Synthesis of Application Specific Schedulers for Heterogeneous Real-Time Systems

Guaranteeing Safety & Allowing QoS Extensions

Christos KLOUKINAS

Christos.Kloukinas@imag.fr

Verimag

(http://www-verimag.imag.fr/)

Grenoble, France

Introduction

- Heterogeneous Applications
  - Periodic, Aperiodic, Sporadic threads
  - Difficult to analyse their behaviour
- Using PIP/PCP for deadlock avoidance
  - Difficult to get priorities right (may need to split threads @ I/O points, at least when modelling)
  - PIP/PCP are ***pessimistic***
- RMA, EDF, *etc.*
  - RMA is only for Periodic (must group non-periodic), max U = 69%
  - EDF : overhead, unstable under overload, priority inversion, if not schedulable you don't know why
- Extending scheduling with ***QoS*** constraints ???

Our Goal

Analyse an application & ***synthesise*** a scheduler which :

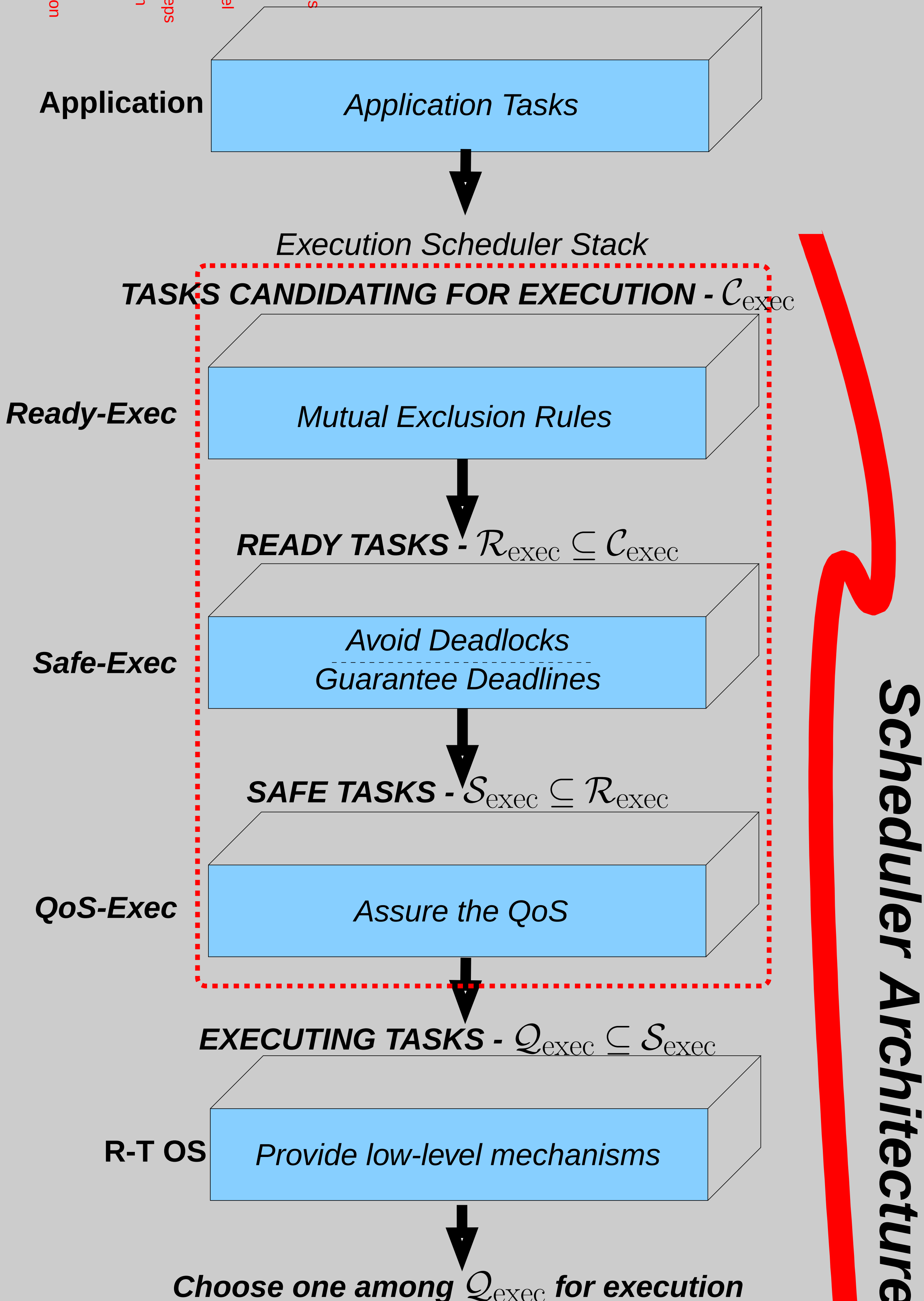
- ***Guarantees Safety*** - no Deadlocks / Deadline Misses
- is not as ***Pessimistic*** as PCP
- is easy to ***extend*** with additional ***QoS constraints***

