# OUTLINE

- Introduction
  - Lecture 1: Motivation, examples, problems to solve
- Modeling and Verication of Timed Systems
  - Lecture 2: Timed automata, and timed automata in UPPAAL
  - Lecture 3: Symbolic verification: the core of UPPAAL
  - Lecture 4: Verification Options in UPPAAL
- Towards a Unified Framework
  - Lecture 5: Modeling, verification, real time scheduling, code synthesis
    From UPPAAL to TIMES



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



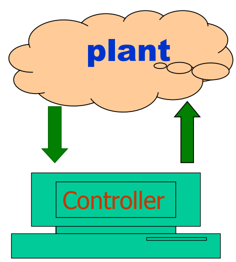
# "Who is Who" in Real Time Systems

- 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)




The main components of a controller
a set of tasks: P1, P2 ... Pn running on
a platform (RTOS: scheduler)

P1 || P2 || ...|| Pn || Scheduler



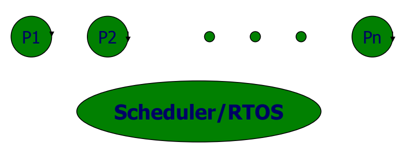
# The design problem

- A set of computation tasks
  - Timing constraints: e.g. Deadlines
  - (QoS constraints: 80% of deadlines met, liveness?)
  - Release patterns i.e Task models
- Design a controller/Schedule
  - To ensure the constraints



# "Classic" Real Time Scheduling

- Periodic tasks




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

## Rate-Monotonic Scheduling

- P1...Pn arrive at fixed rates
- Fixed Prioirity Order: Higher frequency => Higher priority
- Always run the task with highest priority (FPS)
  P1 || P2 || ...|| Pn || FPS
- Schedulability can be tested by utilization bound (or equation solving)



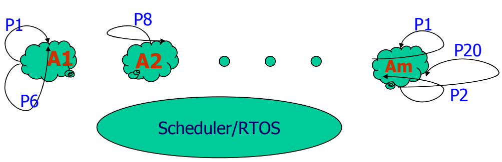
## In real life, tasks may

- share many resources (not only CPU time)
- have complex control stuctures and interactions
- have to satisfy mixed logical, temporal & resouce constraints



## Automata-based Approaches

A controller = a set of timed automata accepting tasks Pi's




How to schedule tasks/automata?



## The TIMES project

Tools for Modeling and Implementation of Embedded Systems

Uppsala University



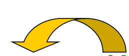
# Vision

- Timed Model to Executable Code Guaranteeing Timing Constraints
- Timing analysis of Concurrent and Time-Critical Software
  - Response time estimation



## Why this work?

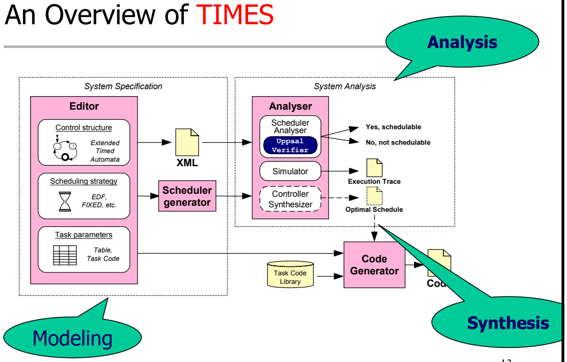
- Timed Automata, [Alur&Dill 1990]
  - generated a lot of work on model checking:



**P** M sat φ ?

  - Consider M as a design specification
  - Construct a program **P** from M

## An Overview of TIMES






## 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
  - (similar to Rate-Monotonic Scheduling)

**Implemented in the TIMES tool**

- Undecidability [TACAS 04]
  - The execution times of tasks are intervals
  - Task completion times influece task release times
- TIMES demo



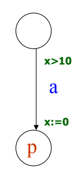
# The MODEL

(Timed Automata with Tasks)



