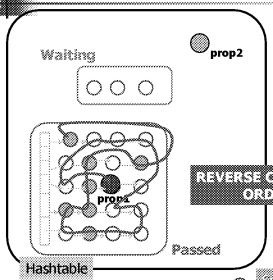

```
A[]  prop1

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

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

Which order to search?

Swapped to secondary memory

Reuse of State Space



REVERSE CREATION ORDER

```
A[]  prop1

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

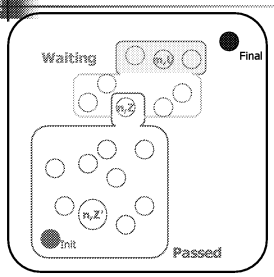
Search in existing **Passed** list before continuing search

Which order to search?

generation order

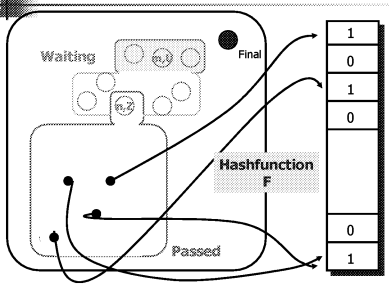
Swapped to secondary memory

Under-approximation
Bitstate Hashing (Holzman,SPIN)



Under-approximation

Bitstate Hashing



Passed=
Bitarray

UPPAAL
8 Mbits

Bitstate Hashing

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

REPEAT
- pick (n,Z) in Waiting
- **if** for some Z' ⊇ Z
  (n,Z') in Passed **then STOP**
- **else** /explore/ add
  { (m,U) : (n,Z) => (m,U) }
  to Waiting;
  Add (n,Z) to Passed

UNTIL Waiting = Ø
or
Final is in Waiting

Passed(F(n,Z)) = **1**

Passed(F(n,Z)) := **1**

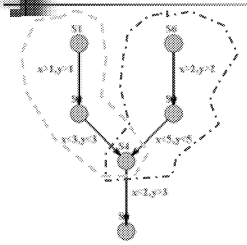
Under Approximation

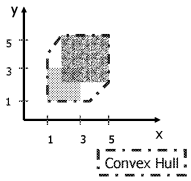
(good for finding Bugs quickly, debugging)

- Possitive answer is safe (you can trust)
  - You can trust your tool if it tells:
    a state is reachable (it means Reachable!)
- Negative answer is Inconclusive
  - You should not trust your tool if it tells:
    a state is non-reachable
  - Some of the branch may be terminated by conflict (the same hashing value of two states)

Over-approximation

Convex Hull





Case Studies

Over-Approximation

(good for safety property-checking)

- Possitive answer is Inconclusive
  - a state is reachable means Nothing
    (you should not trust your tool when it says so)
  - Some of the transitions may be enabled by Enlarged zones
- Negative answer is safe
  - a state is not reachable means Non-reachable
    (you can trust your tool when it says so)

NOW, YOU CAN GO HOME
to MAKE a good MODEL CHECKER
☺

Lecture 5 (on going research)

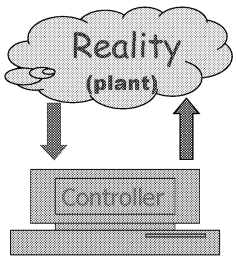
Unification of Scheduling, Model-Checking and Code Synthesis:
From UPPAAL to TIMES

"Who is Who" in Real Time Systems

- Real Time Scheduling [RTSS ...]
  - Task models, Schedulability analysis
  - Real time operating systems
- Automata/logic-based methods [CAV,TACAS ...]
  - FSM, PetriNets, Statecharts, Timed Automata
  - Modelling, Model checking ...
- (RT) Programming Languages [...]
  - Esterel, Signal, Lustre, Ada ...
- ... ...

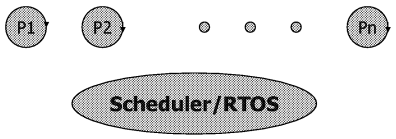
The Same Goal: Reliable Controllers

(with minimal resource consumption)



Real Time Scheduling: 'reality' is periodic/cyclic

- Controller = a set of periodic tasks + a scheduler



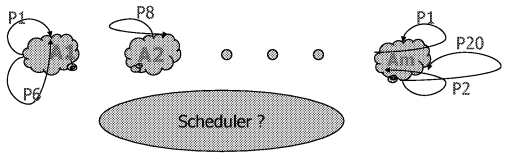
- well-developed techniques e.g. Rate-Monotonic Scheduling

What to do if the 'reality' is non-periodic?

In real life, tasks may
- share have many resources (CPU time, memory, power ...)
- have complex control stuctures (e.g. phone call)
- have to satisfy mixed logical, temporal and resouce constraints ... ...

Automata-based Approaches

- Controller = a set of timed automata
  - accepting 'events' and triggering tasks Pi's



- How to schedule tasks/automata? Schedulability? Executable code ?

TIMES

Tools for Modeling and Implementation of Embedded Systems

Joint work with President of Russia


Leonid Mokrushin Elena Fersman







Tobias Amnell Johan Bengtsson Alexandre David John Håkansson Annika Karlsson Paul Petterssn Wang Yi

Uppsala University, Sweden

Vision

Timed Model to (Verified) Executable Code Guaranteeing Timing Constraints

OUTLINE

- A Unified Model for Timed Systems [1998]
  - Timed automata with tasks
- Schedulability and Decidability [TACAS 02]
  - Timed automata with bounded subtraction
- More Efficient Algoritms [TACAS 03]
  - Schedulability analysis using 2 clocks
  - (Rate-Monotonic Scheduling and Beyond)
- TIMES tool (demo)
  - Schedulability analysis and Code Synthesis

The MODEL

(Timed Automata with Tasks)

Modelling Real Time Systems

- Events
  - synchronization
  - interrupts
- Timing constraints
  - specifying event arrivals
  - e.g. Periodic and sporadic
- Tasks (executable programs)
  - interrupt processing
  - Internal computation
  - triggered by events and scheduled in the reday queue of RTOS

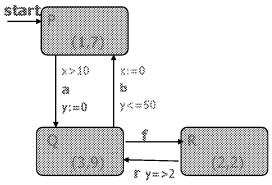
Timed Automaton + tasks

Tasks = Executable Programs (e.g. C, Java)

- Task parameters: C, D etc
  - C: Computing time
  - D: Relative Deadline
  - (other parameters for schedulling e.g. Priority)
- Task Interface: $v_1 \ldots v_n$ (a set of variables updated)

```
Task P
{ ...
  ...
v1 := F1(v1...vn)
...
vn := Fn(v1...vn)
}
```

Timed Automata with Tasks (Example)



- Processor 1 (event handler)
  - Initially, P is in the queue
  - Run-to-Completion/Stablization
    - Whenever a is available and x>10, Q is put in the queue
    - Then
      - Whenever b is available and y<=50, P is put in the queue
      - Whenever f appears, it releases R
- Processor 2 (task handler)
  - Scheduling and Computing tasks in the queue

P	P	Q	Q	Q	R	...	...	...

Timed Automata withTasks [1998]

- Assume a set of tasks Pr
- A timed automaton with tasks is a tuple: $<N,n_0,T,M>$
  - $<N,n_0,T>$ is a standard timed automaton [Alur,Dill 90]
    - N is a set of nodes
    - n_0 is the initial node
    - $T \subseteq N \times \{B(C) \times Act \times 2^C\} \times N$ is the set of 'edges'
      - C is a set of clocks
      - Act is a set of actions
      - B(C) is the set of clock constraints e.g. X <10 etc
  - $M: N \rightarrow 2^{Pr}$ is a mapping which assigns each node a set of tasks

States/Configurations of automata

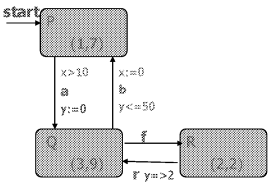
A state is a triple: (m, u, q)

Location (node)

clock assignment (valuation)

task queue

Example



Initial State: (P, x=y=0, [P(1,7)])

Example transitions:

(P, x=y=0, [P(1,7)]) –0.6→

(P, x=y=0.6, [P(0.4, 6.4)]) –9.5→

(P, x=y=10.1, []) –a→

(Q, x=10.1,y=0, [Q(3,9)]) –f→

(R, x=10.1,y=0, [Q(3,9),R(2,2)]) –2→

(R, x=12.1,y=2, [Q(3,7)]) –r→

(Q, x=12.1,y=2, [Q(3,7),Q(3,9)]) –b→

(P, x=0,y=2, [Q(3,7),Q(3,9),P(1,7]) ...

We need to handle the queue by Run and Sch

Sch and Run

- Sch is a function sorting task queues according to a given scheduling strategy e.g FPS,EDF,FIFO etc

  Example: EDF [P(2, 10), Q(4, 7)] = [Q(4, 7), P(2, 10)]

- Run is a function corresponding to running the first task of the queue for a given amount of time.

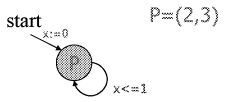
  Examples: Run (0.5, [Q(4, 7), P(2, 10)]) = [Q(3.5, 6.5), P(2, 9.5)]
  Run (5, [Q(4, 7), P(2, 10)]) = [P(1, 5)]

Semantics (as transition systems)

- States: $<m,u,q>$
  - m is a location
  - u is a clock assignment (valuation)
  - q is a queue of tasks (ready to run)
- Transitions:
  1. (m,u,q) –a→ (n, r(u), Sch[M(n)::q]) if $m \xrightarrow{g\ a\ r} n$ & g(u)
  2. (m,u,q) –d→ (m, u+d, Run(d,q)) where d is a real

OBS: q is growing (by actions) and shrinking (by delays)

Zenoness = Non-Schedulability



P=(2,3)

Zeno: ∞ many P's may arrive within 1 time unit !

P(2,3)	P(2,3)	P(2,3)	P(2,3)	P(2,3)	...					

But after 2 copies, the queue will be non-schedulable

SCHEDULABILITY

Schedulability of automata

a state is a triple: (m, u, q)

location

clock assignment

task queue

•A state is schedulable if q is schedulable

•An automaton is schedulable if all reachable states are

Schedulability of Automata

Assume a scheduling policy Sch:

- A state (m,u,q) is schedulable with Sch if
  - Sch(q)= $[P_1(c_1,d_1)P_2(c_2,d_2)...P_n(c_n,d_n)]$ and
  - $(c_1+...+c_i)<=d_i$ for all i<=n (i.e. all deadlines met)
- An automaton is schedulable with Sch if all its reachable states are schedulable
- An automaton is schedulable with a class of scheduling policies if it is schedulable with every Sch in the class.

Other verification/scheduling problems

- Location Reachability (just as for timed automata)
  - a nice property of the model !
- Boundedness of the task queue |q|<M
  - memory requirement
- Schedule synthesis (ongoing work with Thiagu)

DECIDABILITY

Schedulability Analysis (Non-preemptive scheduling)

FACT [1998]

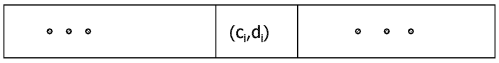
For Non-preemptive scheduling strategies,
the schedulability of an automaton can be checked
by reachability analysis on ordinary timed automata.

NOTE
- Reachability for TA is decidable
- Thus the schedulability checking problem is decidable for a given non preemptive scheduling.

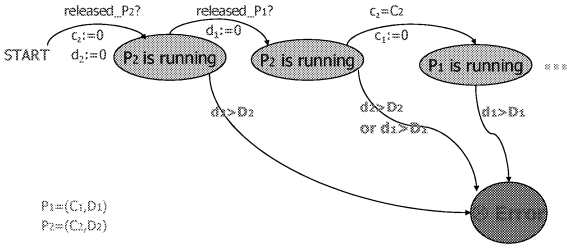
Proof ideas (1):

Size of schedulable queues is bounded

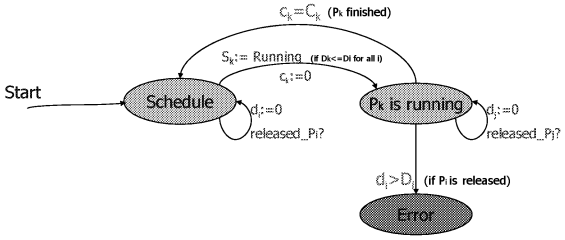
- The maximal number of instances of P_i in a schedulable queue is bounded by $Mi = \lceil Di/Ci \rceil$
- The maximal size of schedulable queues is bounded by $M1 + M2 + \ldots + Mn$
- To code the queue/scheduler, for each task instance, use 2 clocks:
  - c_i remembers the computing time
  - d_i remembers the deadline



Proof ideas (2):
The scheduler as an automaton



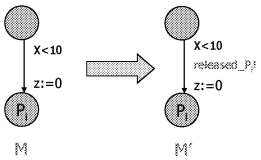
The scheduler automaton



SCHEDULER

Proof Ideas (3)

- Modify the original automaton M: adding 'release!' to inform the scheduler



- Check reachability of the error state for
  M' || SCHEDULER

How about preemptive scheduling?

- We may try the same ideas
  - Use clocks to remember computing times and deadlines
- BUT a running task may be stopped to run a more 'urgent' task
  - Thus we need stop-watches to remember computing times

Conjecture (1998 @ Grenoble, VHS meeting):

- The schedulability problem for Preemptive scheduling is undecidable.
- The intuition: we need stop-watch to code the scheduler and the reachability problem for stop-watch automata is undecidable
- This is wrong !!!

Decidability Result [TACAS 2002]

FACT [2002]

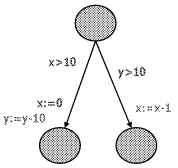
For Preemptive scheduling strategies, the schedulability of an automaton can be checked by reachability analysis on Bounded Substraction Timed Automata (BSA).

NOTE
- Reachability for BSA is decidable
- Preemptive EDF is optimal; thus the general schedulability checking problem is decidable for all scheduling strategies.

Timed automata with subtraction

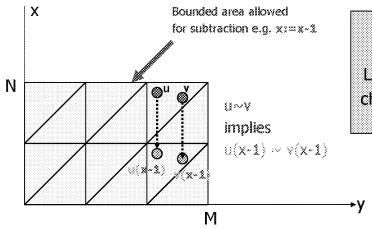
i.e. Subtraction Automata, [McManis and Varaiya, CAV94]

- Subtraction automata are timed automata extended with subtraction on clocks
- That is, in addition to reset x:=0, it is also allowed to update a clock x with x:= x-n where n is a natural number



Bounded Subtraction Automata

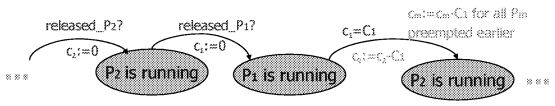
- A subtraction automaton is bounded if its clocks are non-negative and bounded with a maximal constant (or subtraction is only allowed in the bounded zone).



FACT:
Location Reachability checking is decidable!

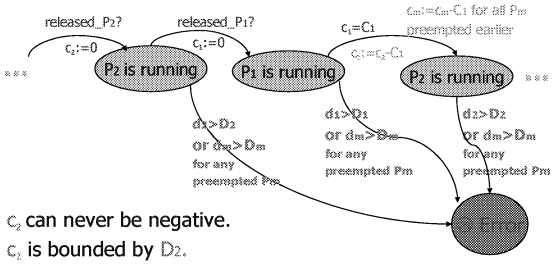
Schedulability Checking as a reachability problem for Bounded Subtraction Automata

Proof ideas (no stop but subtraction :-)



- Model the scheduler as a subtraction automaton
  - Do not stop the computing clock c_1 when a new task P_1 is released
  - Let c_2 for P_2 (preempted) run until the task P_1 (with higher priority) finishes, then perform $c_2:=c_2-C_1$ (note: C_1 is the computing time for P_1).

Proof ideas (clocks are bounded):



- c_2 can never be negative.
- c_2 is bounded by D_2.

DONE

Complexity

#clocks (needed)
= 2 x #instances (maximal number of schedulable task instances)
= 2 x $\sum \lceil D_i/C_i \rceil$

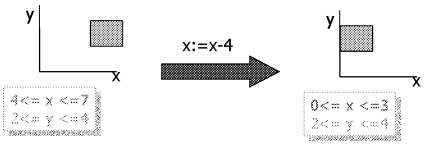
This is a huge number in the worst case
But the run-time complexity is not so bad!

It works anyway !!!

- #active tasks in the queue is normally small, and the run-time complexity is only related to #active clocks
- If Too many active tasks in the queue (i.e. Too many active clocks), the check will stop sooner and report "non-schedulable"
- AND the analysis can be done symbolically!

Schedulability analysis based on Constraints (DBM's)

Subtraction on Clocks, added to DBM-library (UPPAAL, Kronos)



(More on constraints and UPPAAL on Friday)

WE CAN DO BETTER ! [TACAS 03]

For fixed priority scheduling strategies (FPS),
we need only 2 clocks (and ordinary timed automata)!