Meant to be used on single-processor systems

States of Threads

Application threads can be :

- ***Blocked***
- ***Ready*** to execute = $\neg$ ***Blocked***
- ***Safe*** to execute $\subseteq$ ***Ready***
- ***Executing*** (*at most one*) $\in$ ***Safe***


Application
Application Tasks
Execution Scheduler Stack
TASKS CANDIDATING FOR EXECUTION - $\mathcal{C}_{\text{exec}}$
Ready-Exec
Mutual Exclusion Rules
READY TASKS - $\mathcal{R}_{\text{exec}} \subseteq \mathcal{C}_{\text{exec}}$
Safe-Exec
Avoid Deadlocks
Guarantee Deadlines
SAFE TASKS - $\mathcal{S}_{\text{exec}} \subseteq \mathcal{R}_{\text{exec}}$
QoS-Exec
Assure the QoS
EXECUTING TASKS - $\mathcal{Q}_{\text{exec}} \subseteq \mathcal{S}_{\text{exec}}$
R-T OS
Provide low-level mechanisms
Choose one among $\mathcal{Q}_{\text{exec}}$ for execution
Scheduler Architecture

Scheduler Layers

- ***Ready-Exec:*** which threads are ***Ready*** ?
  (mutual-exclusion, waiting for notification)
- ***Safe-Exec:*** which among the ***Ready*** are ***Safe*** ?
  (no deadlocks, deadlines honoured)
- ***QoS-Exec:*** which of the ***Safe*** meet the ***QoS*** req. ?
  (speed, memory, energy)
- The Real-Time OS picks one among the latter as the ***Executing*** thread

We synthesise the ***Ready-Exec*** & ***Safe-Exec*** layers, considering a *maximal* ***QoS-Exec*** scheduler (*i.e.*, all ***Safe*** threads meet the ***QoS*** requirements) — the user provides a different ***QoS-Exec*** scheduler, if he so wishes.

QoS Policies

- What is a QoS policy (our definition) ?
  - Something that helps increase some “goodness” metric
  - Something that cannot kill you
    So, it’s only applicable on ***safe*** threads
- Examples :
  - (Locally) Minimise Context-Switches
    Helps with optimising Speed, Memory accesses (fewer cache flushes/misses), Energy . . .
- Exception : (there’s always one . . .)
  - Non-Preemption
    Too important to forbid – for network messages, it’s the only way

Scheduler Synthesis - The Casting

- The ***Good : Scheduler***
- The ***Bad : Environment*** (*i.e.*, Time)
  Completely uncontrollable
- And the ***Ugly : Application***
  Only controllable at pre-specified points