## Modelling Real Time Systems

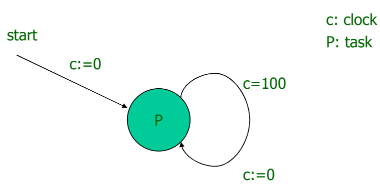
- Events
  - synchronization
  - interrupts
- Timing constraints
  - specifying event arrivals
  - e.g. Periodic and sporadic
- Tasks (executable programs)
  - interrupt processing
  - Internal computation
  - triggered by events and scheduled in the reday queue of RTOS




Timed Automaton + tasks



## Example: periodic tasks





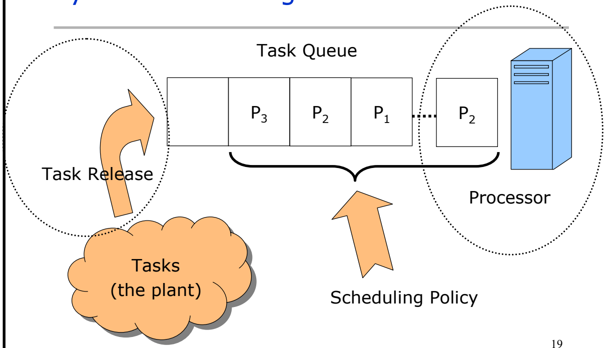

## Tasks = Executable Programs (e.g. C, Java)

- Task parameters:
  - C: WCET
  - D: Relative Deadline
  - (other parameters for schedulling e.g. Priority)
- Task Interface:

**Task P**
**{**
**$v_1$ := $F_1(v_1 \ldots v_n)$**
**…**
**$v_n$ := $F_n(v_1 \ldots v_n)$**
**}**

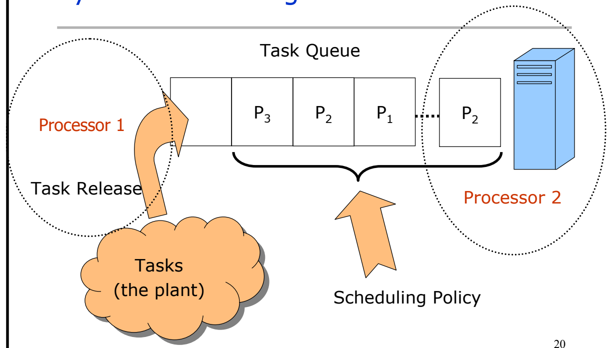
(a set of variables updated)

## System's Processing Unit



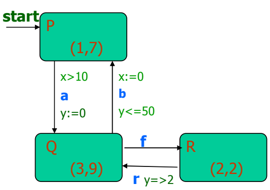



## System's Processing Unit






## Timed Automata with Tasks (Example)




- Processor 1 (event handler)
  - Initially, P in the queue
  - Run-to-Completion/Stablization
    - Whenever **a** available and x>10, Q is put in the queue
    - Then
      - Whenever **b** available and y<=50, P is put in the queue
      - Whenever **f** available, R is put in the queue.
- Processor 2 (task handler)
  - Schedule and Compute 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
    - 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



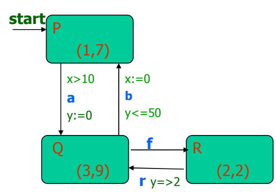
## States/Configurations of automata

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

- m — Location (node)
- u — clock assignment (valuation)
- q — 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)]



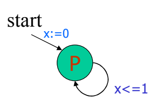
## Semantics (as transition systems)

- 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) −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)

- m → location
- u → clock assignment
- q → 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
  - $Sch(q)= [P_1(c_1,d_1)P_2(c_2,d_2)\ldots P_n(c_n,d_n)]$ and
  - $(c_1+\ldots+c_i)<=d_i$ 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