The "OPTIMAL" SOLUTION

(for fixed-priority scheduling strategies)

Main Idea

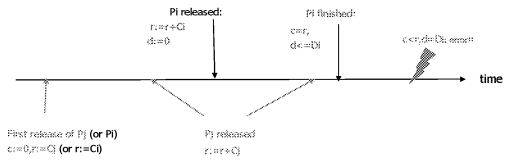
- Check the schedulability of tasks one by one according to priority order (highest priority first)
- This is similar to response time analysis in RMS

To code the queue/scheduler, we need:

- 1 integer variable:
  - r denotes the response time for Pi as in RMS
    (the total computing time needed before Pi finishes)
- 2 clocks:
  - c remembers the accumulated computing time
    (so much has been computed so far)
  - d remembers the "deadline" for the task Pi to be analyzed

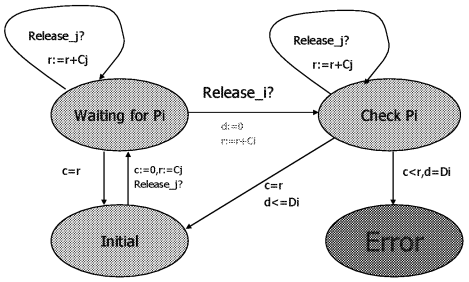
Intuition of the encoding: $R_i = C_i + \Sigma_{pri(P_j)>pri(P_i)} C_j$

– Assume: priority(Pj) > priority(Pi)



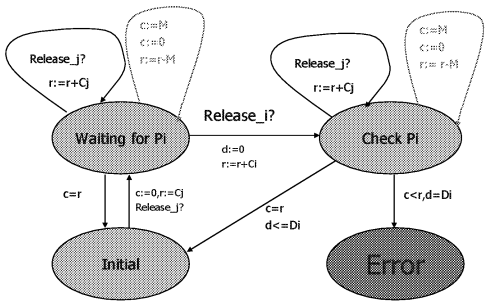
When Pi finishes, r = Ri

The "FPS scheduler": analyzing Pi



Note that it is not clear that c and r are not bounded !

The "FPS scheduler": analyzing Pi
(we need the boundedness)



OBS: r-c is the only interesting info, so M can be any integer! Let M=Ci

c and r are bounded

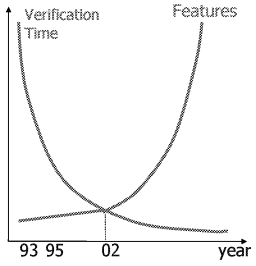
- c is bounded by M
- r is bounded by $r_{max} + C_i$
  - Where r_{max} is the maximal value of r from previous analysis for all tasks P_j with higher priority

So the scheduler is a standard TA END

Conclusions/Comments

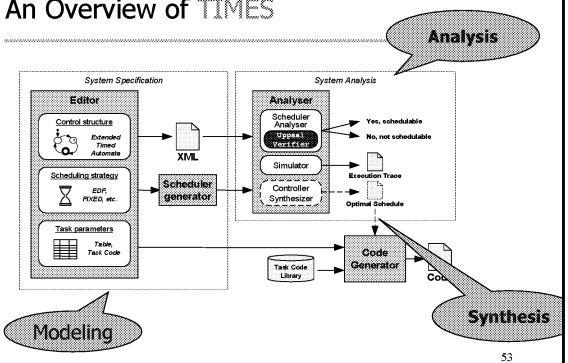
- Schedulability is a fundamental notion for automata: the first decidability result on dense time models for preemptive scheduling
- Unification of model-checking, real time scheduling, and synchronous programming: a unified model for timed systems (can express complex temporal and resource constraints).
- Verification to Synthesis (of verified real time software): **TIMES**

The past and future of UPPAAL



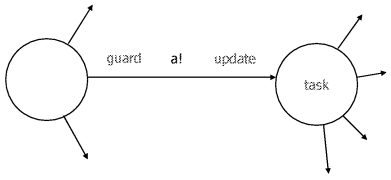
- Commercial Tools
  - focus: code generation
    - new features ...
- Academic Tools
  - focus: modeling & verification
    - new engines ...
- **Verification to Synthesis !**

An Overview of TIMES



The INPUT LANGUAGE is very much like "guarded commands"

OBS: guard and update may contain data variables (integer, array)



- guard, update: "synchronous" computation which takes "no time"
  - we adopt the synchronous hypothesis
- task: "asynchronous" computation whch takes time

Tasks = Executable Programs (e.g. C, Java)

- Task Type
  - Synchronous or Asynchronous
  - Non-Periodic (triggered by events) or Periodic
- Task parameters: C, D etc
  - C: Computing time and D: Relative Deadline
  - other parameters for schedulling e.g. priority, period
- Task Interface (variables updated 'atomically')
  - $X_i := F_i(X_1 \ldots X_n)$
- Tasks may have shared variables
  - with automata
  - with other tasks (priority ceiling protocols)
- Tasks with Precedence constraints

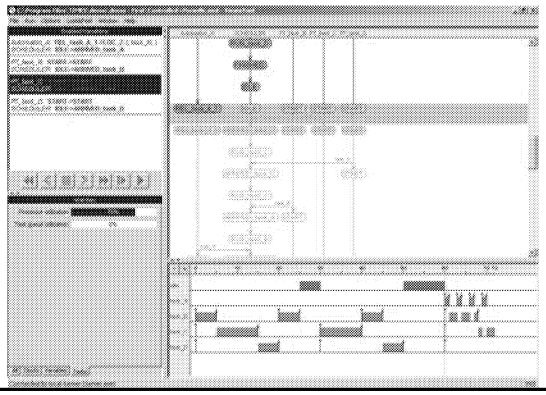
Functionality/Features of TIMES

- GUI
  - Modeling: automata with (a)synchronous tasks
  - editing, task library, visualization etc
- Simulation
  - Symbolic execution as MSC's and Gant Charts
- Verification
  - Safety, bounded liveness properties (all you do with UPPAAL)
  - Schedulability analysis
- Synthesis
  - Verified executable code (guaranteeing timing constraints)
    - Traces(Code) $\subseteq$ Traces(Model)
  - Schedule synthesis (ongoing)

CODE SYNTHESIS in TIMES

- Run Time Systems
  - Event Handler
    - OS interrupt processing system or Polling
  - Task scheduler
    - generated from task parameters
- Application Tasks = threads (or processes)
  - Already there! (written in C)
  - Current version of TIMES support LegoOS !

TIMES



Insup Lee

University of Pennsylvania

Resource-bound process algebras for Schedulability and Performance Analysis of Real-Time and Embedded Systems

Insup Lee[1], Oleg Sokolsky[1], Anna Philippou[2]

[1]SDRL (Systems Design Research Lab)
RTG (Real-Time Systems Group)
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA

[2] Department of Computer Science
University of Cyprus
Nicosia, CY

Outline

- Real-Time and Embedded systems
- Resource-bound computation
- Resource-bound formalisms
  - ACSR (Algebra of communicating shared resources)
  - Schedulability Analysis Problem
  - PACSR (Probabilistic ACSR)
  - Schedulability analysis for soft real-time systems
  - Design framework for embedded systems
  - P^2ACSR (Probabilistic ACSR with power consumption)
  - Scheduling synthesis and parametric schedulability analysis
  - ACSR-VP (ACSR with Value-Passing)
- Conclusions

Real-time, Embedded Systems

- Difficulties
  - Increasing complexity
  - Decentralized
  - Safety critical
  - End-to-end timing constraints
  - Resource constrained
    - Non-functional: power, size, etc.
- Development of reliable and robust embedded software

Properties of embedded systems

- Adherence to safety-critical properties
- Meeting timing constraints
- Satisfaction of resource constraints
- Confinement of resource accesses
- Supporting fault tolerance
- Domain specific requirements
  - Mobility
  - Software configuration

Real-time Behaviors

- Correctness and reliability of real-time systems depends on
  - Functional correctness
  - Temporal correctness
- Factors that affect temporal behavior are
  - Synchronization and communication
  - Resource limitations and availability/failures
  - Scheduling algorithms
  - End-to-end temporal constraints
- An integrated framework to bridge the gap between concurrency theory and real-time scheduling

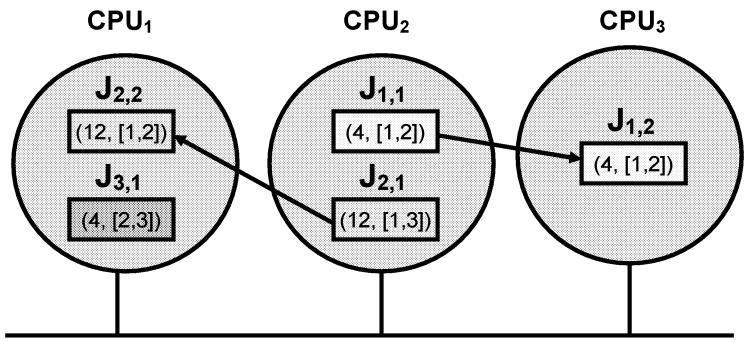
Scheduling Problems

- Priority Assignment Problem
- Schedulability Analysis Problem
- Soft timing/performance analysis (Probabilistic Performance Analysis)
- End-to-end Design Problem
  - Parametric Analysis
  - End-to-end constraints, intermediate timing constraints
  - Execution Synchronization Problem
  - Start-time Assignment Problem with Inter-job Temporal Constraints
- Fault tolerance: dealing with failures, overloads

Scheduling Factors

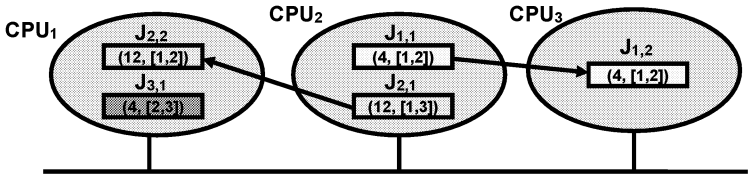
- Static priority vs dynamic priority
  - Cyclic executive, RM (Rate Monotonic), EDF (Earliest Deadline First)
- Priority inversion problem
- Independent tasks vs. dependent tasks
- Single processor vs. multiple processors
- Communication delays
- Uncertainty in execution times
- Resource use tradeoffs
- End-to-end timing requirements

Example: Simple Scheduling Problem

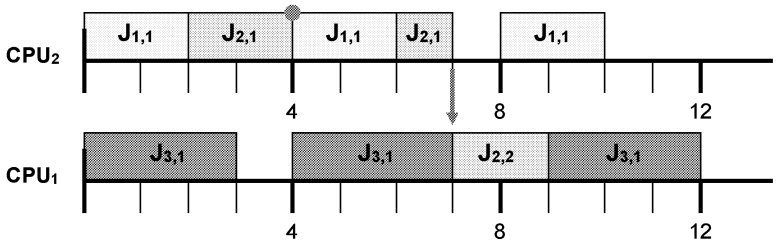


- (period, [e^-, e^+]), where e^- and e^+ are the lower and upper bound of execution time, respectively.
- Goal is to find the priority of each job so that jobs are schedulable
- Considering only worst case leads to scheduling anomaly

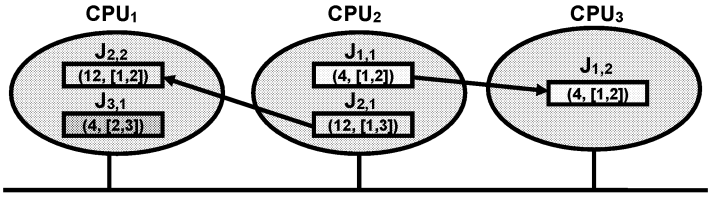
Example (2)



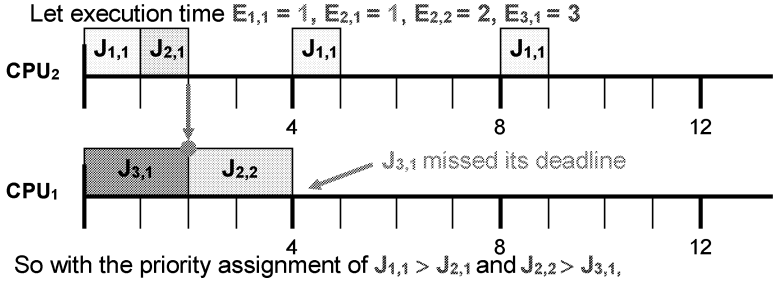
Let $J_{1,1} > J_{2,1}$ and $J_{2,2} > J_{3,1}$
Consider worst case execution time for all jobs, i.e.,
Execution time $E_{1,1} = 2$, $E_{2,1} = 3$, $E_{2,2} = 2$, $E_{3,1} = 3$



Example (3)



With same priorities, $J_{1,1} > J_{2,1}$ and $J_{2,2} > J_{3,1}$
Let execution time $E_{1,1} = 1$, $E_{2,1} = 1$, $E_{2,2} = 2$, $E_{3,1} = 3$



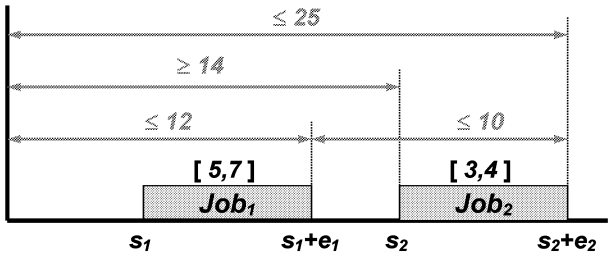
So with the priority assignment of $J_{1,1} > J_{2,1}$ and $J_{2,2} > J_{3,1}$,
jobs cannot be scheduled and scheduling problems are in general NP-hard

End-to-end Design Problem

- Given a task set with end-to-end constraints on inputs and outputs
  - Freshness from input X to output Y (F(Y|X)) constraints: bound time from input X to output Y
  - Correlation between input X1 and X2 (C(Y|X1,X2)) constraints: max time-skew between inputs to output
  - Separation between output Y (u(Y) and l(Y)) constraints: separation between consecutive values on a single output Y
- Derive scheduling for every task
  - Periods, offsets, deadlines
  - priorities
- Meet the end-to-end requirements
- Subject to
  - Resource limitations, e.g., memory, power, weight, bandwidth

Example: Start-time Problem

Start-time Assignment Problem with Inter-job Temporal Constraints



Goal is to statically determine the range of start times for each job so that jobs are schedulable and all *inter-job temporal constraints* are satisfied.

Example: power-aware RT scheduling

- Dynamic Voltage Scaling allows tradeoffs between performance and power consumption
- Problem is how to minimize power consumption while meeting timing constraints.
- Example: three tasks with probabilistic execution time distribution

Task	Worst-case execution time	Period
1	3	8
2	3	10
3	2	14

Our approach and objectives

- Design formalisms for real-time and embedded systems
  - Resource-bound real-time process algebras
  - Executable specifications
  - Logic for specifying properties
- Design analysis techniques
  - Automated verification techniques
  - Parameterized end-to-end schedulability analysis
- Toolset implementation

Resource-bound computation

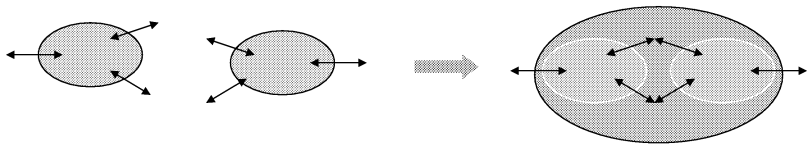
- Computational systems are always constrained in their behaviors
- Resources capture physical constraints
- Resources should be supported as a first-class notion in modeling and analysis
- Resource-bound computation is a general framework of wide applicability

Resources

- Resources capture constraints on executions
- Resources can be
  - Serially reusable:
    - processors, memory, communication channels
  - Consumable
    - power
- Resource capacities
  - Single-capacity resources
  - Multiple-capacity resources
  - Time-sliced, etc.

Process Algebras

- Process algebras are abstract and compositional methodologies for concurrent-system specification and analysis.
- "*Design methodology which systematically allows to build complex systems from smaller ones*" [Milner]



Process Algebras

- A process algebra consists of
  - a set of operators and syntactic rules for constructing processes
  - a semantic mapping which assigns meaning or interpretation to every process
  - a notion of equivalence or partial order between processes
  - a set of algebraic laws that allow syntactic manipulation of processes.
- Ancestors
  - CCS, CSP, ACP,…
  - focus on communication and concurrency

Advantages of Process Algebra

A large system can be broken into simpler subsystems and then proved correct in a modular fashion.

1. A hiding or restriction operator allows one to abstract away unnecessary details.
2. Equality for the process algebra is also a congruence relation; and thus, allows the substitution of one component with another equal component in large systems.