Scheduler Synthesis

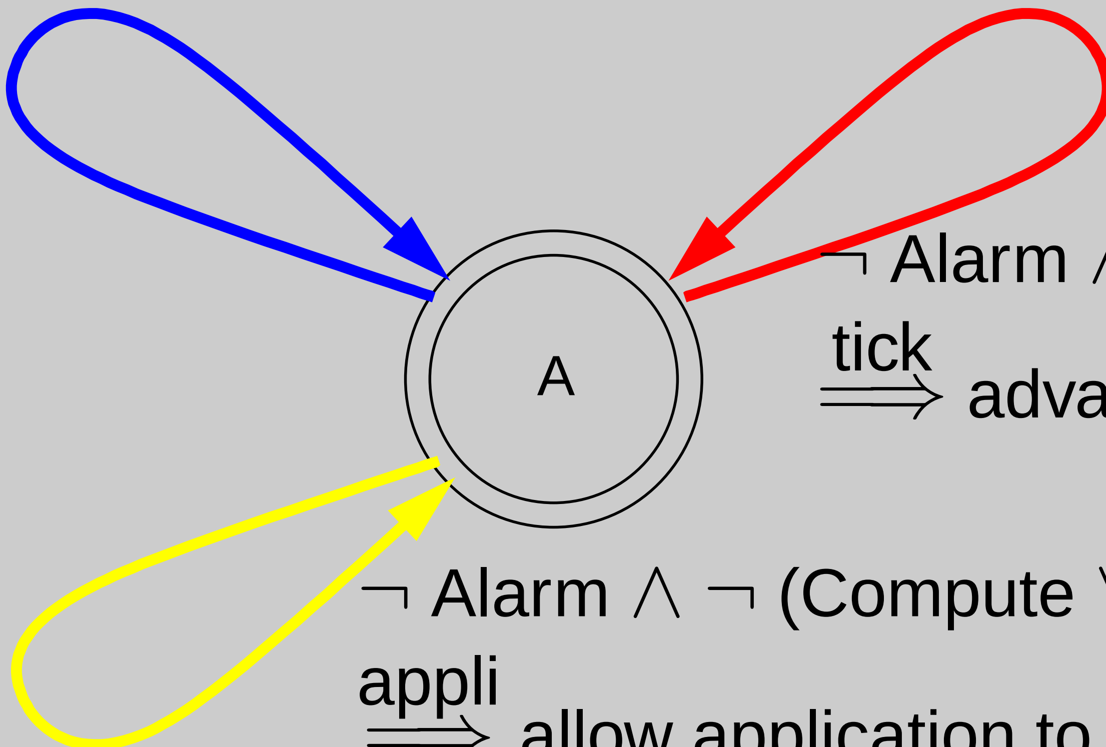
- Even though in theory we could control each instruction, in practice we can only observe & control instructions of the type:
  - ***lock/unlock*** ;
  - ***wait/waitTimed*** ;
  - ***waitForNextPeriod*** ; and
  - ***timer expirations***
- ***No re-scheduling at notify/notifyAll !***
  - The notified thread(s) immediately block on the (re-)lock, so no need to re-schedule

Scheduler Synthesis - II

- Effectively, a two-player game,
  Environment *versus* ***Scheduler*** ,
  where the ***Scheduler*** is aided by the ***Application***
- Each player moves in turn
- Scheduler/Controller Synthesis amounts to :
  - Find a winning strategy for the game
    "What action must I (*i.e.*, the ***Scheduler***) take to prevent my opponent (*i.e.*, the ***Environment***) from winning (*i.e.*, cause the ***Application*** to miss deadlines) ?"