# 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.



## Proof ideas (1):

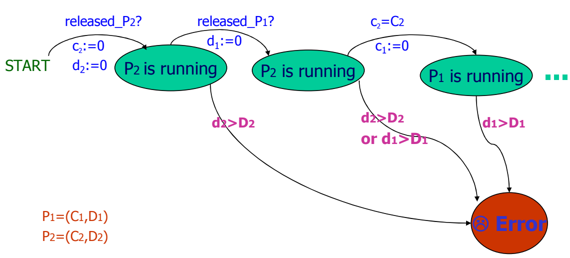
Size of schedulable queues is bounded

- The maximal number of instances of $P_i$ in a schedulable queue is bounded by $M_i = \lceil D_i/C_i \rceil$
- The maximal size of schedulable queues is bounded by $M_1 + M_2 + \ldots + M_n$
- To code the queue/scheduler, for each task instance, use 2 clocks:
  - $c_i$ remembers the computing time
  - $d_i$ remembers the deadline

| | | |
|---|---|---|
| • • • | $(c_i,d_i)$ | • • • |



## Proof ideas (2): The scheduler as an automaton

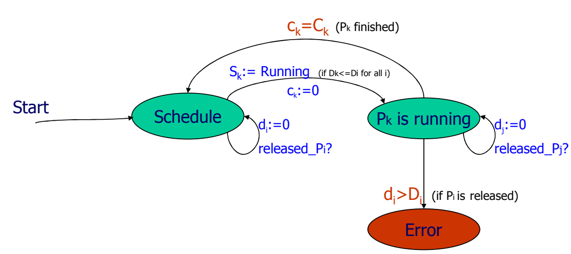



$P_1=(C_1,D_1)$

$P_2=(C_2,D_2)$



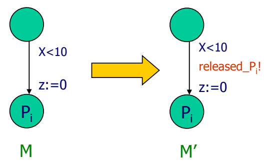
## The scheduler automaton




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

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

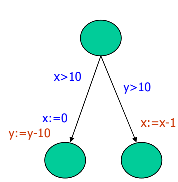
NOTE
- Reachability for BSA is decidable
- Preemptive EDF is optimal; thus the general schedulability checking problem is decidable.



## Timed automata with subtraction

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

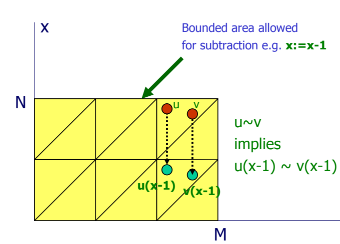
- Subtraction automata are timed automata extended with subtraction on clocks
- That is, in addition to reset x:=0, it is also allowed to update a clock x with x:= x-n where n is a natural number






## 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).

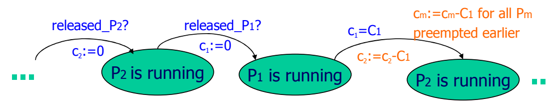



**FACT:**
Location Reachability checking is decidable!

## Schedulability Checking
## as a reachability problem for
## Bounded Subtraction Automata



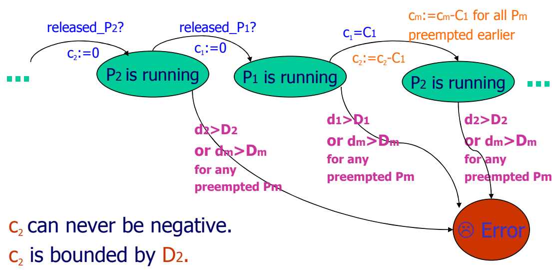
## Proof ideas (no stop but subtraction :-)




- Model the scheduler as a subtraction automaton
  - Do not stop the computing clock $c_2$ 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 $D_2$.