ACSR

ACSR

- ACSR (Algebra of Communicating Shared Resource)
  - A real-time process algebra which features discrete time, resources, and priorities
  - Timeouts, interrupts, and exception handling
  - Two types of actions:
    - Instantaneous events
    - Timed actions

Events

- Events represent non-time consuming activities
  - events are instantaneous: `crash`

  - point-to-point synchronization

Events

- Events
  - have priorities: **(job,10^{10})**

  - have input and output capabilities

or **(e, p_1) ($\bar{e}$, p_2)**

(e ? , p_1) (e! , p_2)

Actions

- Actions represent activities that
  - take time
  - require access to resources
  - each resource usage has priority of access

$$A = \{(r_1, p_1), (r_2, p_2)\}$$

  - each resource can be used at most once
  - resources of action A: $\rho(A)$
  - idling action: $\varnothing$
- Examples:

  {(cpu,2}}, {(cpu$_1$,3),(cpu$_2$,4)},
  {(semaphore,5)}

Syntax for ACSR processes

- Process terms
- Process names

$$C \stackrel{def}{=} P$$

$$
\begin{array}{lcl}
P & ::= & NIL \\
 & | & A : P \\
 & | & (a, n).P \\
 & | & P + P \\
 & | & P \parallel P \\
 & | & P \Delta_t^a (Q, R, S) \\
 & | & [P]_I \\
 & | & P \setminus F \\
 & | & b \rightarrow P \\
 & | & C
\end{array}
$$

Constant and Nil

$$C \stackrel{def}{=} P$$

C is a constant that represents the process algebra expression P

P = NIL

P does nothing

Prefix Operators

P = A:Q

P performs timed action A and then behaves as Q

P = (a,n).Q

P performs event (a,n) and then behaves as Q

EXAMPLE

Operator $\stackrel{def}{=}$ (ring,1).(pickup,1).Talk

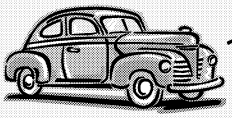
Talk = {(phone,2)} : (hangup,1).Operator

Choice

P = Q+R

P can choose nondeterministically to behave like Q or R

EXAMPLE



$$CAR \stackrel{def}{=} (goleft,1).CAR' + (goright,1).CAR''$$

Parallel Composition

$P = Q \parallel R$

P is composed by Q and R that may synchronize on events and must synchronize on timed actions

EXAMPLE

$$Operator \stackrel{def}{=} (ring?,1).\{(phone,2)\} : (hangup?,1).Operator$$

$$Caller \stackrel{def}{=} (ring!,2).\{(phone',3)\} : (hangup!,1).Caller$$

$$Converse \stackrel{def}{=} Operator \parallel Caller$$

Scope

$$P \stackrel{def}{=} Q \ \Delta_t^a (R,S,T)$$

Q may execute for at most t time units. If message a is produced, control is delegated to R, else control is delegated to S. At any time T may interrupt.

EXAMPLE

$$Runner \stackrel{def}{=} Run \ \Delta_{10}^{finish} (GoForCoffee, GoToWork, BeepedToWork)$$

$$Run \stackrel{def}{=} \{(run,1)\} : Run + finish!.NIL$$

Hiding/Restriction

$P = [Q]_I$

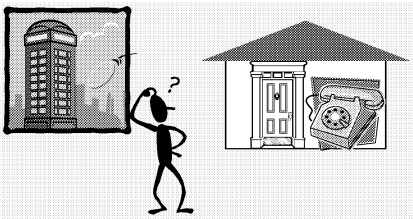
P behaves just as Q but resources in I are no longer visible to the environment

$P = Q\backslash F$

P behaves just as Q but labels in F are no longer visible to the environment

EXAMPLE

$\text{Caller} \parallel \text{PayPhone} \parallel [\text{Home}]_{\text{phone}}$



ACSR semantics

- Gives an unambiguous meaning to language expressions.
- Semantics is operational, given by a set of semantic rules.

ACSR specification → Semantic rules → Labeled transition system

- Example of a labeled transition system:

$$P_0 \xrightarrow{\varnothing} P_1 \xrightarrow{NC} P_2 \xrightarrow{\{gate,\, train\}} P_3 \xrightarrow{\{gate,\, train\}} P_4 \xrightarrow{IC} \ldots$$

ACSR semantics

- Two-level semantics:
  - A collection of inference rules gives the unprioritized transition relation

$$P \xrightarrow{\alpha} P'$$

  - A preemption relation on actions and events disables some of the transitions, giving a prioritized transition relation

$$P \xrightarrow{\alpha}_{\pi} P'$$

Unprioritized transition relation

- Prefix operators

ActT $\dfrac{-}{A : P \xrightarrow{A} P}$ **ActI** $\dfrac{-}{(a,p) : P \xrightarrow{(a,p)} P}$

- Choice

ChoiceL $\dfrac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}$

- Parallel

ParIL $\dfrac{P \xrightarrow{(a,p)} P'}{P \,\|\, Q \xrightarrow{(a,p)} P' \,\|\, Q}$

Unprioritized transition relation (II)

- Resource-constrained execution

$$\textbf{ParT} \quad \frac{P \xrightarrow{A_1} P' \quad Q \xrightarrow{A_2} Q'}{P \,||\, Q \xrightarrow{A_1 \cup A_2} P' \,||\, Q'} \quad \rho(A_1) \cap \rho(A_2) = \varnothing$$

- Priority-based communication

$$\textbf{ParCom} \quad \frac{P \xrightarrow{(a?,p_1)} P' \quad Q \xrightarrow{(a!,p_2)} Q'}{P \,||\, Q \xrightarrow{(\tau,p_1+p_2)} P' \,||\, Q'}$$

- Resource closure

$$\textbf{CloseT} \quad \frac{P \xrightarrow{A_1} P'}{[P]_I \xrightarrow{A_1 \cup A_2} [P']_I} \quad A_2 = \{(r,0) \mid r \in I - A_1\}$$

Examples

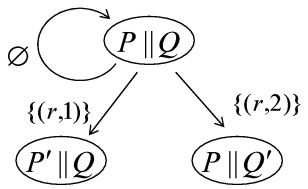
- Resource conflict

$P = \{(r,1)\} : P' \qquad Q = \{(r,2)\} : Q' \qquad P \,||\, Q \sim NIL$

- Processes must provide for preemption

$P = \{(r,1)\} : P' + \varnothing : P \qquad Q = \{(r,2)\} : Q' + \varnothing : Q$

- Unprioritized transitions:



Unprioritized transition relation (III)

ScopeCT $$\frac{P \xrightarrow{A} P'}{P\Delta_t^a(Q,R,S) \xrightarrow{A} P'\Delta_{t-1}^a(Q,R,S)} \quad (t>0)$$

ScopeCI $$\frac{P \xrightarrow{e} P'}{P\Delta_t^a(Q,R,S) \xrightarrow{e} P'\Delta_t^a(Q,R,S)} \quad (l(e) \neq a, t>0)$$

ScopeE $$\frac{P \xrightarrow{(a,n)} P'}{P\Delta_t^a(Q,R,S) \xrightarrow{(\tau,n)} Q} \quad (t>0)$$

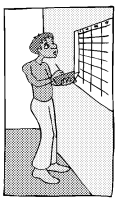
ScopeT $$\frac{R \xrightarrow{\alpha} R'}{P\Delta_t^a(Q,R,S) \xrightarrow{\alpha} R'} \quad (t=0)$$

ScopeI $$\frac{S \xrightarrow{\alpha} S'}{P\Delta_t^a(Q,R,S) \xrightarrow{\alpha} S'} \quad (t>0)$$

Example

- A Scheduler

$$Sched = \phi : Sched$$
$$+ (tc,1).\phi^\infty \Delta_{t_{\max}}^y (NIL, \overline{kill}.Sched, rc.Sched)$$



$$Sched \xrightarrow{(tc,1)} \phi^\infty : \Delta_{t_{\max}}^y(\ldots) \xrightarrow{\phi} \phi^\infty : \Delta_{t_{\max}-1}^y(\ldots) \xrightarrow{\phi} \ldots \xrightarrow{\phi} \phi^\infty : \Delta_0^y(\ldots)$$

$$Sched \xrightarrow{\varnothing}$$

$$\phi^\infty : \Delta_{t_{\max}}^y(\ldots) \xrightarrow{rc} Sched, \quad \phi^\infty : \Delta_{t_{\max}-1}^y(\ldots) \xrightarrow{rc} Sched, \quad \phi^\infty : \Delta_0^y(\ldots) \xrightarrow{\overline{kill}} Sched$$

Preemption relation

- To take priorities into account in the semantics we define the relation α *is preempted by* β : $\alpha \prec \beta$
- An action α preempts action β iff
  - no lower priorities: $\forall r \in \rho(\alpha), \pi_r(\alpha) \leq \pi_r(\beta)$
  - some higher priorities: $\exists r \in \rho(\beta), \pi_r(\alpha) < \pi_r(\beta)$
  - it contains fewer resources $\rho(\beta) \subseteq \rho(\alpha)$

  e.g. $\{(r_1,3),(r_2,5)\} \prec \{(r_1,7),(r_2,5)\}$
- An event preempts another event iff
  - same label, higher priority e.g. $(a!,1) \prec (a!,3)$
- An event preempts an action iff
  - τ with non-zero priority preempts all actions e.g. $\{(r,4)\} \prec (\tau,1)$

Prioritized transition relation

- We define

$$P \xrightarrow{\alpha}_{\pi} P'$$

  when
  - there is an unprioritized transition

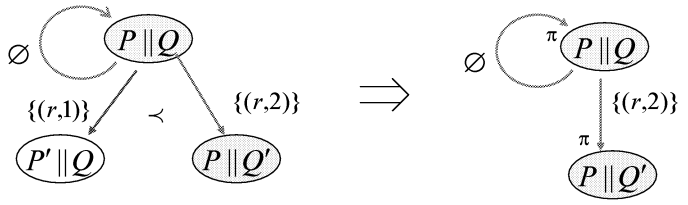
$$P \xrightarrow{\alpha} P'$$

  - there is no $P \xrightarrow{\beta} P''$ such that $\alpha \prec \beta$
- Compositional

Example

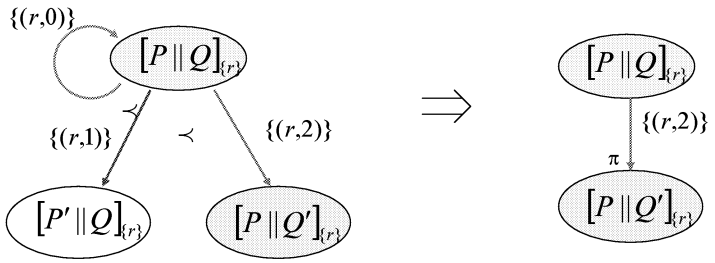
- Unprioritized and prioritized transitions:

$$P = \{(r,1)\} : P' + \varnothing : P \qquad Q = \{(r,2)\} : Q' + \varnothing : Q$$



Example (cont.)

- Resource closure enforces progress



Compositionality of preemption relation

- Given

$$P_1 = (a,2).S_1 + (b,1).S_2$$
$$P_2 = (a,2).S_1$$
$$Q_1 = (\bar{a},3).T_1 + (\bar{b},5).T_2$$
$$Q_2 = (\bar{a},3).T_1 + (\bar{b},2).T_2$$
$$R_1 = (a,2).S_1 + (a,1).S_2$$
$$R_2 = (a,2).S_1$$

- Given P_1 and P_2, can they be treated as equivalent? That is, for all Q, $P_1 \mid\mid Q = P_2 \mid\mid Q$?

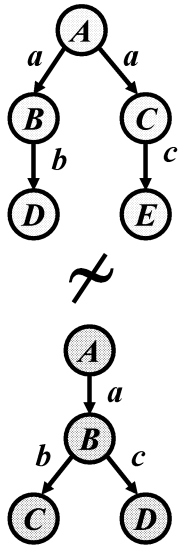
- How about R_1 and R_2?

Bisimulation

- Observational equivalence is based on the idea that two equivalent systems exhibit the same behavior at their interfaces with the environment.

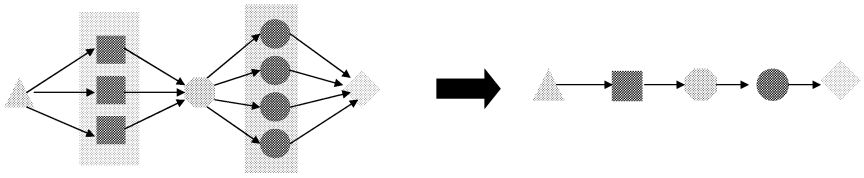
- This requirement was captured formally through the notion of bisimulation, a binary relation on the states of systems.

- Two states are bisimilar if for each single computational step of the one there exists an appropriate matching (multiple) step of the other, leading to bisimilar states.



Prioritized strong equivalence

- An equivalence relation is congruence when it is preserved by all the operators of the language.
- This implies that replacement of equivalent components in any complex system leads to equivalent behavior.



- Strong bisimulation $\sim_\pi$ over $P \xrightarrow{\alpha}_\pi P'$ is a congruence relation with respect to the ACSR operators.

Equational Laws

- Equational laws are a set of axioms on the syntactic level of the language that characterize the equivalence relation.
- They may be used for manipulating complex systems at the level of their syntactic (ACSR) description.
- There is a set of laws that is complete for finite state ACSR processes:

$$P + NIL = P \qquad P + P = P$$

$$P + Q = Q + P \qquad (P \| Q) \| R = P \| (Q \| R)$$

...

Equational Laws

- ACSR-specific laws for scope and resource closure:

$$
\begin{array}{rcll}
A{:}P\Delta_t^a\,(Q,R,S) & = & A{:}\big(P\Delta_{t-1}^a\,(Q,R,S)\big)+S & \text{if } t>0 \\
e.P\Delta_t^a\,(Q,R,S) & = & e.\big(P\Delta_t^a\,(Q,R,S)\big)+S & \text{if } t>0\wedge\overline{l(e)}\neq a \\
e.P\Delta_t^a\,(Q,R,S) & = & (\tau,\pi(e)).Q+S & \text{if } t>0\wedge\overline{l(e)}=a \\
P\Delta_0^a\,(Q,R,S) & = & R & \\
[A_1{:}P]_I & = & (A_1\cup A_2){:}P & A_2=\{(r,0)\,|\,r\in I-\rho(A_1)\} \\
[e.P]_I & = & e.[P]_I &
\end{array}
$$

Laws (1)

Choice(1)	$P+\mathrm{NIL}=P$
Choice(2)	$P+P=P$
Choice(3)	$P+Q=Q+P$
Choice(4)	$(P+Q)+R=P+(Q+R)$
Choice(5)	$\alpha P+\beta Q=\beta Q \quad \text{if } \alpha\prec\beta$
Par(1)	$P\,\|\,Q=Q\,\|\,P$
Par(2)	$(P\,\|\,Q)\,\|\,R=P\,\|\,(Q\,\|\,R)$
Par(3)	$(\sum_{i\in I}A_i:P_i+\sum_{j\in J}e_j.Q_j)\,\|\,(\sum_{k\in K}B_k:R_k+\sum_{l\in L}f_l:S_l)$

$$
=\left[\begin{array}{l}
\sum_{\substack{i\in I,k\in K,\\ \rho(A_i)\cap\rho(B_k)=\emptyset}}(A_i:B_k):(P_i\,\|\,R_k) \\
+\sum_{j\in J}e_j\cdot(Q_j\,\|\,(\sum_{k\in K}B_k:R_k+\sum_{l\in L}f_l\cdot S_l)) \\
+\sum_{l\in L}f_l\cdot((\sum_{i\in I}A_i:P_i+\sum_{j\in J}e_j\cdot Q_j)\,\|\,S_l) \\
\sum_{\substack{j\in J\,l\in L,\\ \overline{l(e_j)=l(f_l)}}}(\tau,\pi(e_j)+\pi(f_l)).(Q_j\,\|\,S_l)
\end{array}\right.
$$

Laws (2)

Scope(1)	$A : P\Delta_t^b(Q,R,S) = A : (P\Delta_{t-1}^b(Q,R,S)) + S \quad \text{if } t > 0$
Scope(2)	$e.P\Delta_t^b(Q,R,S) = e.(P\Delta_{t-1}^b(Q,R,S)) + S \quad \text{if } t > 0 \wedge \overline{l(e)} \neq b$
Scope(3)	$e.P\Delta_t^b(Q,R,S) = (\tau,\pi(e)).Q + S \quad \text{if } t > 0 \wedge \overline{l(e)} = b$
Scope(4)	$P\Delta_0^b(Q,R,S) = R$
Scope(5)	$(P_1 + P_2)\Delta_t^b(Q,R,S) = P_1\Delta_t^b(Q,R,S) + P_2\Delta_t^b(Q,R,S)$
Scope(6)	$\text{NIL}\Delta_t^b(Q,R,S) = S \quad \text{if } t > 0$
Res(1)	$\text{NIL} \setminus F = \text{NIL}$
Res(2)	$(P+Q) \setminus F = (P \setminus F) + (Q \setminus F)$
Res(3)	$(A : P) \setminus F = A : (P \setminus F)$
Res(4)	$((a,n).P) \setminus F = (a,n).(P \setminus F) \quad \text{if } a,\overline{a} \notin F$
Res(5)	$((a,n).P) \setminus F = \text{NIL} \quad \text{if } a,\overline{a} \in F$
Res(6)	$P \setminus E \setminus F = P \setminus E \cup F$
Res(7)	$P \setminus \emptyset = P$

Laws (3)