System Model - The Environment

Alarm $\overset{alarm}{\Longrightarrow}$ change state & reschedule



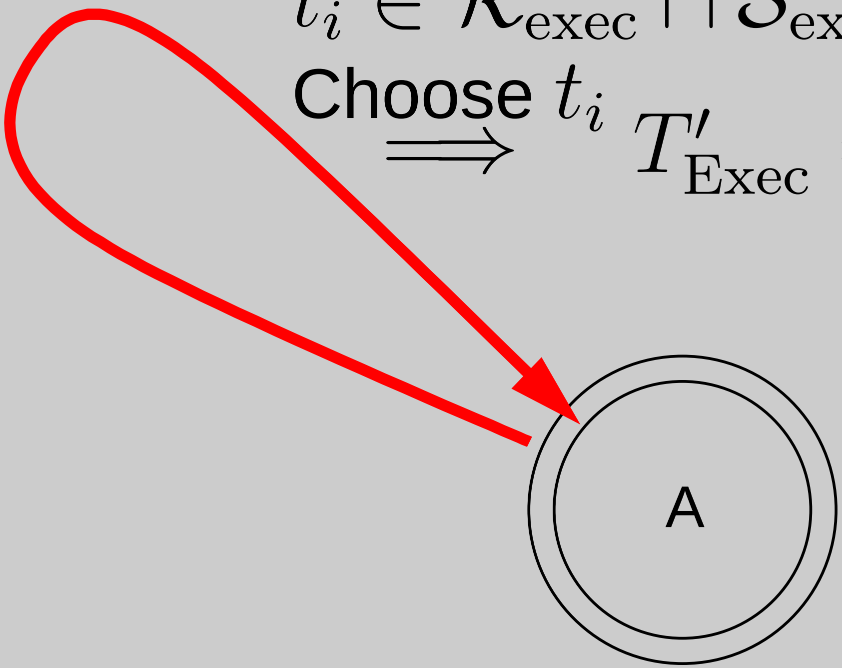
$\neg$ Alarm $\wedge$ (Compute $\vee$ wait) $\overset{tick}{\Longrightarrow}$ advance clocks

$\neg$ Alarm $\wedge$ $\neg$ (Compute $\vee$ wait) $\overset{appli}{\Longrightarrow}$ allow application to run