# END of proof



## Complexity

#clocks (needed)
= 2 x #instances (maximal number of schedulable task instances)
= 2 x $\sum_i \lceil Di/Ci \rceil$

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

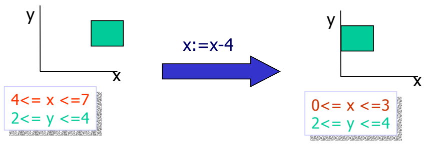


## It works anyway !!!

- #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)






## WE CAN DO BETTER ! [TACAS 03]

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



# The 2-CLOCK ENCODING

(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



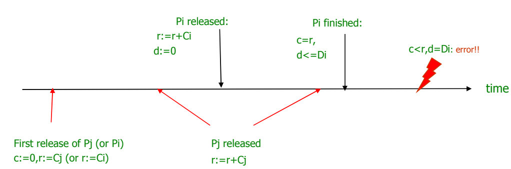
## To code the queue/scheduler, we need:

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



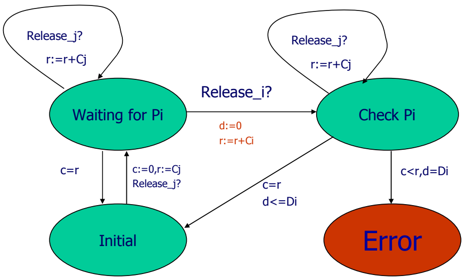
## Intuition of the encoding: $Ri = Ci + \Sigma_{pri(Pj)>pri(Pi)} Cj$

– Assume: priority(Pj) > priority(Pi)  and Pi is analyzed




When Pi finishes, r = Ri

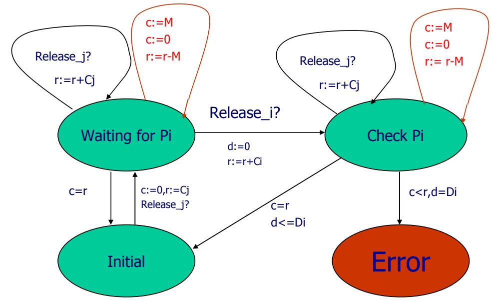
## The "FPS scheduler": analyzing Pi




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

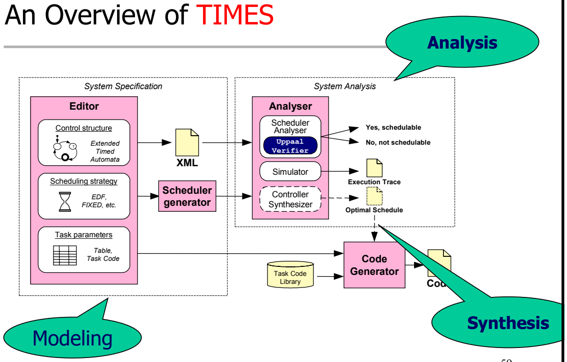
- c is bounded by M
- r is bounded by $r_{max}$ + Ci
  - Where $r_{max}$ is the maximal value of r from previous analysis for all tasks Pj with higher priority

So the scheduler is a standard TA END





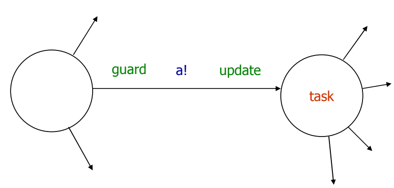

## An Overview of TIMES






## 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)

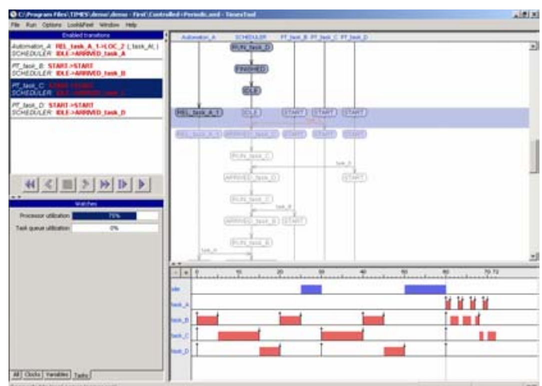


## CODE SYNTHESIS in TIMES

- 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



## Conclusions/Remarks

- A unified model for timed systems (can express synchronization, computation and complex temporal and resource constraints).
- The first decidability result (and efficient algorithms) for preemptive scheduling in dense time models:
  - The analysis is symbolic (using DBM's in the UPPAAL tool)
  - The results can be adopted for schedulability analysis of message transmission.
- Implementation: **TIMES**