Close(1)	$[\text{NIL}]_I = \text{NIL}$
Close(2)	$[P+Q]_I = [P]_I + [Q]_I$
Close(3)	$[A_1 : P]_I = (A_1 \cup A_2) : [P]_I \quad \text{where } A_2 = \{(r,0) \mid r \in I - \rho(A_I)\}$
Close(4)	$[e.P]_I = e.[P]_I$
Close(5)	$[[P]_I]_J = [P]_{I \cup J}$
Close(6)	$[P]_\emptyset = P$
Close(7)	$[P \setminus E]_I = [P]_I \setminus E$
Rec(1)	$rec\ X.P = P[rec\ X.P / X]$
Rec(2)	If $P = Q[P/X]$ and X is guarded in Q then $P = rec\ X.Q$
Rec(3)	$rec\ X.(P + \sum_{i \in J}[X \setminus E_i]_{U_i}) = rec\ X.(\sum_{J \subseteq I}[P \setminus E_J]_{U_J})$
	where $E_J = \bigcup_{i \in J} E_i, U_J = \bigcup_{i \in J} U_i$, I is finite and X is guarded in P

Soundness of the laws

- **Theorem:**

  if $P=Q$ then $P \sim_{\pi} Q$

- Proof approach:
  - Construct the set of prioritized derivations for each P
  - Prove that if $P=Q$, then the sets of derivations are the same

Completeness of the laws

- **Theorem:**

  if P and Q are finite-state processes and $P \sim_{\pi} Q$ then P=Q

Schedulability Analysis

Schedulability Analysis

- Can all real-time tasks meet their deadlines?
- Factors include
  - Delay caused by synchronization between tasks
  - Delay caused by precedence between tasks
  - Delay caused by resource constraints
  - Scheduling disciplines and synchronization protocols

Outline

- ACSR-VP: ACSR with value-passing and dynamic priorities
- Specifying real-time systems using ACSR-VP
  - Specifying task models
  - Specifying scheduling disciplines
- Analyzing real-time systems using bisimulation
  - Specification correctness
  - Schedulability analysis
- Schedulability analysis using VERSA (ACSR Toolkit)

ACSR (Algebra of Communicating Shared Resources)

- A timed process algebra based on CCS with notions of time, resources and priorities
- Discrete time and dense time
- Static priorities
- Actions: Instantaneous Events + Timed Actions
  - Timed action: a set of (resource, priority) pairs $\{(cpu, 4),(data, 3)\}$, $\{(cpu_1, 2),(cpu_2, 3)\}$, $\emptyset$
  - Instantaneous event: (event, priority) pair $(signal, 2)$, $(chan, 2)$ $(\tau, 3)$
- Real-time operators for timeout, interrupt, exception
- Graphical specification language (GCSR)
- Toolkit (VERSA)
- No value passing communication, no variables for priorities

ACSR-VP (ACSR with Value Passing)

- Extends ACSR with variables and value passing communications
- Values can be specifies using expressions
  - Timed Actions:
    {(*cpu*, *x*), (*data*, *y* + 1)}
  - Instantaneous events:
    (*signal*!8, *x*) – output
    (*chan*?y, 2) – input
- Dynamic priorities
- Exchange priority information without global variables

ACSR-VP Syntax

P	$::=$	NIL	process that does nothing
	$\mid$	$A : P$	timed action prefix
	$\mid$	$e.P$	instantaneous event prefix
	$\mid$	$be \rightarrow P$	conditional process (be : boolean expression)
	$\mid$	$P_1 + P_2$	choice
	$\mid$	$P_1 \parallel P_2$	parallel composition
	$\mid$	$[P]_I$	resource close
	$\mid$	$P \setminus F$	event restriction
	$\mid$	$P \setminus\setminus I$	resource hiding
	$\mid$	$C(\underline{x})$	process name defined to be a process $C(\underline{x}) = P$

ACSR-VP Example

Preemptable and Non-preemptable Jobs

- Both jobs execute *c* time units on *cpu* with priority π
- Non-preemptable job: once it acquires *cpu*, it executes to completion

$$\text{Job}_1 \stackrel{def}{=} \varnothing : \text{Job}_1 + \text{Exec}_1(0)$$
$$\text{Exec}_1(s) \stackrel{def}{=} (s < c) \rightarrow \{(cpu, \pi)\} : \text{Exec}_1(s+1)$$

- Preemptable job: its execution can be preempted by actions on *cpu* of other jobs with higher priorities

$$\text{Job}_2 \stackrel{def}{=} \varnothing : \text{Job}_2 + \text{Exec}_2(0)$$
$$\text{Exec}_2(s) \stackrel{def}{=} (s < c) \rightarrow \{(cpu, \pi)\} : \text{Exec}_2(s+1) + \varnothing : \text{Exec}_2(s)$$

Unprioritized Operational Semantics

Act $\quad \mathrm{A} : \mathrm{P} \xrightarrow{\mathrm{A}} P$

ActI1 $\quad (l?\langle \underline{x} \rangle, \upsilon e).P \xrightarrow{(l?\langle \underline{n} \rangle, [\upsilon e])} P[\underline{n}/\underline{x}]$

ActI2 $\quad (l!\langle \underline{\upsilon e_2} \rangle, \upsilon e_1).P \xrightarrow{(l!\langle [\underline{\upsilon e_2}] \rangle, [\upsilon e_1])} P$

ActI3 $\quad (\tau, \upsilon e).P \xrightarrow{(\tau, [\upsilon e])} P$

ParT $\quad \dfrac{P \xrightarrow{A_1} P', Q \xrightarrow{A_2} Q'}{P \| Q \xrightarrow{A_1 \cup A_2} P' \| Q'} (\rho(A_1) \cap \rho(A_2) = \varnothing$

ParC2 $\quad \dfrac{P \xrightarrow{(l!\langle \underline{k} \rangle, m)} P', Q \xrightarrow{(l?\langle \underline{k} \rangle, n)} Q'}{P \| Q \xrightarrow{(\tau, m+n)} P' \| Q'}$

Unprioritized Operational Semantics

CloseT $$\frac{P \xrightarrow{A_1} P'}{[P]_I \xrightarrow{A_1 \cup A_2} [P']_I} (A_2 = \{(r,0) \mid r \in I - \rho(A_1)\})$$

CloseI $$\frac{P \xrightarrow{e} P'}{[P]_I \xrightarrow{e} [P']_I}$$

HideT $$\frac{P \xrightarrow{A} P'}{P \backslash\backslash I \xrightarrow{A'} P \backslash\backslash I} (\{(r,p) \in A \mid r \notin I\})$$

HideI $$\frac{P \xrightarrow{e} P'}{P \backslash\backslash I \xrightarrow{e} P \backslash\backslash I}$$

Preemption

A preemption relation is defined for two any actions α and β, denoted $\alpha \prec \beta$, read β preempts α.

Examples:

- $\{(r_1,2),(r_2,5)\} \prec \{(r_1,7),(r_2,5)\}$
- $\{(r_1,2),(r_2,5)\} \not\prec \{(r_1,7),(r_2,3)\}$
- $\{(r_1,2),(r_2,0)\} \prec \{(r_1,7)\}$
- $\{(r_1,2),(r_2,1)\} \not\prec \{(r_1,7)\}$
- $(\mathrm{a},2) \prec (a,5)$
- $(\mathrm{a},1) \not\prec (b,2)$
- $(\tau,1) \prec (\tau,2)$
- $\{(r_1,2),(r_2,5)\} \prec (\tau,2)$

Prioritized Operational Semantics

The operational semantics of ACSR-VP, the *prioritized transition relation* $\xrightarrow{\alpha}_{\pi}$ is defined as follows:

$$P \xrightarrow{\alpha}_{\pi} P' \text{ iff } (1)\ P \xrightarrow{\alpha} P'$$
$$(2) \text{ there is no } P \xrightarrow{\beta} P'' \text{ such that } \alpha \prec \beta$$

Example: $P \overset{def}{=} \{(cpu,2)\} : P_1 + \{(cpu,3)\} : P_2$

- Unprioritized transition: $\begin{cases} P \xrightarrow{\{(cpu,2)\}} P_1 \\ P \xrightarrow{\{(cpu,3)\}} P_2 \end{cases}$
- Prioritized transition: $P \xrightarrow{\{(cpu,3)\}}_{\pi} P_2$

Modeling a Real-Time System

- A real-time system consists of a set of tasks running in parallel under a specific scheduling discipline
- A task is a process composed of a sequence of jobs executed serially
- A task can be
  - Independent or dependent
  - Preemptable or non-preemptable
  - Periodic or aperiodic
- Possible timing constraints of a task are:

b	Starting time
c, d	Execution time and deadline
p	Period for periodic task
p_1, p_2	Minimum and maximum inter-arrival times for aperiodic task

Specification of a real-Time System

A real-time system is specified by the process **RTS**:

$$\mathrm{RTS} \stackrel{def}{=} \left[\mathrm{T}_1 \parallel \mathrm{T}_2 \parallel \cdots \parallel \mathrm{T}_n \right]_R$$

Tasks are specified by the processes $\mathbf{T}_i$:

$$\mathrm{T}_i \stackrel{def}{=} (\mathrm{Job}_i \parallel \mathrm{Activator}_i) \setminus \{start, end\}$$

- Process $\mathbf{Job}_i$: internal characteristics, e.g.,:
  - resource requirements
  - synchronization
- Process $\mathbf{Activator}_i$: external timing attributes, e.g.,
  - periodic or aperiodic
  - period and deadline
- Events *start, end* are synchronization events:
  - *start*: activate jobs
  - *end*: mark deadlines of jobs – deadlock if unsuccessful

Sample Activators

Activator 1. A periodic task with (b, d, p)

$$\begin{aligned} \mathrm{Activator} &\stackrel{def}{=} \varnothing^{\mathrm{b}} : \mathrm{Activator}' \\ \mathrm{Activator}' &\stackrel{def}{=} (start!,1).\varnothing^{\mathrm{d}} : (end!,2). \\ &\quad\ \varnothing^{\mathrm{p-d}} : \mathrm{Activator}' \end{aligned}$$

Activator 2. An aperiodic task with (b, d, p_1, p_2)

$$\begin{aligned} \mathrm{Activator} &\stackrel{def}{=} \varnothing^{\mathrm{b}} : \mathrm{Activator}' \\ \mathrm{Activator}' &\stackrel{def}{=} (start!,1).\varnothing^{\mathrm{d}} : (end!,2). \\ &\quad\ \varnothing^{\mathrm{p_1-d}\ldots\mathrm{p_2}-d} : \mathrm{Activator}' \end{aligned}$$

where

$$\varnothing^{n} \stackrel{def}{=} \varnothing : \cdots : \varnothing \qquad \text{(idling for } n \text{ time units)}$$

$$\varnothing^{m..n} \stackrel{def}{=} \varnothing^{m} + \varnothing^{m+1} + \cdots + \varnothing^{n}$$

Sample Jobs

Job 1

- preemptable, independent jobs
  running on *cpu*
  priority π and execution time c:

$$\begin{aligned}
\text{Job} &\stackrel{def}{=} \varnothing : \text{Job} + (start\,?,1).\text{Exec}(0,0) \\
\text{Exec}(s,t) &\stackrel{def}{=} (s < c) \rightarrow (\{cpu, \pi)\} : \text{Exec}(s+1, t+1) \\
&\qquad + \varnothing : \text{Exec}(s, t+1)) \\
&\quad + (s = c) \rightarrow \text{Wait} \\
\text{Wait} &\stackrel{def}{=} \varnothing : \text{Wait} + (end\,?,1).\text{Job}
\end{aligned}$$

- *s* for accumulated execution time
- *t* for the elapsed time
- **Job** can response to *end* event only when its current execution is finished

Sample Jobs

Job 2

- nonpreemptable, independent jobs
  on multiprocessors $cpu_1, \ldots, cpu_k$
  with priorities $\pi_1, \ldots, \pi_k$ and execution time c:

$$\begin{aligned}
\text{Job} &\stackrel{def}{=} \varnothing : \text{Job} + (start\ ?,1).\text{Exec} \\
\text{Exec} &\stackrel{def}{=} \sum_{1 \le i \le k} (\{(cpu_i, \pi_i)\}^c : \text{Wait}) \\
\text{Wait} &\stackrel{def}{=} \varnothing : \text{Wait} + (end\ ?,1).\text{Job}
\end{aligned}$$

- A job can be executed on any of the processors
- Once a processor is assigned to a job, the job executes on that processor until completion

Sample Jobs

Job 3

- dependent jobs on processor *cpu* with priority π and execution time c
  a single preemptable critical section of length cs on resource *data* (with priority π') after at c' time units execution:

$$
\begin{array}{lcl}
\text{Job} & \stackrel{def}{=} & \varnothing : \text{Job} + (start\ ?,1).\text{Exec}(0,0) \\
\text{Exec} & \stackrel{def}{=} & (s < c \wedge s \neq c') \rightarrow (\{(\text{cpu}, \pi)\} : \text{Exec}\ (s+1, t+1) \\
 & & \qquad + \varnothing : \text{Exec}\ (s, t+1)) \\
 & & + (s = c') \rightarrow ((p!,0).\text{CS}\ (s,t) \\
 & & \qquad + \varnothing : \text{Exec}\ (s, t+1)) \\
 & & + (s = c) \rightarrow \text{Wait} \\
\text{Wait} & \stackrel{def}{=} & \varnothing : \text{Wait} + (end\ ?,1).\text{Job} \\
\hline
\text{CS}\ (s,t) & \stackrel{def}{=} & (s < c' + cs) \rightarrow (\{(cpu, \pi)\} : \text{CS}\ (s+1, t+1) \\
 & & \qquad + \varnothing : \text{CS}\ (s, t+1)) \\
 & & + (s = c' + cs) \rightarrow (v!,0).\text{Exec}\ (s,t) \\
\hline
\text{P} & \stackrel{def}{=} & (p?,\ 0).\text{V} + \varnothing : \text{P} \\
\text{V} & \stackrel{def}{=} & (v?,\ 0).\text{P} + \varnothing : \text{V}
\end{array}
$$

- P and V operations are modeled by the processes P and V with events (p?,0) and (v?,0)
- When *s* equals c', Exec waits for (p?,0) to enter the critical section CS(s,t)

Scheduling Disciplines

Earliest Deadline First

- Tasks $\quad \text{T}_i \stackrel{def}{=} \text{Job}\,1 + \text{Activator}\,1$
- Priority $\quad \pi_i = \text{d}_{max} - (\text{d}_i - t)$

where $\text{d}_{max} \stackrel{def}{=} (1 + \max\{\text{d}_1, \cdots, \text{d}_n\})$

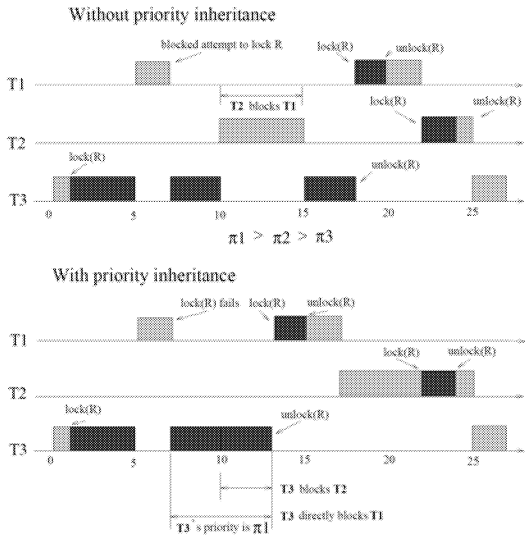
$$
\begin{array}{lcl}
\text{EDFSys} & \stackrel{def}{=} & \left[\text{T}_1 \parallel \text{T}_2 \parallel \cdots \parallel \text{T}_n\right]_{\{cpu\}} \\
\text{T}_i & \stackrel{def}{=} & (\text{Job}_i \parallel \text{Activator}_i) \setminus \{start, end\} \\
\hline
\text{Job}_i & \stackrel{def}{=} & \varnothing : \text{Job}_i + (start\ ?,1).\text{Exec}_i(0,0) \\
\text{Exec}_i(s,t) & \stackrel{def}{=} & (s < c_i) \rightarrow \{(cpu, \text{d}_{max} - (\text{d}_i - t))\} \\
 & & \qquad : \text{Exec}_i(s, t+1) \\
 & & \qquad + \varnothing : \text{Exec}_i(s, t+1) \\
 & & + (s = c) \rightarrow \text{Wait} \\
\text{Wait}_i & \stackrel{def}{=} & \varnothing : \text{Wait}_i + (end\ ?,1).\text{Job}_i \\
\hline
\text{Activator}_i & \stackrel{def}{=} & (start!,1).\varnothing^{\text{d}_i} : (end!,2).\varnothing^{\text{p-d}} : \text{Activator}_i
\end{array}
$$

Other Time-Driven Scheduling Disciplines

Deadline Monotonic	$\pi_i \overset{def}{=} \mathrm{d}_{max} - \mathrm{d}_i$
Shortest Remaining Time First	$\pi_i \overset{def}{=} \mathrm{c}_{max} - (\mathrm{c}_i - s)$
Least Laxity First	$\pi_i \overset{def}{=} \mathrm{d}_{max} - (\mathrm{d}_i - t) - (\mathrm{c}_i - s)$

where $\mathrm{c}_{max} \overset{def}{=} (1 + \max\{\mathrm{c}_1, \cdots, \mathrm{c}_n\})$