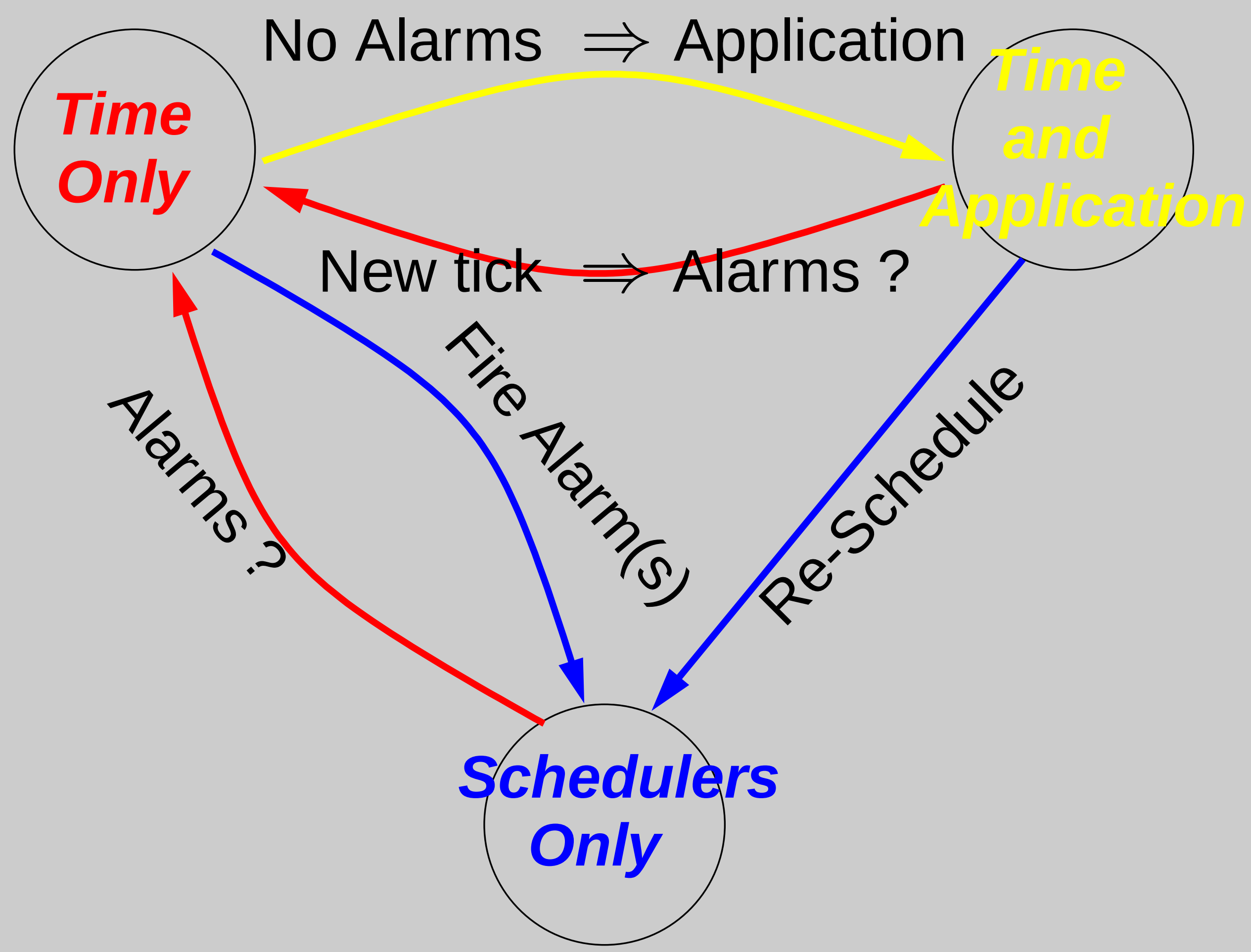
System Model - The Scheduler

$$t_i \in \mathcal{R}_{\text{exec}} \cap \mathcal{S}_{\text{exec}} \cap \mathcal{Q}_{\text{exec}}$$

Choose $t_i \implies T'_{\text{Exec}} = t_i$



System Execution Modes



Scheduler Synthesis Basic Idea

- Construct the entire state space
- Find ***Deadlock*** states
  - These identify states where the ***Application*** deadlocked due to a cycle in the shared resources; and
  - states where the ***Application*** missed a deadline.

Scheduler Synthesis Basic Idea - II

- From each deadlock state:
  - Go backwards in the trace up to the first state where the scheduler could have made this trace impossible (*i.e.*, don't allow some thread to execute)
  - All these constraints are effectively the constraints we need for implementing the ***Safe*** scheduler layer.
  - So, ***Safe-Exec*** is a table, indexed by a system state, containing the ***Application*** threads which we must not allow to execute at that state
- If no thread can be scheduled at some state, add the state to the deadlocked states
- If no thread can be scheduled at the initial state, there is no ***Safe*** scheduler for this application

Synthesis Steps

- Use the ***untimed*** model to remove deadlocks
  - Obtain a ***Deadlock-Free*** scheduler for any combination of underlying hardware and algorithms used for the computations
    - avoid ***dormant*** deadlocks
    - be ***robust*** against wrongly estimated execution durations
    - ***know*** if you got it ***really*** wrong . . .
  - Do so using a much smaller model than the ***original timed*** one
  - Use the ***Deadlock-Free*** scheduler to constrain the ***timed*** model
  - Find additional constraints for avoiding missed deadlines

State Space Reduction

- ***Branching Bisimulation Equivalence (bbe)*** reduction
  - Series of consecutive ticks are substituted by a single “super” tick
  - A ***safe reduction*** , since we could not perform scheduling anywhere inside such a series in the first place
  - We only pay for the ticks we can ***observe*** . . .
  - A ***74%*** reduction of the state space on a small example