The Priority Inversion Problem



Task parameters

Resources:	*cpu*		processor		
Constants:	ready time	:	$r_1 = 5$	$r_2 = 10$	$r_3 = 0$
	comp. time	:	$c_1 = 6$	$c_2 = 8$	$c_3 = 13$
	deadline	:	$d_1 = 30$	$d_2 = 30$	$d_3 = 30$
	start time of CS	:	$cs_1 = 3$	$cs_2 = 5$	$cs_3 = 1$
	length of CS	:	$c'_1 = 2$	$c'_2 = 2$	$c'_3 = 10$
	priority	:	$\pi_1 = 3$	$\pi_2 = 2$	$\pi_3 = 1$
	max priority	:	$\pi_{max} = 4$		

Priority Inheritance Protocol

$$T_i \stackrel{def}{=} \text{Job } 3 + \text{Activator } 1 + \text{Priority - Passing Events}$$

$$\text{PIPSys} \stackrel{def}{=} \left[\left(T_1 \| T_2 \| T_3 \| P\right) \backslash \{req, chan, p, v\}\right]_{\{cpu\}}$$

$$T_i \stackrel{def}{=} \left(\text{Job}_i \| \text{Activator}_i\right) \backslash \{start, end\}$$

$$\text{Job}_i \stackrel{def}{=} \varnothing : \text{Job}_i + (start\ ?,1).\text{Exec}_i(0,0)$$

$$\begin{aligned}\text{Exec}_i(s) \stackrel{def}{=}\ & (s < c_i \wedge s \neq cs_i) \rightarrow (\{cpu, \pi_i)\} : \text{Exec}_i(s+1) \\ & \qquad + \varnothing : \text{Exec}_i(s)) \\ & + (s = cs_i) \rightarrow ((req!\pi_i, \pi_i).\text{Req}_i(s) + \varnothing : \text{Exec}_i(s)) \\ & + (s = c_i) \rightarrow \text{Wait}\end{aligned}$$

$$\text{Wait}_i \stackrel{def}{=} \varnothing : \text{Wait}_i + (end\ ?,1).\text{Job}_i$$

$$\text{Req}_i(s) \stackrel{def}{=} (p!\pi_i, \pi_i).\text{CS}_i(s, \pi_i) + \varnothing : \text{Req}_i(s)$$

$$\begin{aligned}\text{CS}_i(s,\pi) \stackrel{def}{=}\ & (s < c'_i + cs_i) \rightarrow (\{cpu, \pi)\} : \text{CS}_i(s+1, \pi) \\ & \qquad + (chan\ ?new,1).\text{CS}_i(s, new) \\ & \qquad + \varnothing : \text{CS}_i(s, \pi) \\ & + (s = c'_i + cs_i) \rightarrow (v\ ?,1).\text{Exec}_i(s)\end{aligned}$$

$$\text{Activator}_i \stackrel{def}{=} \varnothing^{r_i} : (start\ !,1).\varnothing^{d_i} : (end\ !,2).\varnothing^{\infty}$$

$$\begin{aligned}P \stackrel{def}{=}\ & (p?x,1).\text{V}(x) + (req\ ?\ x, \pi_{max}).(p\ ?\ x,1).\text{V}(x) + \varnothing : P \\ \text{V}(max) \stackrel{def}{=}\ & (v!,1).P + \varnothing : \text{V}(max) \\ & \quad + (req?x,1).\ ((x > max) \rightarrow (chan!\ x,1).\text{V}(x) \\ & \qquad + (x \leq max) \rightarrow \text{V}(max))\end{aligned}$$

Parameters of T_i
- pty_i Priority
- c_i Execution time of a job
- c'_i Time for enetring critical section
- cs_i Execution time in critical section

Traces of tasks

Time	process T_1	process T_2	process T_3	process P
0	{}	{}	*start*?,{(*cpu*,1)}	P
1	{}	{}	*req*!1, *p*!1,{(*cpu*,1)}•	*req*?1, *p*?1, V(1)
2	{}	{}	{(*cpu*,1)} •	V(1)
3	{}	{}	{(*cpu*,1)} •	V(1)
4	{}	{}	{(*cpu*,1)} •	V(1)
5	*start?,{(cpu,3)}*	{}	{}	V(1)
6	{(*cpu*,3)}	{}	{}	V(1)
7	*req*!3,{}	{}	*chan*?3,{(*cpu*,3)} •	*req*?3, *chan*!3, V(3)
8	{}	{}	{(*cpu*,3)} •	V(3)
9	{}	{}	{(*cpu*,3)} •	V(3)
10	{}	*start*?,{}	{(*cpu*,3)} •	V(3)
11	{}	{}	{(*cpu*,3)} •	V(3)
12	{}	{}	{(*cpu*,3)},*v*? •	*v*!, P
13	*p*!3,{(*cpu*,3)} •	{}	{}	*p*?3, V(3)
14	*{(cpu,3)},v?* •	{}	{}	*v*!, P
15	{(*cpu*,3)}	{}	{}	P
16	{(*cpu*,3)}	{}	{}	P
17	{}	{(*cpu*,2)}	{}	P
18	{}	{(*cpu*,2)}	{}	P
19	{}	{(*cpu*,2)}	{}	P
20	{}	{(*cpu*,2)}	{}	P
21	{}	{(*cpu*,2)}	{}	P
22	{}	*req*!2, *p*!2,{(*cpu*,2)}•	{}	*req*?2, *p*?2, V(2)
23	{}	{(*cpu*,2)},*v*? •	{}	*v*!P
24	{}	{(*cpu*,2)}	{}	P
25	{}	{}	{(*cpu*,1)}	P
26	{}	{}	{(*cpu*,1)}	P

(•: in critical section)

Weak Bisimulation

Def. If $t \in D^*$, then $\hat{t} \in (D - \{\tau\})^*$ is the sequence derived by deleting all occurrences of π from t.

Def. If $t = \alpha_1 \ldots \alpha_n \in D^*$, then $E \overset{t}{\Rightarrow} E'$ if
$P(\xrightarrow{(\tau,\_)})^* \xrightarrow{\alpha_1} (\xrightarrow{(\tau,\_)})^* \cdots (\xrightarrow{(\tau,\_)})^* \xrightarrow{\alpha_n} (\xrightarrow{(\tau,\_)})^* P'$,
where "_" in $(\tau, \_)$ represents arbitrary integer.

Def. For a given transition system "$\rightarrow$", any binary relation r is a weak bisimulation if, for $(P,Q) \in r$ and for any action $\alpha \in D$,

1. if $P \xrightarrow{\alpha} P'$, then, for some Q', $Q \overset{\alpha}{\Rightarrow} Q'$ and $(P',Q') \in r$, and
2. if $Q \xrightarrow{\alpha} Q'$, then, for some P', $P \overset{\alpha}{\Rightarrow} P'$ and $(P',Q') \in r$.

Def. $\approx_\pi$ is the largest weak bisimulation over "$\rightarrow_\pi$". It is an equivalence relation (though not a congruence) for ACSR.

Analyzing Real-Time Systems in ACSR-VP

- Two types of analyses
  - Validation
  - Schedulability analysis
- Basic idea
  - Checking weak bisimulation $\approx_\pi$
  - Searching deadlocked states
- Practical Issues
  - Ensure that the **EDFS**ys and **PIPS**ys processes are finite state
  - Translate ACSR-VP processes to ACSR processes and use VERSA, the toolkit for ACSR

Validating the EDFSys Specification

Construct a correctness specification, EDFSpec, that is sequential and easy to inspect

Verify that **EDFSys** $\approx_\pi$ ***EDFSpec***

$$\mathrm{EDFSpec} \stackrel{\mathrm{def}}{=} [\mathrm{S}(0,\cdots,0,0)]_{\{cpu\}}$$

$$\mathrm{S}(s_1,t_1,\cdots,s_n t_n) \stackrel{def}{=} \sum_{1\le i\le n} \begin{cases} (s_i = \mathrm{c}_i \wedge t_i = \mathrm{p}_i) \\ \qquad \rightarrow (\tau,1).\mathrm{S}(\cdots,s_{i-1},t_{i-1},0,0,s_{i+1},t_{i+1},\cdots) \\ + \ (s_i < \mathrm{c}_i \wedge t_i = \mathrm{d}_i) \\ \qquad \rightarrow (\tau,1).\mathrm{NIL} \\ + \ (s_i = \mathrm{c}_i \wedge t_i < \mathrm{p}_i) \\ \qquad \rightarrow \varnothing : \mathrm{S}(\cdots,s_{i-1},t_{i-1}+1,s_i,t_i+1,s_{i+1},t_{i+1}+1,\cdots) \\ + \ (s_i < \mathrm{c}_i \wedge t_i < \mathrm{d}_i) \\ \qquad \rightarrow \{(cpu,\mathrm{d}_{max} - (\mathrm{d}_i - t))\} \\ \qquad\quad : \mathrm{S}(\cdots,s_{i-1},t_{i-1}+1,s_i+1,t_i+1,s_{i+1},t_{i+1}+1,\cdots) \end{cases}$$

Schedulability Analysis

Lemma 1 *If* **EDFSys** *is deadlock free, then it is schedulable.*

Lemma 2 *If*

$$\text{EDFSys} \setminus\setminus \{cpu\} \approx_{\pi} \varnothing^{\infty},$$

then **EDFSys** *is deadlock free.*

Lemma 3 *If* **PIPSys** *is deadlock free, then it is schedulable.*

Lemma 4 *If*

$$\text{PIPSys} \setminus\setminus \{cpu\} \approx_{\pi} \varnothing^{\infty},$$

then **PIPSys** *is deadlock free*

Example 1

- Consider an instance **EDFS**ys$_1$ of **EDFS**ys where:

  Task T_1: $c_1 = 1$, $d_1 = 2$, $p_1 = 3$

  Task T_2: $c_2 = 2$, $d_2 = 3$, $p_2 = 3$
- The following sufficient condition for schedulability from [Liu and Lay 73] is not satisfied:

$$\frac{c_1}{d_1} + \frac{c_2}{d_2} \le 1$$

- The following equation $\text{EDFSys} \setminus\setminus \{cpu\} \approx_{\pi} \varnothing^{\infty},$

  is satisfied, i.e., the task system is *schedulable.*
  More specifically, we have

$$\text{EDFSys1} \xrightarrow{(\tau,2)}_{\pi} \xrightarrow{(\tau,2)}_{\pi} \xrightarrow{\{(\text{cpu},2)\}}_{\pi} \xrightarrow{\{(\text{cpu},3)\}}_{\pi} \xrightarrow{(\tau,3)}_{\pi}$$

$$\xrightarrow{\{(\text{cpu},3)\}}_{\pi} \xrightarrow{(\tau,3)}_{\pi} \text{EDFSys1}$$

Example 2

- Consider another instance **EDFS**ys$_2$ of **EDFS**ys where:

  Task T_1: $c_1 = 2$, $d_1 = 2$, $p_1 = 3$

  Task T_2: $c_2 = 2$, $d_2 = 3$, $p_2 = 3$
- The equivalence

$$\mathrm{EDFSys}_2 \backslash\backslash \{cpu\} \approx_\pi \varnothing^\infty,$$

  is *false* and the task system is therefore *not schedulable.*

- More specifically, we have

$$\mathrm{EDFSys2} \xrightarrow{(\tau,2)}_\pi \xrightarrow{(\tau,2)}_\pi \xrightarrow{\{(\mathrm{cpu},2)\}}_\pi \xrightarrow{\{(\mathrm{cpu},2)\}}_\pi \xrightarrow{(\tau,3)}_\pi \xrightarrow{\{(\mathrm{cpu},3)\}}_\pi \mathrm{NIL}$$

Summary

- The ACSR paradigm:
  - Formalism for modular specification of real-time systems along with scheduling disciplines
  - Formal characterization of the schedulability analysis in process algebra
- Automated schedulability analysis
  - Provide techniques for detecting timing anomalies before an implementation is developed
  - Integrate into a methodology for engineering reliable real-time systems
- Tools:
  - GCSR (Graphical ACSR)
  - XVERSA: VERSA and GCSR

Probabilistic ACSR for soft real-time scheduling analysis

PACSR (Probabilistic ACSR)

- ACSR extension for probabilistic behaviors.
- Objective :
  - formally describe behavioral variations in systems that arise due to failures in physical devices.
- Since failing devices are modeled by resources we associate a failure probability `p(r)` with every resource `r`
  - at any time unit, `r` is down with probability `p(r)` or up with probability `1-p(r)`
  - failures are assumed to be independent

Syntax for PACSR processes

- Similar to ACSR
- Process terms

$$P ::= NIL \quad | \quad A : P \quad | \quad (a,n).P \quad | \quad P + P \quad | \quad P \parallel P$$
$$| \quad P\Delta_t^a(Q,R,S) \quad | \quad [P]_I \quad | \quad P \setminus F \quad | \quad b \rightarrow P \quad | \quad C$$

- Process names

$$C \stackrel{def}{=} P$$

- Distinction: For all resources r we write $\overline{\mathbf{r}}$ for the *failed occurrence of resource* r. Thus, an action can specify access to failed resources.

Resource failures and recoveries

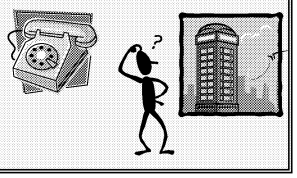
- An action containing resource r cannot be taken when r is failed, i.e.,

  r is failed, $r \in \rho(A) \Rightarrow A : P = NIL$

- Failed resources: $\overline{r}$, $\mathbf{pr}(\overline{r}) = 1 - \mathbf{pr}(r)$
- Recoveries are modeled by using failed resources in actions

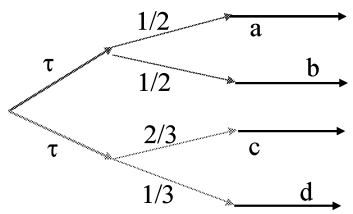
EXAMPLE

$$\{(phone,1)\} : PlaceCall$$
$$+ \{(\overline{phone},1)\} : UsePayPhone$$



PACSR Semantics

- Semantics of a PACSR process is given in terms of *probabilistic transition systems*: some transitions are labeled with probabilities and others with actions/events.
- Labeled Concurrent Markov Chain (LCMC)



PACSR Semantics

- Configurations are pairs of the form (P,W), where
  - *P* is a PACSR process, and
  - *W* is a world capturing the state of resources as follows

$$\forall r, r \in W \Rightarrow \bar{r} \notin W \quad \text{and} \quad \forall r, \bar{r} \in W \Rightarrow r \notin W$$

- A configuration (P,W) is characterized as
  - Probabilistic, if P requires resources whose state is not in W.

    Example: ($\{r_1,1\}$:Q , $\{r_2\}$)
  - Nondeterministic, if all resource information required by P is in W.

    Example: ((a,1):NIL , $\varnothing$)

PACSR semantics (II)

- The semantics is given via a pair of transition relations:
  - Probabilistic transition relation,
$$(P,W) \xrightarrow{pr}_p (P,W')$$
  - Nondeterministic transition relation,
$$(P,W) \xrightarrow{\alpha} (Q,W)$$

- Let imr(P) be resources that can be used in the first step:
$$\{r \mid P \xrightarrow{A} P', r \in \rho(A)\}$$

Operational semantics

- The probabilistic transition relation is as follows:
$$\frac{P \in S_p, Z_1 = \text{imr}(P) - (W \cup \overline{W}), Z_2 \in \mathcal{W}(Z_1)}{(P,W) \xrightarrow{pr(Z_2)}_p (P, W \cup Z_2)}$$

  W(Z) is a set of all possible scenarios of resources; e.g.,
$$\mathcal{W}(\{r_1, \overline{r_2}\}) = \{(\overline{r_1}, \overline{r_2}), (\overline{r_1}, r_2), (r_1, \overline{r_2}), (r_1, r_2)\}$$

- The nondeterministic transition relation is taken from ACSR, with one exception:
$$\textbf{ActT} \quad \frac{-}{(A:P,W) \xrightarrow{A} (P,\varnothing)} \quad \rho(A) \subseteq W$$

Example

- Let $P = \{(r_1,2),(\overline{r_2},3)\} : Q$, pr(r_1) = $\frac{1}{2}$ and pr(r_2) = 1/3.

  Then imr(P) = {r_1,r_2} and W({r_1,r_2})={{r_1,r_2}, {r_1,$\overline{r_2}$}, {$\overline{r_1}$,r_2}, {$\overline{r_1}$,$\overline{r_2}$} }

- Thus by the probabilistic transition relation

$$(P,\phi) \xrightarrow{1/6}_p (P,\{r_1,r_2\}) \qquad (P,\phi) \xrightarrow{1/6}_p (P,\{\overline{r_1},r_2\})$$

$$(P,\phi) \xrightarrow{1/3}_p (P,\{r_1,\overline{r_2}\}) \qquad (P,\phi) \xrightarrow{1/3}_p (P,\{\overline{r_1},\overline{r_2}\})$$

- and by the nondeterministic transition relation

$$(P,\{r_1,r_2\}) \not\longrightarrow$$

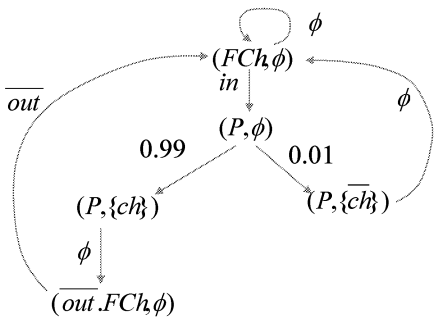
$$(P,\{\overline{r_1},r_2\}) \not\longrightarrow \qquad (P,\{r_1,\overline{r_2}\}) \xrightarrow{\{(r_1,2),(\overline{r_2},3)\}} (Q,\phi)$$

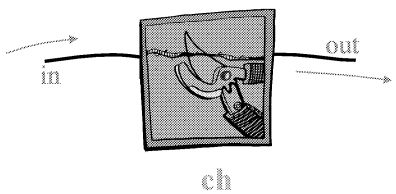
$$(P,\{\overline{r_1},\overline{r_2})\} \not\longrightarrow$$

Example: A faulty channel

$$\mathrm{FCh} = \phi : \mathrm{FCh}$$
$$+\, \mathrm{in}.(\{\mathrm{ch}\} : \mathrm{out}!.\ \mathrm{FCh}$$
$$+\, \{\overline{\mathrm{ch}}\}.\ \mathrm{FCh}) \setminus \{\mathrm{ch}\}$$

where pr(ch) = 0.99





Probabilistic HML with until

- In order to analyze PACSR specifications we may want to check whether a specification satisfies a property written as a logical formula.
- We use a probabilistic HML with an 'until' operator
- The 'until' operator is parameterized with regular expressions over event names.
- Syntax

$$f ::= tt \mid \neg f \mid f \wedge f' \mid f\langle\Phi\rangle_{\omega p} f' \mid f\langle\Phi\rangle^{t}_{\omega p} f'$$

where Φ is a regular expression over actions and $\omega \in \{\leq, \geq\}$

The until operator

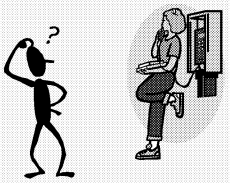
$$P \models f\langle\Phi\rangle^{t}_{\leq q} f'$$

There is some execution with probability ≤ q for which f holds until f' becomes true within time t and observable behavior from Φ

EXAMPLE

$$\text{true}\langle\{\text{talk}, \text{wait}\}^{*}\text{hangup}\rangle^{20}_{0.01}\text{true}$$

≡ the probability that within 20 time units after any number of talk and wait actions action hangup arises is ≤ 0.01



Semantics for *until*

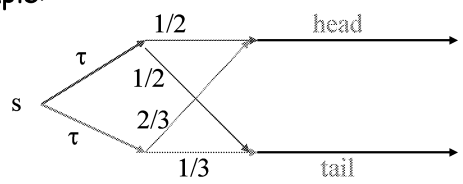
- $s \models f_1 \langle \Phi \rangle_{>\pi} f_2$

  if there exists a scheduler σ such that the set of computations that
  - start at s
  - contain only states (except the last) satisfying f_1
  - have observable content Φ
  - end in a state satisfying f_2

  have probability greater than π

Resolving non-determinism

- Analysis involves computing the probability of reaching a set of desired states (within a time period) via an acceptable set of behaviors.
- Example:



- What is the probability that event head takes place?
- Such probability depends on how the nondeterminism of s is resolved.

Model Checking

- Schedulers are used for resolving non-determinism. These are functions that given a computation ending in a nondeterministic state choose the next transition to take place.
- Given a scheduler σ of a system P, sets of states A and B, and a regular expression Φ, we may compute probabilities
  - $Pr_A(P \rightarrow B, \Phi, t, \sigma)$, the probability of reaching a state in B, passing only via states in A, via paths with observable content in Φ, and within t time units
- So for example:

$$P \models f \langle \Phi \rangle^{t}_{\leq q} f'$$

iff there is scheduler σ such that
$q \geq Pr_A(P \rightarrow B, \Phi, t, \sigma)$
where $A = \{ P' \mid P' \models f \}$,
$B = \{ P' \mid P' \models f' \}$

Model checking *until*

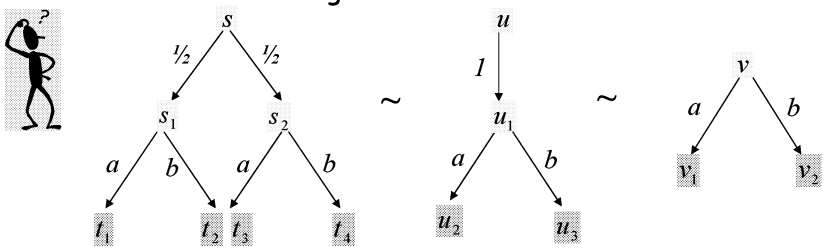
- To check $s \models f_1 \langle \Phi \rangle_{>\pi} f_2$
  - Compute the least solution to the set of equations:

$$X^{s}_{f_1 \langle \Phi \rangle f_2} = \begin{cases} \sum_{s \xrightarrow{\pi} s'} \pi \cdot X^{s'}_{f_1 \langle \Phi \rangle^{\leq p} f_2} & s \in S_p \\ \max_{s \xrightarrow{\alpha} s'} (X^{s'}_{f_1 \langle \Phi \rangle f_2}) & s \in S_n, s \models f_1 \\ 1 & s \in S_n, s \models f_2, \varepsilon \in \Phi, p \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

  - Return true if $X^{s}_{f_1 \langle \Phi \rangle f_2} > \pi$

Equivalence Relations

- New notions of equivalence for the LCMC model taking account both action types and probabilities.
- In particular two LCMCs are strongly bisimilar if
  1. they reach sets of bisimilar states with the same probability, and
  2. for each nondeterministic step of one there exists a step of the other leading to bisimilar states.



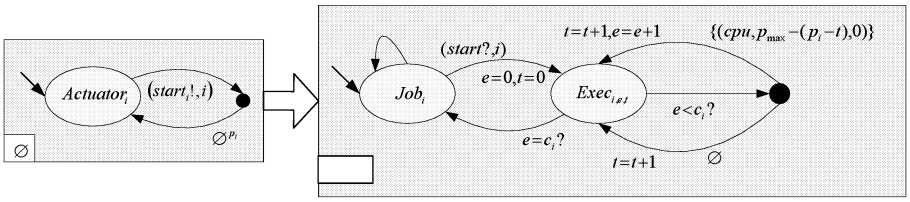
Equivalence Relations

- There is a set of laws that completely axiomatizes strong bisimulation for PACSR processes.
- Other equivalence notions include *weak bisimulation* which relates systems that have the same observable behavior, that is, it ignores τ actions.

Two Examples

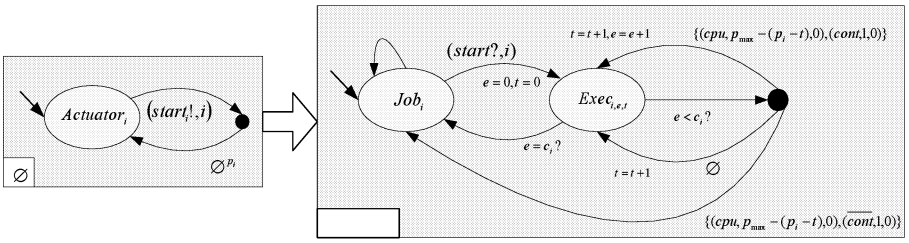
- EDF with probabilistic execution time
- Telecommunication application

EDF task scheduling



- Periodic process *Job*: Period p_i, computation time c_i
  - At each step, total time t increases, active time e increases only if resource *cpu* is available; complete when $e=c_i$
- Resource *cpu*: scheduling
  - Priority of a task dynamically increases closer to the deadline
- Process *Actuator* keeps timing deadlines
  - Every p_i seconds, signal *start* is sent to the task, which can accept it only if it has finished execution

EDF task with probabilistic completion

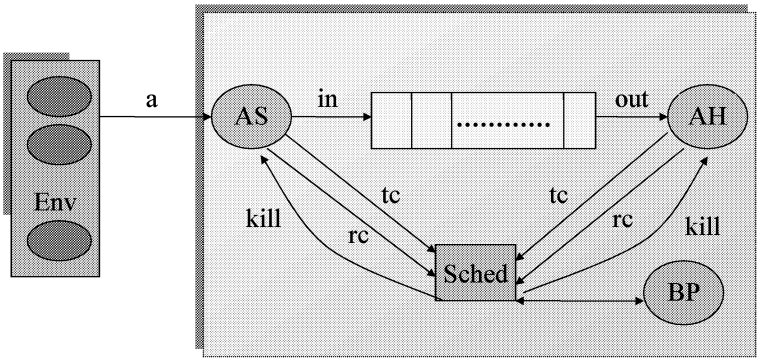


- The task may decide to become inactive after completing a computation step
- Resource *cont* controls probabilistic completion
  - failure means "terminate early"

A Telecommunication Application

- Based on the specification of a switching system considered in AJK97.
- The system consists of a number of concurrent processes with real-time constraints.
- Probabilistic behavior is present in the form of
  - probabilistic arrival of alarms, and
  - uncertain execution times of processes.

Example: A Telecommunication Application



PACSR Specification

- The System

$$Sys = (Env \,||\, B_0 \,||\, \phi : Sched \,||\, AS \,||\, AH \,||\, BP) \setminus F \setminus\setminus I$$

The system in its initial state: *a parallel composition of all the components*

- The environment

$$Env = \Pi_{1 \le i \le n} P_i$$

$$P_i = \{r_i\} : P_i + \{\overline{r_i}\} : (P_i \,||\, Q_i)$$

$$Q_i = \overline{a} : NIL + \phi : Q_i$$

The environment provides probabilistic alarms: *at the failure of any of resources* r_i *an alarm is sent via channel* a

PACSR Specification

- Background Process

$$BP = (\overline{tc},0).BP' \Delta^{h}_{\infty}(NIL, NIL, kill.BP) + \phi : BP$$

$$BP' = (\{r\} : BP' + \{\overline{r}\} : \overline{rc}.BP) \backslash\backslash \{r\}$$

The background process competes for processor time managed by the scheduler. Its duration is geometrically distributed.

- The Scheduler

$$Sched = \phi : Sched$$

$$+ (tc,1).\phi^{\infty} \Delta^{y}_{t_{\max}}(NIL, \overline{kill}.Sched, rc.Sched)$$

PACSR Specification

- The buffer

$$B_0 = in.B_1 + \phi : B_0$$

$$B_i = in.B_{i+1} + \Sigma_{1 \le i \le j} d_j.B_{i-j} + \phi : B_i + \overline{out}_i.B_i$$

$$B_n = in.\overline{overflow}.NIL + \Sigma_{1 \le i \le n} d_j.B_{n-j} + \phi : B_n + \overline{out}_n.B_n$$

- The Alarm Samper and the Alarm Handler

$$AS = AS' \| (\phi^{p} : AS) \qquad AH = \Sigma_i out_i.AH_{n(i)} + \phi : AH$$

$$AS' = (\overline{tc},2).AS'' + \phi : AS' \qquad AH_i = (\overline{tc},2).AH_i^{A} + \phi : AH$$

$$AS'' = a.\overline{in}.AS'' + \phi : \overline{rc}.NIL \qquad AH_i^{A} = \phi^{pt(i)} : \overline{d}_i.\overline{rc}.AH$$

Two configurations

- Consider two versions of the system:
  S_1 with
  - Possibility of 1 alarm per time unit,
  - Buffer size of 3
  - Capability of processing 2 alarms per time unit, and

  S_2 with
  - Possibility of 2 alarms per time unit
  - Buffer size of 6
  - Capability of processing 4 alarms per time unit
- Comparison criterion: What is the probability of overflow in the alarm buffer?

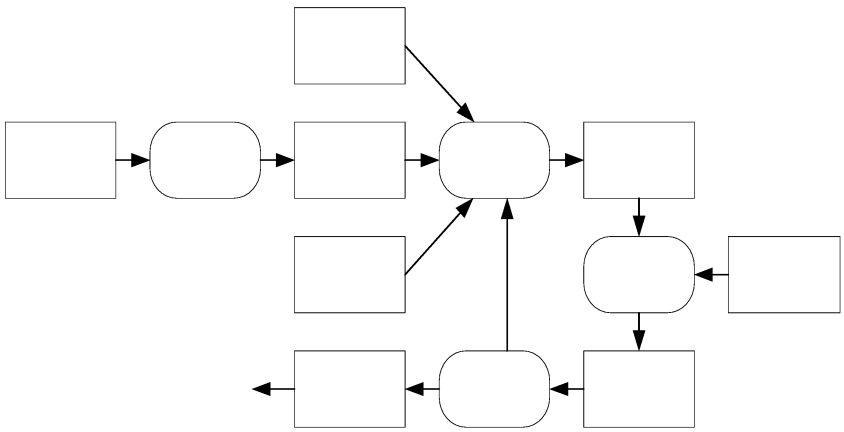
Checking f = $tt\langle\overline{overflow}\rangle^{t}_{\leq q}\, tt$

T(time units)	S_1	S_2
10	2×10^{-6}	3×10^{-10}
20	5×10^{-6}	6×10^{-10}
30	9×10^{-6}	1.0×10^{-9}
40	1.2×10^{-5}	1.3×10^{-9}
50	1.5×10^{-5}	1.6×10^{-9}
60	1.9×10^{-5}	2.1×10^{-9}
70	2.2×10^{-5}	2.4×10^{-9}
80	2.5×10^{-5}	2.8×10^{-9}
90	2.9×10^{-5}	3.1×10^{-9}
100	3.2×10^{-5}	3.5×10^{-9}

The table shows for various values of t, the probability q that makes property f true for each of the systems.

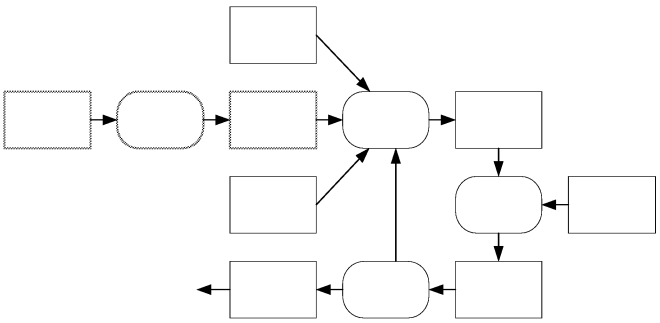
Design of Embedded Systems in a Resource-oriented Framework

Embedded systems design process



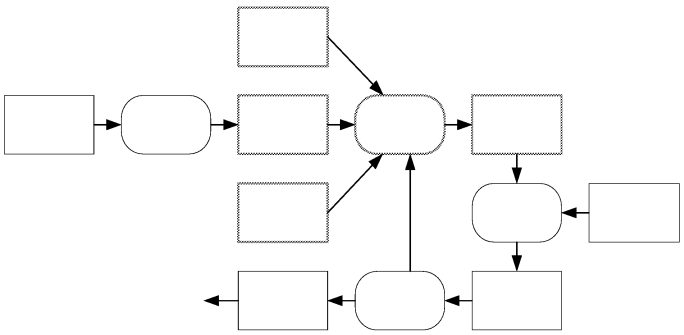
Modeling and code generation

- High-level model captures functionality of the system and assumptions about the environment
- Code generation breaks the functional behavior into a set of tasks



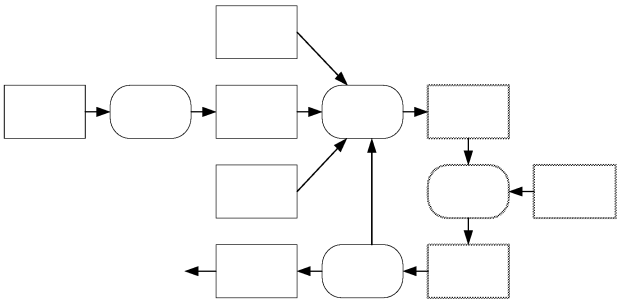
Timing parameter estimation

- Estimate the execution time for task on a given platform
- Assign task periods based on end-to-end timing constraints



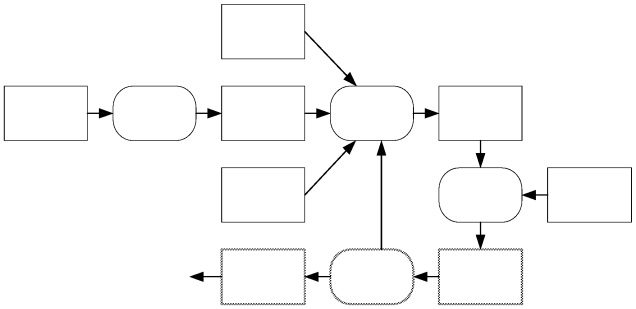
Resource modeling

- Resource is a critical notion in embedded and real-time system design, yet lacks systematic formal treatment
- Key idea: resource attributes capture tradeoffs



Formal schedulability analysis

- Resource conflicts introduce execution delays
- Violations of timing constraints lead to deadlocks in the model behavior
- Discovered by state-space exploration



Modeling and Analysis of Power-Aware Systems

Motivation

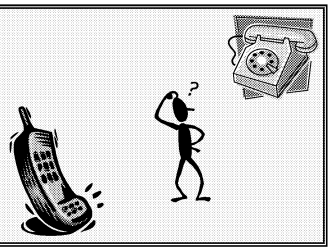
- Features of mobile embedded systems:
  - Resource constraints
    - Limited battery life
  - Uncertainty
    - changing communication delays, failures
- Solution:
  - a unified formal framework for designing and reasoning about power-constrained, timed systems with probabilistic behavior

P^2ACSR – A power-aware extension of PACSR

- A unified framework for modeling and analyzing power-aware real-time systems.
- We associate a further attribute to resource usage, that of power consumption.
- The syntax remains the same, except that actions are tuples of the form (r,p,c), where r is the *resource*, p is the *priority level* and c the *power consumption* of the resource usage.