State Space Reduction - II

- ***Non-Preemption***
  - When an alarm fires, while some thread is computing, we never preempt it
  - Conservative reduction - we may fail to find a scheduler
  - Sometimes this is the reality (messages in networks)
  - If negative, the system is ***overloaded*** and we identify the set of problematic threads
  - A ***40%*** reduction (on its own), ***80%*** when combined with *bbe*
- Use the constraints found in the previous steps (deadlocks,non-preemption) to constrain the system when preemption is allowed

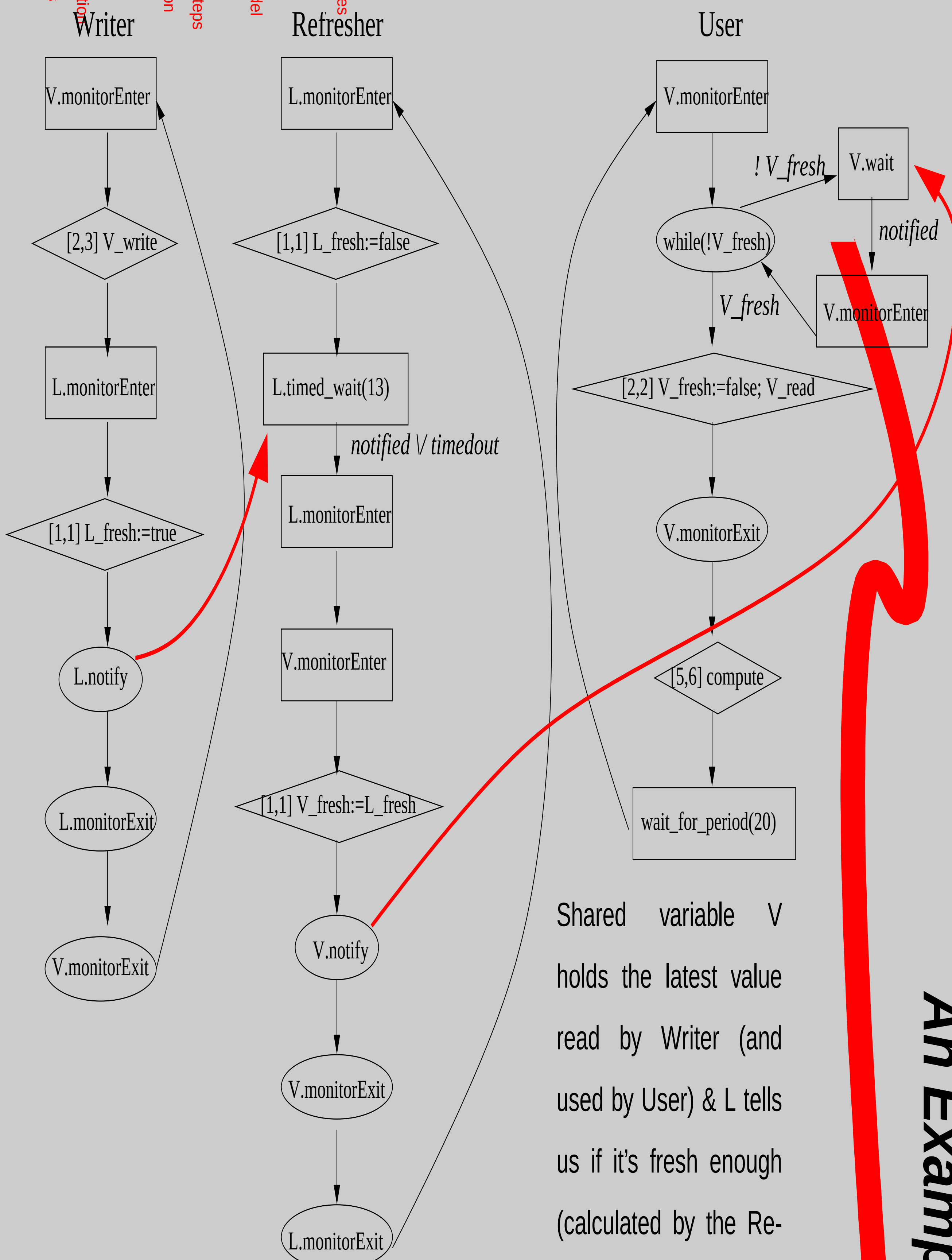
Synthesised Scheduler

- Constraints : $\overrightarrow{PC_i} \wedge \overrightarrow{Clocks_i} \Rightarrow \text{Do not execute } \overrightarrow{Thr_{i_j}}$ (where i is a state in the state space)
- The scheduler needs to examine the *PC*'s and the Clocks of all the threads to identify the current state i
- ***Reality Check :***
  - Observing clocks is ***costly***
  - Using clocks (*i.e.*, ***watchdogs***) is not for free either
  - Can we avoid paying this overhead ?
    - Try to find a set of constraints which are ***independent*** of time, *i.e.*, $\overrightarrow{PC_i} \Rightarrow \text{Do not execute } \overrightarrow{Thr'_{i_j}}$

No Clocks

- Removing clocks from constraints can cause deadlines to be missed
- So, remove clocks & synthesise again (by now the system is reduced ~***89%***)
- If it fails what ?
  - The application is not ***robust*** with respect to time !
  - Either :
    - Rewrite it (change algos / hardware); or
    - Use the safe scheduler which examines the clocks but make sure it ain't too slow !

An Example



Shared variable V holds the latest value read by Writer (and used by User) & L tells us if it's fresh enough (calculated by the Refresher)

Example (continued)

- Original Timed Model: 45.5 K States
- 10 states in the untimed model are Deadlock states
- It Misses a Deadline in 367 states in the timed model
- Synthesised :
  - 8 constraints for avoiding Deadlocks
  - 30 constraints for guaranteeing Deadlines when QoS =Non-Preemption
  - 18 additional constraints when allowing Preemption & no longer observing clocks
- 56 constraints in total – final ***safe*** state space has 1.6 K states

Implementation

- From Java application to abstract model : By hand
- From abstract to detailed model (slicing, *etc.*) : AWK
- CADP tools for the *bbe* reduced state space
- Synthesis algorithm : Lisp
- Constraints to C code : AWK script
- Basic library for eCos: C
- 330 MHz Pentium II – Min/Avg/Max/Avg-Dev (in μs)
  - ***Schedule*** : ***0.00 / 0.66 / 4.00 / 0.45***
  - Context Switch : ***0.00 / 0.77 / 1.00 / 0.35***
  - Trylock (unlocked) : ***0.00 / 0.69 / 2.00 / 0.47***
  - Unlock (locked) : ***0.00 / 0.75 / 3.00 / 0.47***

Conlcusions

- Weak points :
  - Initial model hand-constructed – some early results let us believe we can solve this
  - We give the CPU to a thread by ***stopping*** the rest & lock the underlying scheduler at certain points – an implementation with priorities in user space
- Strong points :
  - Completely automatic analysis – no need for RMA, selecting priorities by hand, *etc.*
  - Durectly usable with ***heterogeneous*** applications
  - Helps in ***analysing*** an application – separates constraints for deadlocks, deadlines (under non-preemption, preemption)
  - Scheduling is easily ***extendible*** with QoS policies

Future

- Forward, On-the-fly synthesis
- Currently away in the mist :
  - Scheduling for memory – minimise total memory needs / minimise cache misses
  - Scheduling for energy – minimise / stabilise energy consumption
  - Multiprocessor systems, with different clock speeds
  - Automate WCET estimation of basic blocks
  - ***Simple*** Model-checker – the time automaton can be an observer
- ***"Optimum"*** Schedulers – variables, order, cost, *etc.*