EXAMPLE

$$\{(phone,1,0)\} : Call_1$$
$$+$$
$$\{(cellphone,1,3)\} : Call_2$$



P^2ACSR

- Semantics is given similarly to PACSR, as a LCMC.
- We can use various techniques to perform various analyses on P^2ACSR models including:
  - Model checking
    We may express temporal logic properties involving power consumption bounds and check that they are satisfied by P^2ACSR processes.
  - Probabilistic bounds on power consumption
    We may compute the probability that power consumption exceeds certain limits.
  - Average power consumption
    We may compute the average power consumption during intervals of interest.

P^2ACSR

- P^2ACSR is an extension of PACSR, a probabilistic real-time process algebra.
- In P^2ACSR:
  - system is a collection of concurrent processes
  - communication among processes is instantaneous
  - access to serially-reusable resources consumes time and power

Resources

- Resources capture constraints on executions
- Features of resources:
  - Serially reusable
    - processors, memory, communication channels
  - Unreliable
    - Fail with a fixed probability in each step
  - Require time and power
    - May allow different levels of power consumption

Actions

- Actions represent computation
  - actions take one unit of time
  - require access to resources
    - each resource r has priority of access p_r
    - each resource r has power use level c_r

      $A = \{(r_1, p_1, c_1), (r_2, p_2, c_2)\}$
    - each resource can be used at most once
  - resources of action A: $\rho(A)$
  - power consumption of action A: $pc(A) = \sum_{r \in \rho(A)} c_r$

Power constraints

- Resource classes $\mathcal{R}_1, \ldots, \mathcal{R}_n$
  - correspond to different power sources
- Attributes of resource class $\mathcal{R}_i$:
  - capacity C_i – maximum amount of power in one step
  - charge $\mathcal{P}_i$ – total amount of power
- Valid actions satisfy capacity constraints:
  - for each $\mathcal{R}_i$, $pc_i(A) = \sum_{r \in \rho(A), r \in \mathcal{R}_i} c_r < C_i$

Processes

- Event and action steps
- Choice $P_1 + P_2$
- Parallel composition $P_1 || P_2$
- Temporal scope, time-outs, exceptions, …

- Structural operational semantic rules build behaviors of complex processes from behaviors of component processes

P²ACSR semantics

- Before steps of a process can be computed, status of relevant resources has to be determined
- Resource status is kept in a world
- Non-deterministic configurations S_n
  - world has complete knowledge of resources
- Probabilistic configurations S_p
  - incomplete knowledge
- Probabilistic steps: $S_p \rightarrow S_n$
  - acquire missing knowledge

Non-deterministic rules

- Action can happen if all resources are available and power constraints are obeyed:

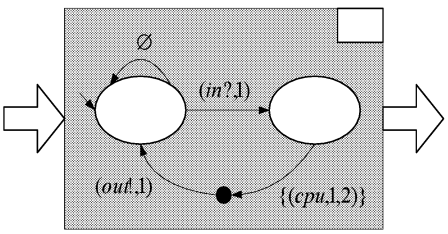
$$(A:P,W) \xrightarrow{A} (P,\varnothing), \quad \rho(A) \subseteq W, valid(A)$$

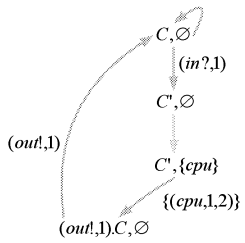
- Parallel processes can proceed if their actions do not conflict and the joint step does not violate constraints

$$\frac{P \xrightarrow{A_1} P' \quad Q \xrightarrow{A_2} Q'}{P \,||\, Q \xrightarrow{A_1 \cup A_2} P' \,||\, Q'} \quad \rho(A_1) \cap \rho(A_2) = \varnothing, valid(A_1 \cup A_2)$$

- Model: Labeled Concurrent Markov Chains

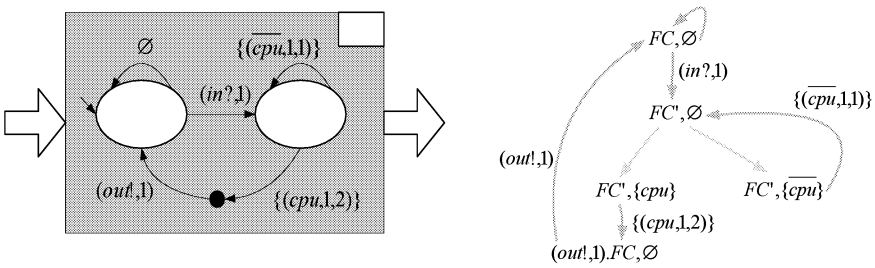
Example





- C is a process that reliably translates messages from *in* to *out* in 1 time unit using 2 units of power per message
- $\pi(cpu) = 1$

Example



- FC (fault-tolerant C) accommodates for cpu failures
  - $\pi(cpu) = 0.99$
- If cpu fails, the message is not delivered, but less power is consumed
- Message is delivered with probability 1
  - What is the expected power consumption per message?

A logic for power constraints

- $\mathcal{L}^{pc}_{PHMLu}$: Power-aware probabilistic HML with *until*
  - Propositional operators $tt, \neg f, f_1 \wedge f_2$
  - *until* operators specify probabilistic bounds on power consumption along a set of paths
    - Basic variant: $f_1 \langle \Phi \rangle^{\leq p}_{>\pi} f_2$
    - With time constraints: $f_1 \langle \Phi \rangle^{\leq p}_{>\pi,t} f_2$
    - With resource class constraints: $f_1 \langle \Phi \rangle^{\leq p,\Re}_{>\pi} f_2$

Semantics for $\mathcal{L}^{pc}_{PHMLu}$: *until*

- $s \models f_1 \langle \Phi \rangle^{\leq p}_{>\pi} f_2$
  if there exists a scheduler σ such that the set of computations that
  - start at s
  - contain only states (except the last) satisfying f_1
  - have observable content Φ
  - consume no more power than p
  - end in a state satisfying f_2

  have probability greater than π

Model checking *until*

- To check $s \models f_1 \langle \Phi \rangle^{\leq p}_{>\pi} f_2$
  - Compute the least solution to the set of equations:

$$X^{s}_{f_1 \langle \Phi \rangle^{\leq p} f_2} = \begin{cases} \sum_{s \xrightarrow{\pi} s'} \pi \cdot X^{s'}_{f_1 \langle \Phi \rangle^{\leq p} f_2} & s \in S_p \\ \max_{s \xrightarrow{\alpha} s'} (X^{s'}_{f_1 \langle \Phi \rangle^{\leq p - pow(\alpha)} f_2}) & s \in S_n, s \models f_1 \\ 1 & s \in S_n, s \models f_2, \varepsilon \in \Phi, p \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

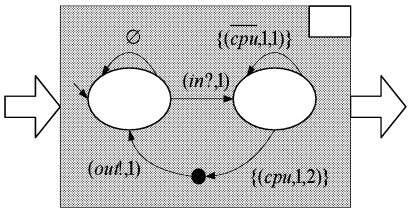
  - Additional annotation $\leq p$ in the variable set
  - Return true if $X^{s}_{f_1 \langle \Phi \rangle^{\leq p} f_2} > \pi$

Example

- Power consumption per message:

$$FC,\varnothing \models tt\left\langle in\cdot\{cpu,\overline{cpu}\}^*\cdot out\right\rangle_{\geq 1}^{>2} tt$$

$$FC,\varnothing \models tt\left\langle in\cdot\{cpu,\overline{cpu}\}^*\cdot out\right\rangle_{>0.999}^{\leq 3} tt$$



Example $FC,\varnothing \models tt\left\langle in\cdot\{cpu,\overline{cpu}\}^*\cdot out\right\rangle_{>0.999}^{\leq 3} tt$

$$X^{FC,\varnothing}_{tt\left\langle in\cdot\{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt} = X^{FC',\varnothing}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt}$$

$$X^{FC',\varnothing}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt} = 0.99\cdot X^{FC',\{cpu\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt} + 0.01\cdot X^{FC',\{\overline{cpu}\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt}$$

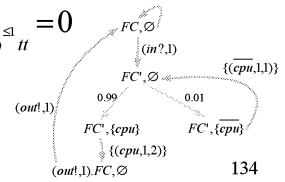
$$X^{FC',\{cpu\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt} = X^{out!.FC,\varnothing}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt} = X^{FC,\varnothing}_{tt\langle \varepsilon\rangle^{\leq 1} tt} = 1$$

$$X^{FC',\{\overline{cpu}\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt} = X^{FC',\varnothing}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 2} tt}$$

$$X^{FC',\varnothing}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 2} tt} = 0.99\cdot X^{FC',\{cpu\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 2} tt} + 0.01\cdot X^{FC',\{\overline{cpu}\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 2} tt}$$

$$X^{FC',\{cpu\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 2} tt} = 1 \quad X^{FC',\{cpu\}}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 2} tt} = X^{FC',\varnothing}_{tt\left\langle \{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 1} tt} = 0$$

$$X^{FC,\varnothing}_{tt\left\langle in\cdot\{cpu,\overline{cpu}\}^*\cdot out\right\rangle^{\leq 3} tt} = 0.9999$$



Power-aware scheduling

- Trade-off: power vs. execution time
- CMOS-based processors can operate at reduced voltage levels
  - Power dissipation is proportional to V^2
  - StrongARM SA2:
    - 600 MHz / 500 mJ or 150 MHz / 160 mJ
- Tasks can take less than worst-case time
  - Adjust frequency dynamically to utilize "time slack"

Dynamic Voltage Scaling

- *Dynamic voltage scaling* is a technique proposed for making energy savings by dynamically altering the power consumed by a processor.
- Lower frequency execution implies longer processing of tasks.
- This may lead to violation of real-time constraints.
- [Pillai and Shin 01] propose extensions to real-time scheduling algorithms to make use of dynamic voltage scaling.

Case study: two kinds of resources

- Power-aware resources:
  - Attributes:
    - Priority (dynamic) – schedulability analysis
    - Power consumption (dynamic) – power calculations
- "abstract" resources:
  - Attributes:
    - Availability (static) – probabilistic completion
    - No power consumption, same priority

Power-Aware Real-Time Scheduling

- Let I be a set of tasks with periods p_i and worst-case execution times c_i, sharing the same CPU.
- In reality tasks often take much less time to execute.
- This probabilistic execution time may be modeled in PACSR as follows:

$$\mathbf{Task}_i = (start?,0)\ .\ \mathbf{Exec}_{i,0,0} + \varnothing : \mathbf{Task}_i \qquad i = \{1,\ldots,I\}$$

$$\begin{aligned}\mathbf{Exec}_{i,e,t} = e < c_i \rightarrow\ (\ &\varnothing : \mathbf{Exec}_{i,e,t+1} \\ &+ \{(cpu,\ dmax-(p_i-t)),(cont,1)\ \} : \mathbf{Exec}_{i,e+1,t+1} \\ &+\{(cpu,dmax-(p_i-t)),(\overline{cont,1})\} : \mathbf{Task}_i\ ) \\ + e = c_i \rightarrow \mathbf{Task}_i\end{aligned}$$

$i = \{1,\ldots,I\}$
$e = \{0,\ldots, c_i\}$
$t = \{0,\ldots, c_i\}$

potential for early termination (geometric distribution)

Power-Aware Real-Time Scheduling

- The algorithm of [Pillai and Shin] takes advantage of the possibility of early termination of a task by then executing the next task at the lowest possible frequency.
- Specifically, on every release or completion of a task it re-computes the sum

$$\alpha = \frac{c_1^{last}}{p_1} + \ldots + \frac{c_n^{last}}{p_n}$$

  where c_i^{last} is the computation time of the last execution of task i or c_i if task i has just been released.
- Based on this value it decides the lowest frequency that is consistent with the current effective utilization.

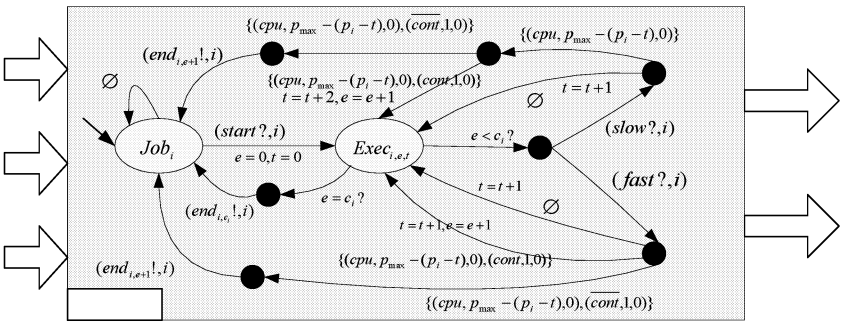
Power-Aware Real-Time Scheduling

- First we extend the model of a task with the ability of executing slower or faster. It responds to messages *fast* and *slow*. In the slow mode a computation step takes twice as long, i.e two time units. It also signals its release when execution commences and its completion time when it completes.

$$\mathbf{Task}_i = (start_i?,0)\ .\ (release_i!, \mathrm{i}).\ \mathbf{Exec}_{i,0,0} + \varnothing : \mathbf{Task}_i \qquad i = \{1,\ldots,I\}$$

$$\begin{aligned}
\mathbf{Exec}_{i,e,t} = e < c_i \rightarrow\ & \\
& ((fast?\ , \mathrm{i})\ (\ \varnothing : \mathbf{Exec}_{i,e,t+1} \\
& \qquad + \{(cpu,\ dmax-(p_i-t)),(cont,1)\} : \mathbf{Exec}_{i,e+1,t+1} \\
& \qquad + \{(cpu,\ dmax-(p_i-t)),\ \overline{(cont,1)}\} : (end_{\mathrm{i,e+1}}!,\mathrm{i}).\ \mathbf{Task}_i\ ) \\
& + (slow?\ , \mathrm{i})\ (\ \varnothing : \mathbf{Exec}_{i,e,t+1} \\
& \qquad + \{(cpu,\ dmax-(p_i-t)),(cont,1)\} : \\
& \qquad (\{(cpu,\ dmax-(p_i-t)),(cont,1)\} : \mathbf{Exec}_{i,e+1,t+2} \\
& \qquad + \{(cpu,\ dmax-(p_i-t)),\ \overline{(cont,1)}\} : (end_{\mathrm{i,e+1}}!,\mathrm{i}).\ \mathbf{Task}_i\ ) \\
& + e = c_i \rightarrow \mathbf{Task}_\mathrm{i}
\end{aligned}$$

Speed-sensitive task



- If operating frequency is *fast*, take one time unit per scheduling step
- If operating frequency is *slow*, take two time units per scheduling step

Power-Aware Real-Time Scheduling

- The DVS algorithm is represented as the P^2ACSR process:

$$DVS = (Scale_{c_1,c_2,c_3} \parallel \mathrm{Proc}_{fast}) \setminus \{f_{up}, f_{down}\}$$

- Scale responds to release and completion signals and triggers the re-computation of α

$$\begin{aligned} Scale_{e_1,e_2,e_3} = & (release_1?,0).SetNew_{c_1,e_2,e_3} \\ & + (release_2?,0).SetNew_{e_1,c_2,e_3} \\ & + (release_3?,0).SetNew_{e_1,e_2,c_3} \\ & + \ldots \\ & + (end_{1,c}?,0).SetNew_{c,e_2,e_3} \\ & + (end_{2,c}?,0).SetNew_{e_1,c,e_3} \\ & + \ldots \end{aligned}$$

Power-Aware Real-Time Scheduling

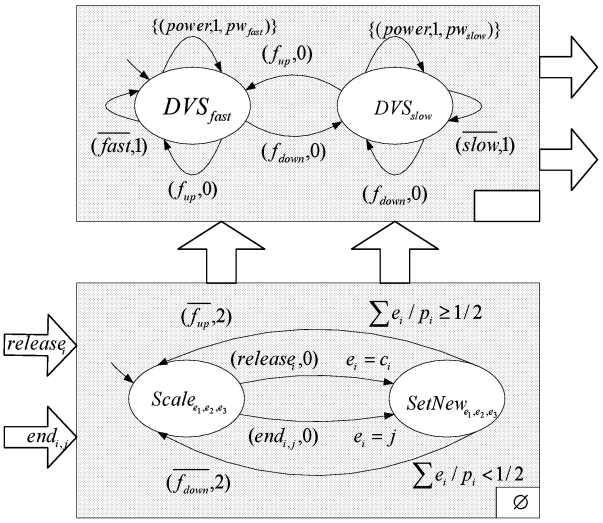
- SetNew decides the lowest frequency to the current effective utilization and sends the appropriate signal

$$\mathbf{SetNew}_{e1,e2,e3} = e_1/p_1 + e_2/p_2 + e_3/p_3 < ½ \rightarrow (f_{down}!,4).\ \mathbf{Scale}_{e1,e2,e3}$$
$$+ e_1/p_1 + e_2/p_2 + e_3/p_3 \geq ½ \rightarrow (f_{up}!,4).\ \mathbf{Scale}_{e1,e2,e3}$$

- DVS_{fast} and DVS_{slow} describe the processor operating in the high and low frequency, respectively

$$\mathbf{DVS}_{fast} = \{(power,1,pw_{fast})\}:\mathbf{DVS}_{fast} + (fast!,1).\mathbf{DVS}_{fast}$$
$$+(f_{down}?,0).\mathbf{DVS}_{slow} + (f_{up}?,0).\mathbf{DVS}_{fast}$$
$$\mathbf{DVS}_{slow} = \{(power,1,pw_{slow})\}:\mathbf{DVS}_{slow} + (slow!,1).\mathbf{DVS}_{slow}$$
$$+(f_{down}?,0).\mathbf{DVS}_{slow} + (f_{up}?,0).\mathbf{DVS}_{fast}$$

Operating frequency manipulation



- Recompute frequency each time a task is released or completed
- Consume pw_{fast} in fast mode and pw_{slow} in slow mode

Analysis of DVS

- We considered the following set of tasks:

Task	Execution time	Period
1	3	8
2	3	10
3	1	14

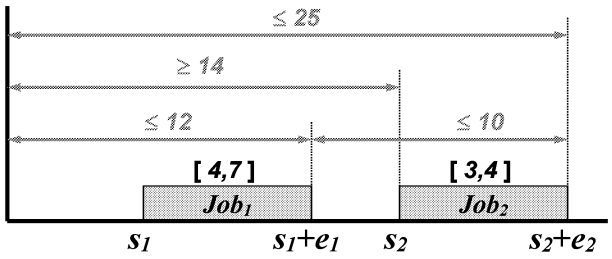
- The algorithm guarantees the task set remains schedulable.
- We computed the expected power consumption for one major frame ($t=p_1 \cdot p_2 \cdot p_3$) for $pr(cont)=1/3$ and $pw_{fast}=2$, $pw_{slow}=1$.
  - With DVS *minimum power consumption = 1906.66* and *maximum power consumption = 1922.65*
  - Without DVS *power consumption = 2240*
  - Thus expected savings between 14% and 14.8%.

Conclusions

- We have developed a timed, probabilistic, process algebra that allows modeling the power consumption of system resources
- Various techniques for quantitative analysis of power properties have been developed and implemented in the PARAGON toolset
  - Probabilistic bounds computation
  - Model checking
- Research direction:
  - Uniform resource attribute model

ACSR-VP
for design synthesis and parametric analysis

Example: A Start-time Assignment Problem



- Start-time Assignment Problem with Inter-job Temporal Constraints
- The order of execution of job is not known
- Goal is to statically determine the range of start times for each job so that jobs are schedulable and all inter-job temporal constraints are satisfied.

ACSR-VP (ACSR With Value-passing)

- Extends ACSR with
  - variables: (a?x,1).(c!x,1)...
  - value passing communications: (c!7,1)... || (c?x,1)...
  - parameterized processes: P(x) = (x > 1) → (a!x,1).nil
- Priorities can be specified using expressions
  - timed actions: {(data, y+1)}
  - instantaneous events: (signal!8, x+3)
- Syntax

$$
\begin{array}{lcl}
P & ::= & NIL \mid a.P \mid A:P \mid P+P \mid P \parallel P \\
 & & b \rightarrow P \mid P \setminus F \mid [P]_I \mid \mathbf{C} \\
a & ::= & (\tau, e) \mid (c?x, e) \mid (c!e_1, e_2) \\
A & ::= & \varnothing \mid \{S\} \\
S & ::= & (r, e) \mid (r, e), S \\
\mathbf{C} & ::= & \mathbf{X} \mid \mathbf{X}(\vec{v})
\end{array}
$$

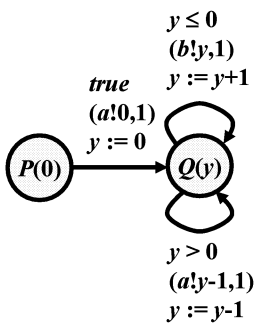
Symbolic Graph With Assignment (SGA)

SGA is a directed graph with edges labeled with b,α, and θ, where b is a Boolean condition, α is an action, and θ is an assignment.

We use SGA to capture the semantics of ACSR-VP

$$P(x) = (a!x,1).Q(x)$$
$$Q(y) = (y \leq 0) \rightarrow (b!y,1).Q(y+1)$$
$$+ (y > 0) \rightarrow (a!y\text{-}1,1).Q(y\text{-}1)$$

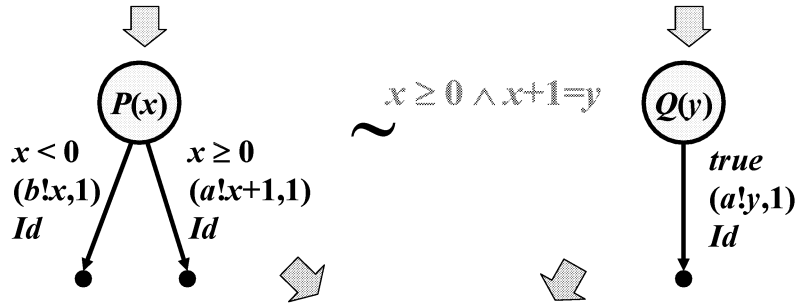
$$P(0) \Rightarrow (a!0,1).(b!0,1).(a!0,1)\ldots$$



Symbolic Bisimulation (Informal Description)

$P(x) = (x < 0) \rightarrow (b!x,1).nil$
$\quad + (x \geq 0) \rightarrow (a!x+1,1).nil$

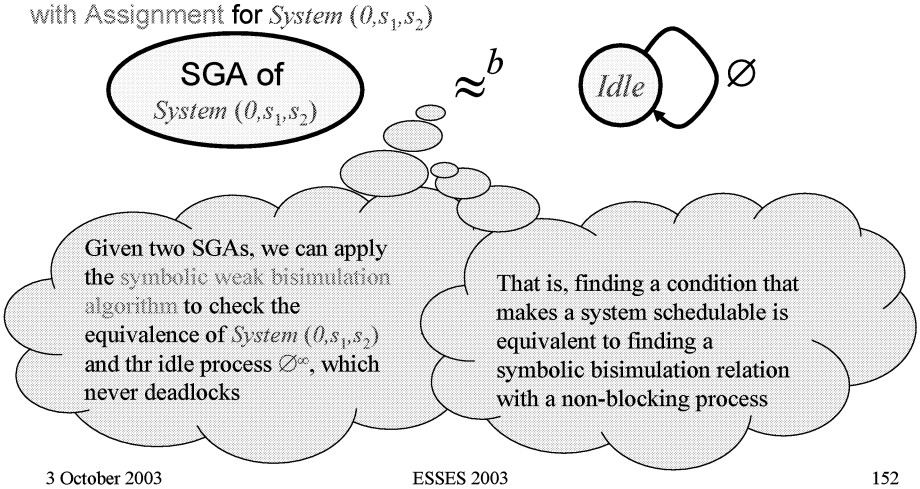
$Q(y) = (a!y,1).nil$



$$X_{PQ}(x,y) = (x < 0 \Rightarrow \mathit{false}) \wedge (x \geq 0 \Rightarrow (\mathit{true} \wedge x+1 = y)) \wedge (\mathit{true} \Rightarrow (x \geq 0 \wedge y = x+1))$$

Schedulability Analysis Using Symbolic Bisimulation

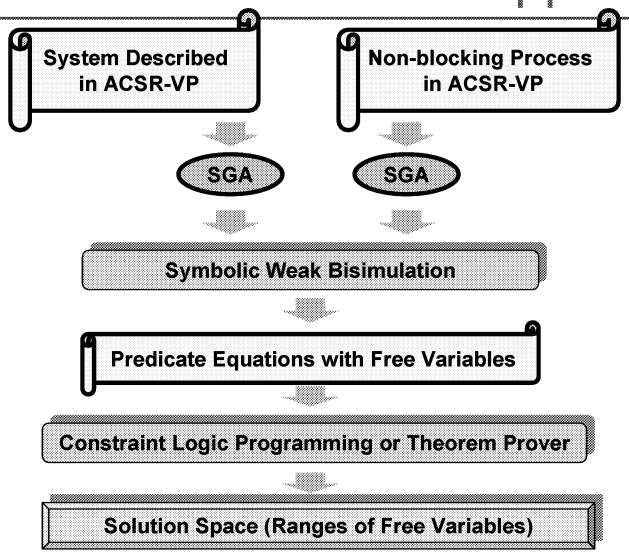
Suppose we have an ACSR-VP term *System* $(0,s_1,s_2)$ that model a real-time system or a scheduling problem. We generate the Symbolic Graph with Assignment for *System* $(0,s_1,s_2)$



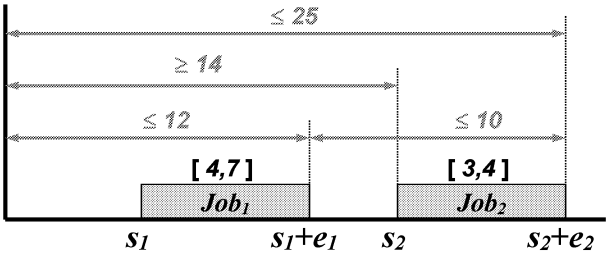
ACSR-VP approach

- Provides a formal framework for modeling real-time systems, especially for real-time scheduling problems such as
  - Priority Assignment Problem
  - Execution Synchronization Problem
    - Start-time assignment problem
    - Period assignment problem
- Deals with unknown parameters in the problems rather than "yes/no" answer (i.e., parametric approach)
- Provides a fully automatic method for the analysis of real-time scheduling problems
- Takes advantages of existing techniques such as integer programming and BDD

Overview of General Approach



Example: Start-time Assignment Problem



- Start-time Assignment Problem with Inter-job Temporal Constraints
- Goal is to statically determine the range of start times for each job so that jobs are schedulable and all inter-job temporal constraints are satisfied.

Modeling With ACSR-VP

- The following fragments of ACSR-VP describe the start time assignment problem with inter-job temporal constraints

$$Job_i(t,s) = (t < s) \rightarrow \varnothing : Job_i(t+1,s) + (t = s) \rightarrow (Start!,1).Job'_i(0,t,s)$$

$$Job'_i(e,t,s) = (e < e_i^-) \rightarrow \{(cpu,1)\} : Job'_i(e+1,t+1,s) + (e = e_i^-) \rightarrow Job''_i(e,t,s)$$

$$Job''_i(e,t,s) = (e < e_i^+) \rightarrow \{(cpu,1)\} : Job''_i(e+1,t+1,s) + (e \leq e_i^+) \rightarrow (Finished!,1).Idle$$

$$Constraint(t) = (start?,1).Constraint_1(t) + \varnothing : Constraint(t+1)$$

$$Constraint_1(t) = (Finished?,1).Constraint_2(t) + \varnothing : Constraint_1(t+1)$$

$$Constraint_2(t) = (t \leq 12) \rightarrow Constraint_3(t,0)$$

$$Constraint_3(t) = \ldots$$

$$System(s_1,\ldots,s_n) = (Job_1(0,s_1)||\ldots||Job_n(0,s_n)||Constraint(0))\backslash\{Start,Finished\}$$

Predicate Equations

- The following fragments of predicate equations are generated from the symbolic weak bisimulation algorithm with the infinite idle process

$$
\begin{aligned}
X_0(t, s_1, s_2) = {} & (t \le 5 \wedge t < s_2) \rightarrow X_1(t+1, s_1, s_2) \\
& \wedge (t \le 5 \wedge t = s_1) \rightarrow X_2(0, t+5, s_2) \\
& \wedge (\ (t \le 5 \wedge t < s_1 \wedge X_1(t+1, s_1, s_2)) \\
& \quad \vee (t < 5 \wedge t = s_1 \wedge X_2(0, t+5, s_2)) \ ) \\
X_1(t, s_1, s_2) = {} & \ldots X_2 \ldots \\
X_2(e, s_1, s_2) = {} & \ldots X_1 \ldots
\end{aligned}
$$

To get the values of s_1 and s_2, we can ask a query $X_0(0, s_1, s_2)$

Solution Space

- The solutions to the predicate equations can be obtained using linear/integer programming techniques, constraint logic programming techniques, or a theorem prover.
- The solutions for the previous example are:

Start time S_1	3	4	4	5	5	5
Start time S_2	14	14	15	14	15	16

An Automatic Approach

- The disadvantage of symbolic weak bisimulation is that it requires to add new τ edges into SGA. This will increase the size of predicate equations
- The disadvantage of CLP is that there is no guarantee that it terminates

- Reachability Analysis: Finding a condition that makes a system schedulable is equivalent to finding a condition that guarantees there is always a cycle in an SGA regardless of a path taken
  - No need to add new τ edges
- Restricted ACSR-VP
  - Give syntactic restriction to identify a decidable subset of ACSR-VP
    - Control Variable : in finite range; Values can be changed
    - Data Variable : could be in infinite range; Values cannot be changed
    - $P(x{:}0..100,y) = (x<0 \wedge x+y>10) \rightarrow \emptyset{:}Q(x+3, y)$
  - Generate a boolean expression or boolean equations (i.e., no need to use CLP)

Conclusions: resources

- We have presented a family of resource-bound process-algebraic formalisms
  - the notion of a resource plays central role
    - Abstractions of physical resources
    - Resource sharing: coordination and synchronization
    - Resource consumption takes time: real-time behavior
    - Resource failures: probabilistic behavior
- Sample application domain: analysis of scheduling problems
  - Other domains: protocol analysis, rapid prototyping

Conclusions: analysis techniques

- Analysis of safety properties by means of deadlock detection
- Conformance analysis by means of equivalence and preorder checking
- Probabilistic analysis techniques:
  - Model checking
  - Resource utilization
- Parametric analysis in ACSR-VP

Extensions

- Presented: serially reusable resources with access constraints
- Other types of resources:
  - Consumable resources: each resource use depletes resource stock
  - Multi-capacity resources: allow simultaneous access by a limited number of processes
- Other kinds of resource constraints:
  - non-functional constraints such as memory, power consumption, weight, etc.

References

- "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," Insup Lee, Patrice Br\'emond-Gr\'egoire and Richard Gerber, Proceedings of the IEEE, Jan 1994, pp. 158-171.
- "A Complete Axiomatization of Finite-state ACSR Processes," Patrice Br\'emond-Gr\'egoire, Jin-Young Choi and Insup Lee, Information and Computation, 138 (2), Nov 1997, pp. 124-159.
- " Process Algebra of Communicating Shared Resources with Dense Time and Priorities," by Patrice Br\'emond-Gr\'egoire and Insup Lee, Theoretical Computer Science, 189, 1997, pp. 179-219.
- "A Process Algebraic Approach to the Schedulability Analysis of Real-Time Systems," Han\^ene Ben-Abdallah, Jin-Young Choi, Duncan Clarke, Young Si Kim, Insup Lee and Hong-Liang Xie, Real-time Systems, 15, 1998, pp. 189-219.
- "Probabilistic Resource Failure in Real-Time Process Algebra," Anna Philippou, Oleg Sokolsky, Insup Lee, Rance Cleaveland, and Scott Smolka, CONCUR '98, Sept 1998.
- "Specification and Analysis of Real-Time Systems with PARAGON," Oleg Sokolsky, Insup Lee, and Han\^ene Ben-Abdallah, Annals of Software Engineering, 7, 1999, pp. 211-234.
- "Modeling and Analysis of Power-Aware Systems," Oleg Sokolsky, Anna Philippou, Insup Lee, and Kyriakos Christou, TACAS 2003, April 2003.
- These papers are also available on-line from www.cis.upenn.edu/~rtg/papers.php3.

Thanks

- … for invitation to ESSES 2003
- … for fundamental work done by my former Ph.D. students:
  - Amy Zwarico
  - Rich Gerber
  - Patrice Bremond-Gregoire
  - Hanene Ben-Abdallah
  - Duncan Clark
  - Hee Hwan Kwak
- … for support from ARO, NSF, ONR over a number of years

Q & A

Sponsored by: