Universit□t Salzburg

Schedule-Carrying Code

Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic

SOFTWARE RESEARCH LAB

Contents

- Giotto
- Schedule Carrying Code
- Real Time Programming
- Elements of Formal Models for Real Time Systems
- Giotto Model, FLET, Tool Chain
- E Code and S Code
- The E Machine and S Machine
- E Code and S Code Instructions and Interpreters
- Simplified Flight Control Example
- Generating vs Checking SCC

Giotto

Giotto is a formal model/ high level programming language for RTS based on FLET and periodic tasks.

Advantages:

- Separation of functional and timing concerns
- Value and time determinism
- Predictability
- Platform-independence
- Multi-modal support (mode switching)
- Compossability

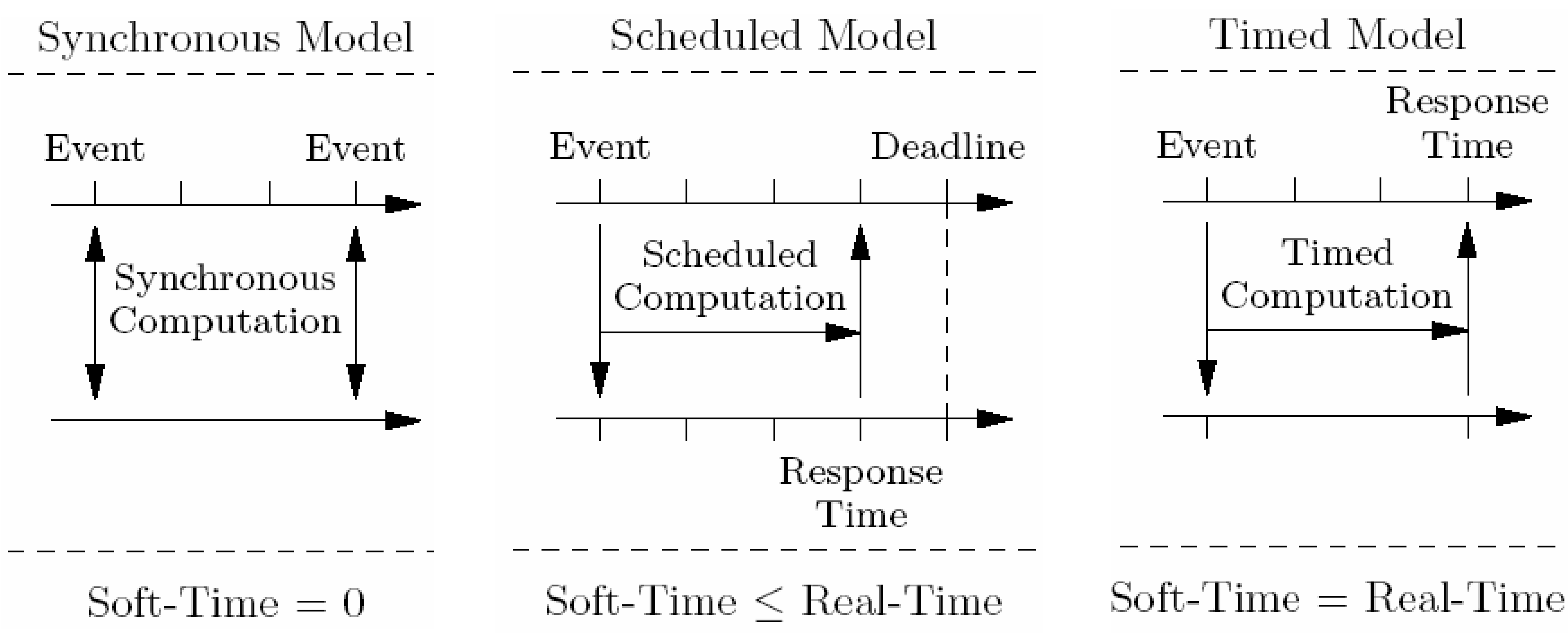
Schedule Carrying Code

SCC is real time code annotated with the description of a schedule, which witnesses the schedulability of the code.

Advantages

- Produced by the compiler
- Produced once and revalidated and executed with each use, on a given platform.
- More efficient
- More flexible
- It is easier to prove time safety.
- Reuse of proofs

Models for Real Time Programming

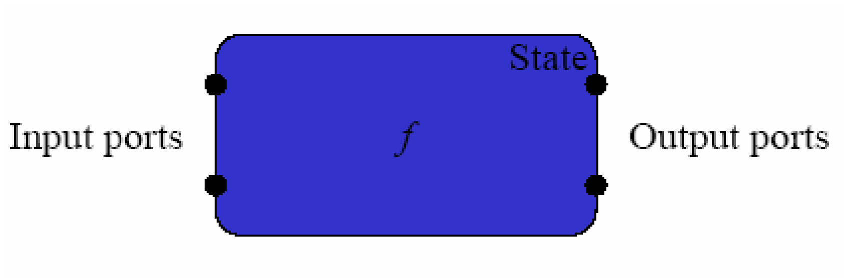


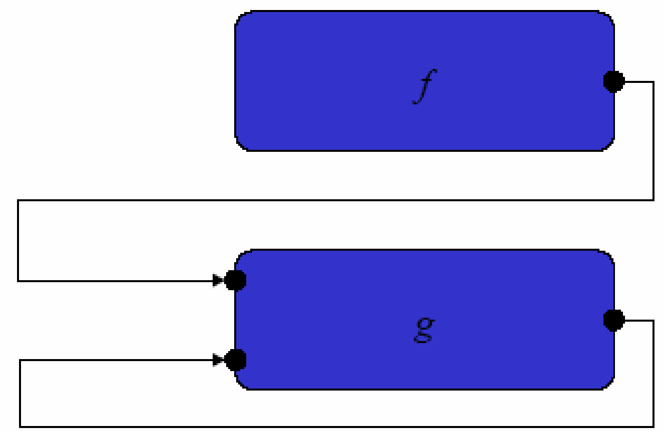
Universit□t Salzburg

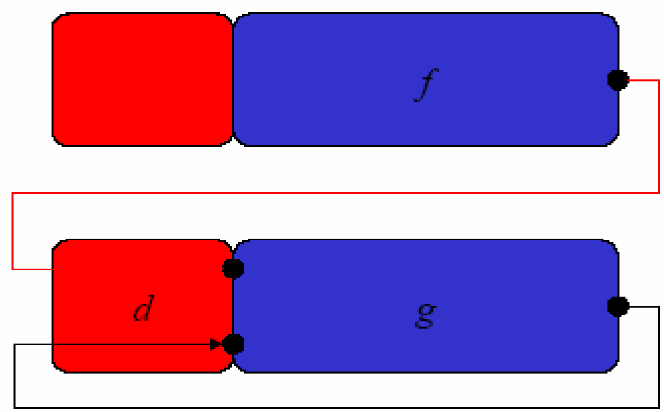
Elements of Formal Models for Real Time Systems

- Components and connectors
- Functional and control description of components
- Time issues
- Environment behavior description
- Processing issues
- Verification technique
- Complexity

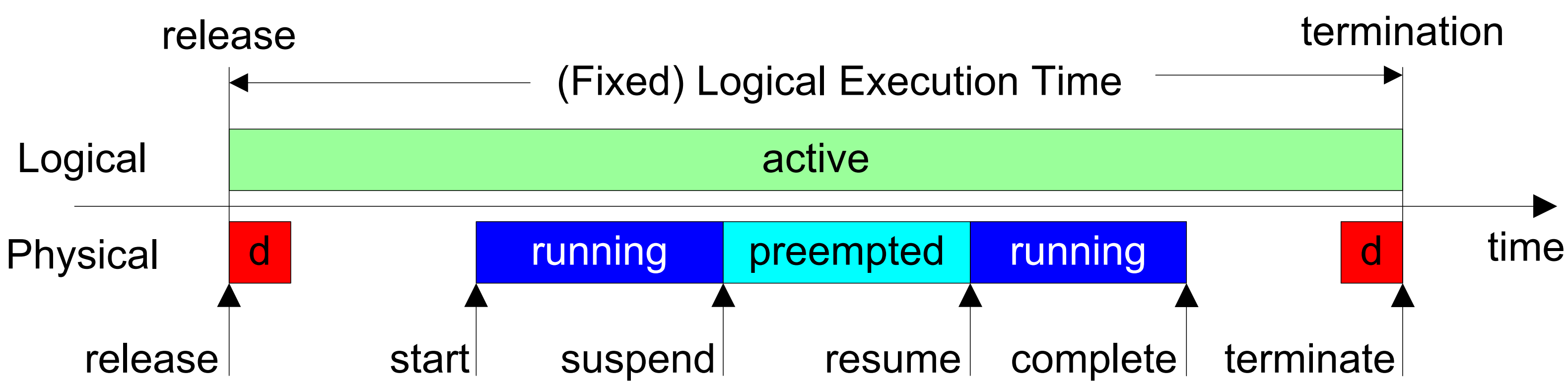
Giotto Model





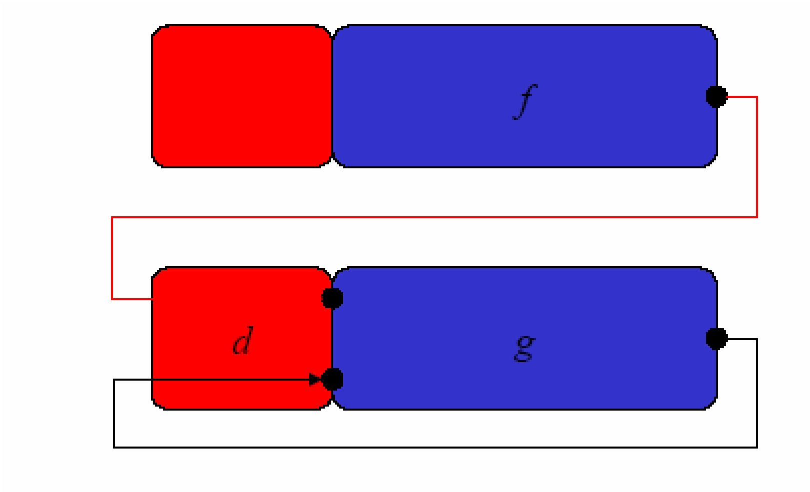


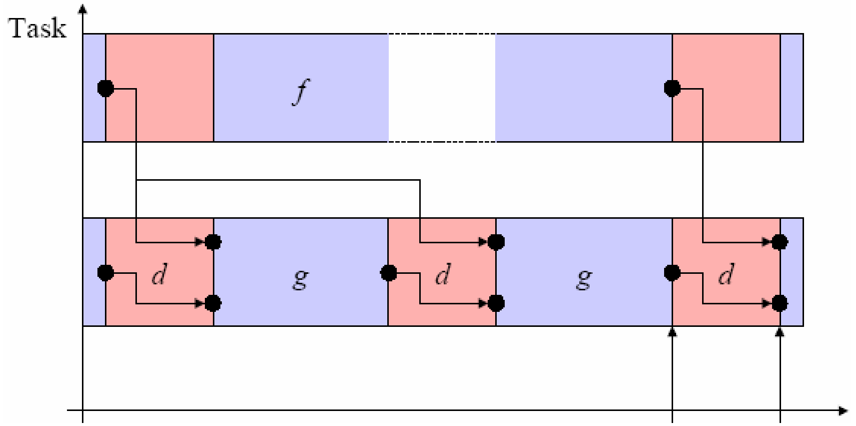
FLET – Task Execution Model



Universit□t Salzburg

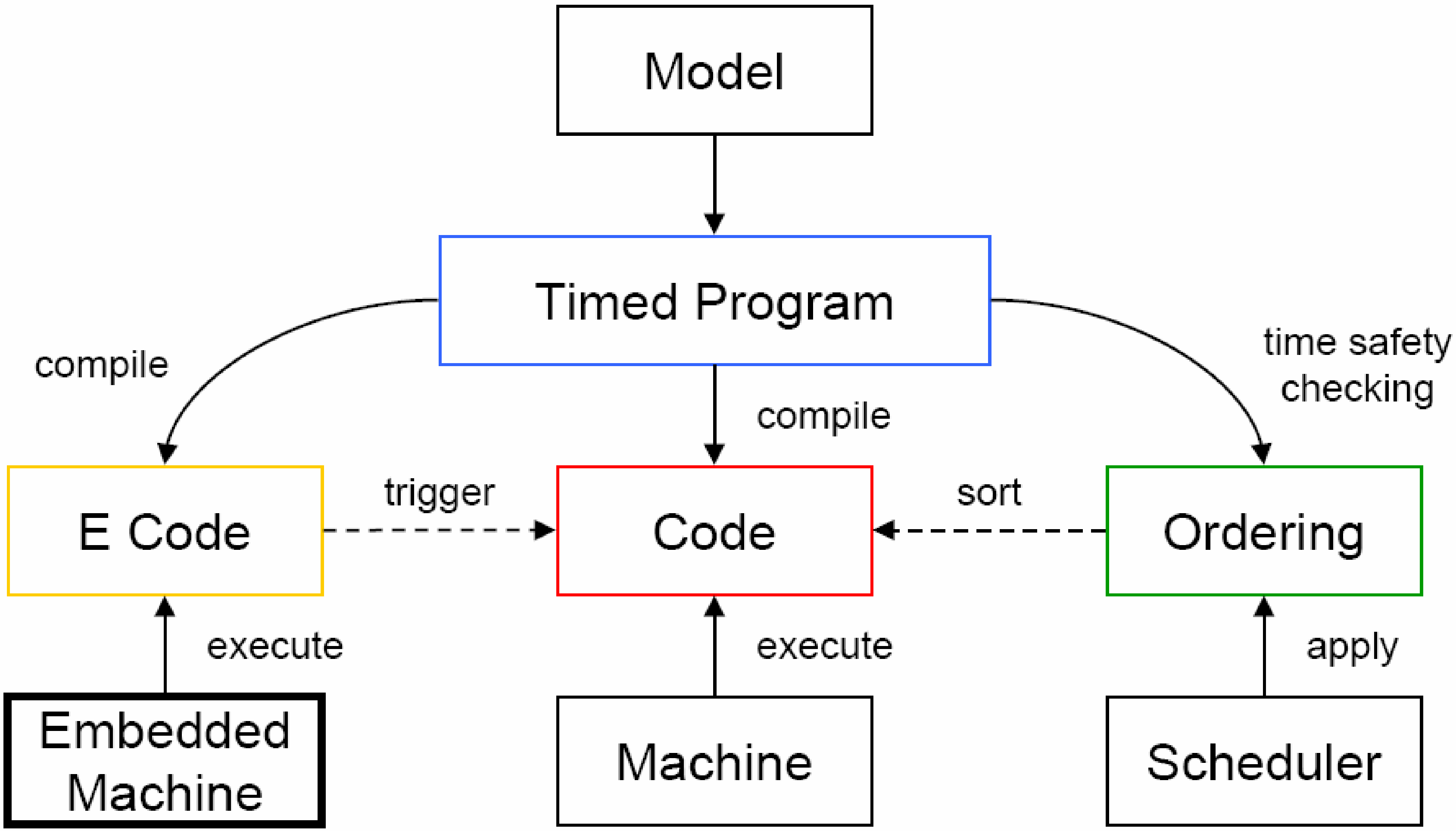
FLET – Communication Between Tasks





Universit□t Salzburg

Giotto Tool Chain



Universit□t Salzburg

E Code

E code

- Reactive/ timing code
- Manages the release times and deadlines of software tasks in reaction to environment events

E code is:

- *Portable*
- *Predictable*
- *Composable*

S Code

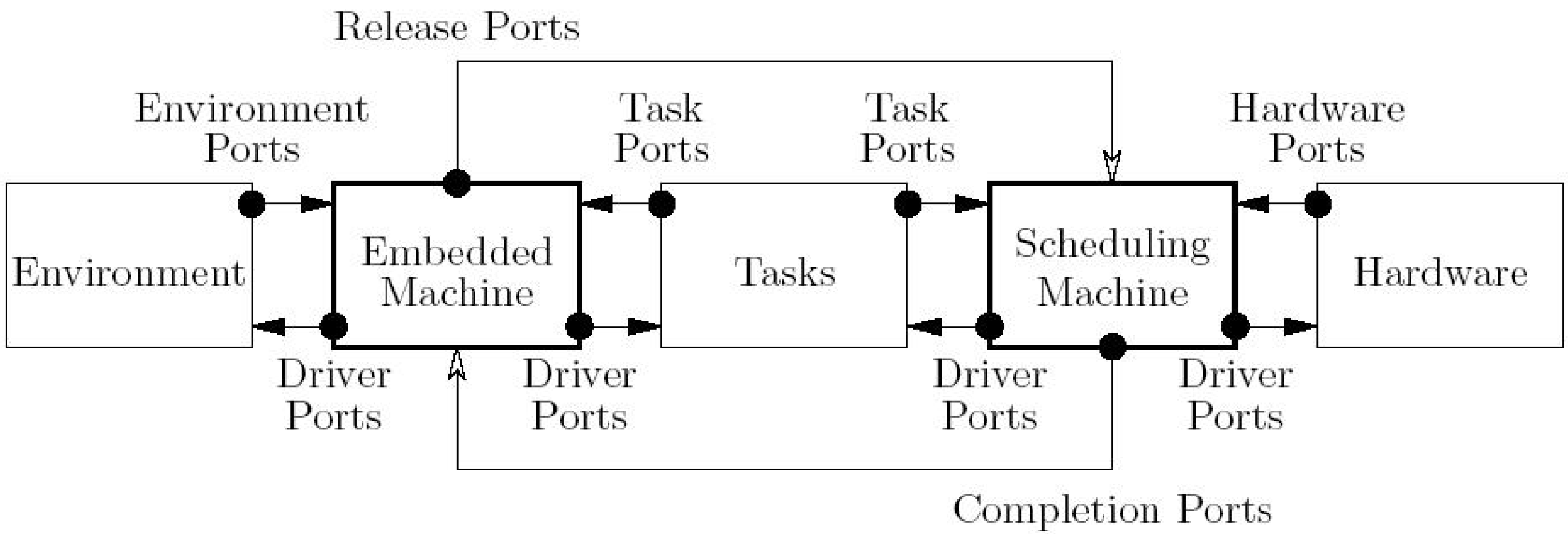
S code

- Proactive/ scheduling code
- Manages the execution of released tasks on available CPUs

S code is:

- *Universal* – any scheduling strategy
- *Verifiable* – fast time safety checking
- *Supports distribution*

The E Machine and S Machine



- Tasks are preemptive, user level code, with non-negligible WCETs
- Drivers are system level code, with negligible WCETs
- E Machine monitors input events through triggers
- S Machine monitors input events through timeouts

E Code Instructions

- Call(d)
- Schedule(t)
- Future(g,a)

The E machine maintains a queue of trigger bindings (g,a,s).

- If (c,a)
- Return

The execution is *time safe* if once released, a task *t* completes:

- Before any driver accessed a port of *t*
- Before *t* is released again

E Code Interpreter

while *ProgramCounter* $\neq \perp$ **do**
    i := *Instruction*(*ProgramCounter*)
    if call(*d*) = *i* **then**
        if driver *d* accesses a port of a task *t* that has been released but not completed
        then throw a time-safety exception **else** execute *d*
    else if schedule(*t*) = *i* **then**
        if task *t* has already been released but not yet completed
        then throw a time-safety exception **else** emit a signal on the release port of *t*
    else if future(*g, a*) = *i* **then**
        append the trigger binding (*g, a, s*) to *TriggerQueue*, where *s* is the current state of the input ports that occur in *g*
    end if
    ProgramCounter := *Next* (*ProgramCounter*)
end while

S Code Instructions

Transient instructions:

- Call(d)
- Fork(a)

The execution is *time sharing* if only one task is dispatched to the CPU.

The S machine maintains a set of thread instances (t, b, h, a, s).

Timed instructions:

- Dispatch(t,h,a) with 2 possible outcomes. S machine proceeds with:
  - The next instruction (t completes, or has not been released)
  - S code at address a (h expires)
- Idle(h)

S Code Interpreter

while *ProgramCounter* ≠⊥ **do**
    i := *Instruction*(*ProgramCounter*)
    ProgramCounter := *Next* (*ProgramCounter*)
    if call(*d*) = *i* **then**
        if driver *d* accesses a port of a task *t* that has been released but not completed
        then throw a time-safety exception **else** execute *d*
    else if dispatch(*t, h, a*) = *i* **then**
        if there is a thread instance in *ThreadSet* with a non-idle task **then**
          throw a time-sharing exception
        else
          insert the thread instance (*t, ProgramCounter, h, a,ReferenceTime*) into
          ThreadSet and set *ProgramCounter* to ⊥
        end if
    else if idle(*h*) = *i* **then**
        insert the thread instance (idle*, ⊥, h, ProgramCounter ,ReferenceTime*) into
        ThreadSet and set *ProgramCounter* to ⊥
    else if fork(*a*) = *i* **then**
        insert the thread instance (idle*, ⊥,* true*, a,s*) into *ThreadSet*, where *s* is the current
        value of the system clock
    end if
end while

Example - Simplified Flight Control

Giotto

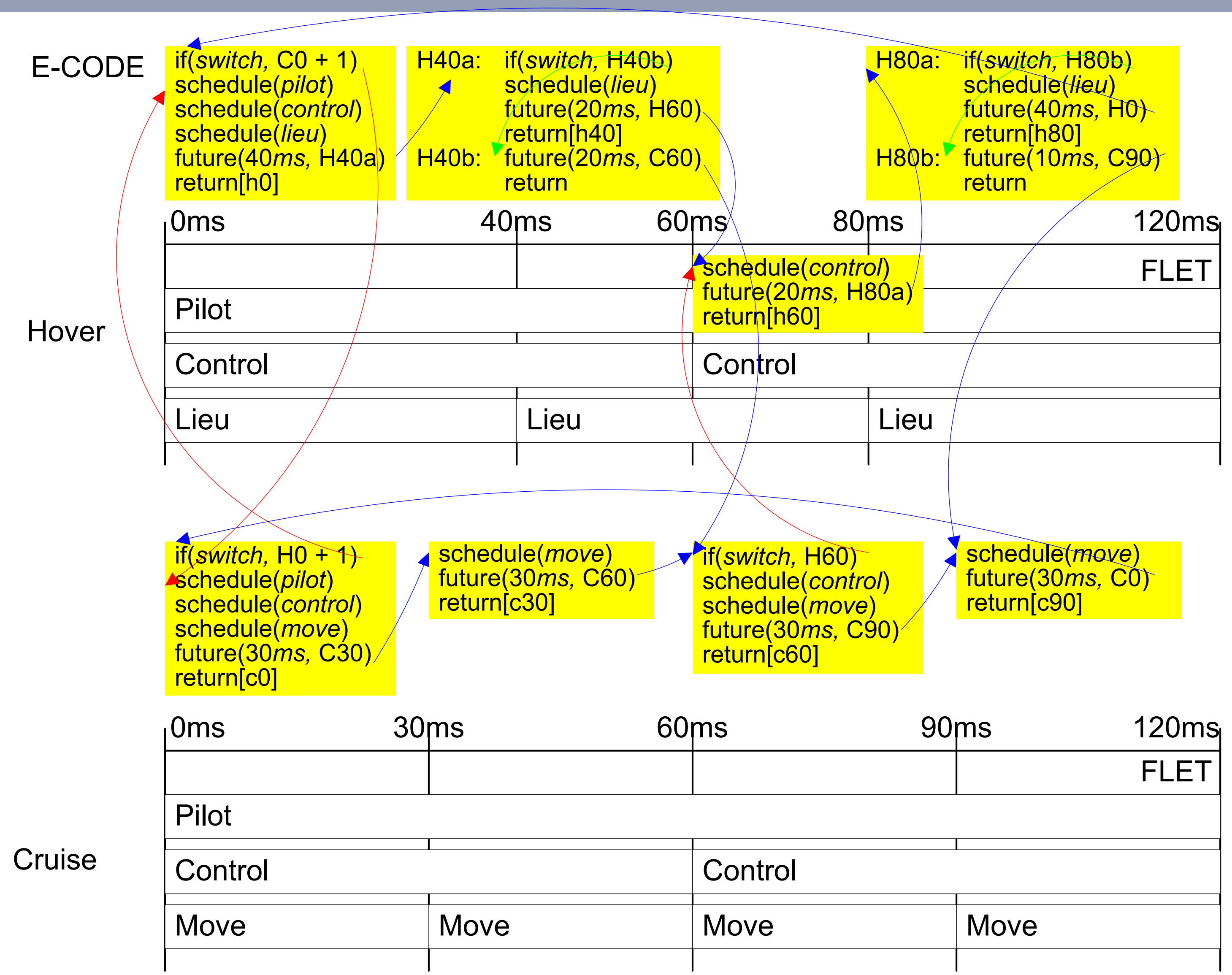
```
start hover {
mode hover() period 120ms {
  exitfreq 3 do cruise(switch);
  taskfreq 1 do pilot();
  taskfreq 2 do control();
  taskfreq 3 do lieu(); }
mode cruise() period 120ms {
  exitfreq 2 do hover(switch);
  taskfreq 1 do pilot();
  taskfreq 2 do control();
  taskfreq 4 do move(); }
```

E Code

```
H0: if(switch, C0 + 1)
      schedule(pilot)
      schedule(control)
      schedule(lieu)
      future(40ms, H40a)
      return[h0]
H40a: if(switch, H40b)
      schedule(lieu)
      future(20ms, H60)
      return[h40]
H40b: future(20ms, C60)
      return
H60: schedule(control)
      future(20ms, H80a)
      return[h60]
H80a: if(switch, H80b)
      schedule(lieu)
      future(40ms, H0)
      return[h80]
H80b: future(10ms, C90)
      return
```

```
C0: if(switch, H0+ 1)
      schedule(pilot)
      schedule(control)
      schedule(move)
      future(30ms, C30)
      return[c0]
C30: schedule(move)
      future(30ms, C60)
      return[c30]
C60: if(switch, H60)
      schedule(control)
      schedule(move)
      future(30ms, C90)
      return[c60]
C90: schedule(move)
      future(30ms, C0)
      return[c90]
```

Simplified Flight Control - E code



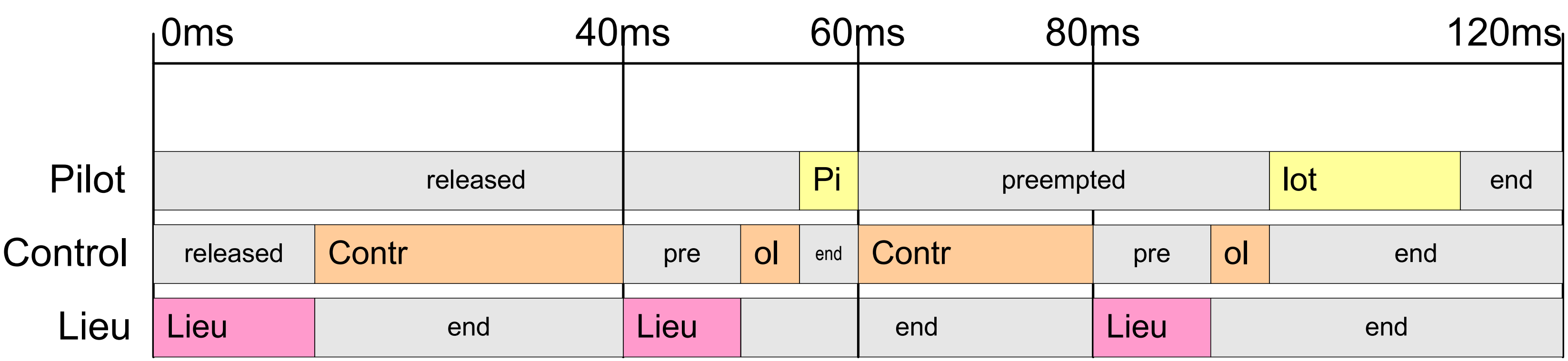
Universit□t Salzburg

Simplified Flight Control - S Code Variants - RM

<u>Rate Monotonic</u>

RM: dispatch(*lieu,* +4)
    dispatch(*control ,* +3)
    dispatch(*pilot,* +2)
    idle()
    fork(RM)
    return

Hover mode
(RM scheduling)



Universität Salzburg

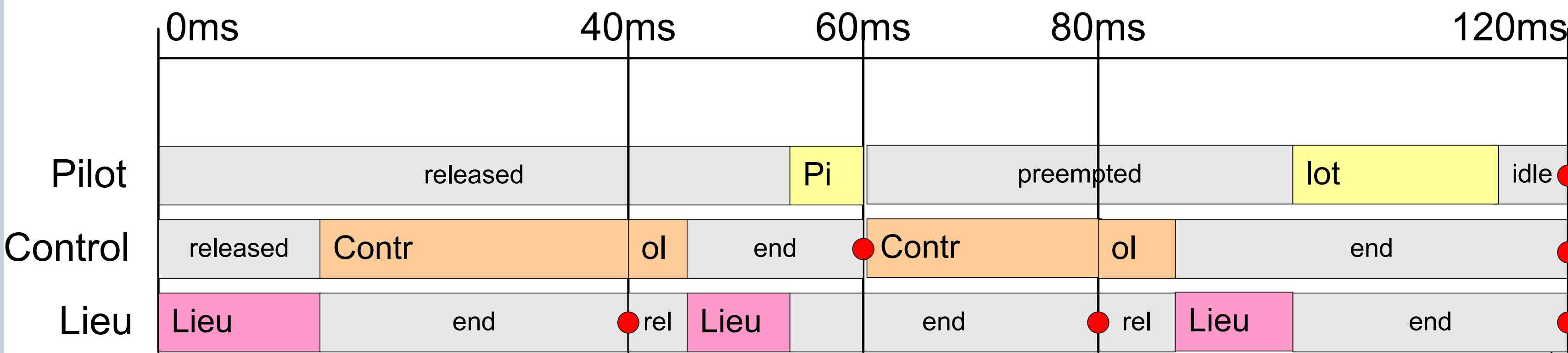
Simplified Flight Control - S Code Variants - EDF

Earliest-deadline-first

EDF0/60: dispatch(*lieu,* +4)
dispatch(*control* , +3)
dispatch(*pilot,* +2)
idle()
fork(EDF40/80)
return

EDF40/80: dispatch(*control,* +4)
dispatch(*lieu,* +3)
dispatch(*pilot,* +2)
idle()
fork(EDF0/60)
return

Hover mode
(EDF scheduling)



Universit□t Salzburg

Simplified Flight Control - S Code Variants

Non-Preemptive S Code for the cruise mode

Is time safe if $w(move) + w(control) \leq 30ms$ and $2 \cdot w(move) + w(pilot) \leq 60ms$

NP0: dispatch(*move*)
dispatch(*control*)
idle()
fork(NP30)
return

NP30: dispatch(*move*)
dispatch(*pilot,* NP60)
idle()
fork(NP60)
return

NP60: dispatch(*pilot*)
dispatch(*move*)
idle()
fork(NP90)
return

NP90: dispatch(*control*)
dispatch(*move*)
idle()
fork(NP0)
return

SCC Rules

- An *SCC program* is a pair (*E, S*) consisting of an E program that shares a set of tasks with an S program

Rules

- if there is an enabled thread instance that contains a completed task, then the S machine must handle that thread instance before the E machine handles any enabled triggers
- if there is an enabled trigger binding, then the E machine must handle that trigger binding before the S machine handles any expired timeouts.

Generating vs Checking SCC

- Use path-insensitive program analysis to check SCC, based on abstract semantics.
- Searching the state space is exponential.
- Checking the Time Safety is simpler for Giotto generated and simplified SCC.

- It is NP hard to generate non-preemptive or distributed schedules for Giotto programs.

- But simple Giotto generated SCC program can be checked in time linear with the size of the E code and frequency of events.

Time Safety Checking

- We can use classical Time Safety Checking for known algorithms.
- EDF schedulability of a single mode can be checked by solving a utilization equation.
- For multimode Giotto programs, if each mode in isolation is time-safe under EDF scheduling, then the whole program is time-safe under EDF
- It can be proved in linear size with the Giotto program that a SCC program corresponds to a certain algorithm.

Universit□t Salzburg

Optimality for schedulers

- A set of tasks is *schedulable* or *feasible* if all deadlines are met by some algorithm.

- A scheduling algorithm A is optimal among a category of scheduling algorithms if:

  Any systems that A cannot schedule cannot be scheduled by any other scheduling algorithms in the same category

Universit□t Salzburg

Optimal Scheduling Algorithms

Rate Monotonic Scheduling (RM)

- Priority = rate = 1/period
- Tasks with smaller periods have higher priorities
- Optimal among all fixed-priority algorithms

Earliest Deadline First (EDF)

- Priority = absolute deadline
- Tasks with earlier deadlines have higher priorities
- Optimal dynamic (varying priority) scheduling algorithm

Processor Utilization - Time Safety Check

- The *processor utilization* factor is the fraction of the processor time spent in the execution of the task set:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

C_i – computation time, WCET

T_i – period

Time safety check: for a given algorithm A, we can compute the *least upper bound* U_{lub} (A)

- If $U > 1$ no scheduling algorithms can guarantee the schedulability
- If $U \le U_{lub}(A)$ the tasks are schedulable by algorithm A

This condition *sufficient but not necessary*:

- If $U_{lub}(A) < U \le 1$, nothing can be said on the feasibility of the task set.

Processor Utilization -Time Safety Check

EDF Utilization Bound

- $U_{lub}=1$
- TSC: $U \le 1$
- EDF is optimal among all algorithms

RM Utilization Bound

- for *n* tasks: $U_{\text{lub}}(n) = n(2^{1/n} - 1)$
- $U_{lub}(2) = 0.828$
- $\lim_{n\to\infty} U_{\text{lub}}(n) = \ln 2 = 0.693$
- TSC: $U \le U_{lub}$
- How do we test schedulability for RM when $U_{lub} < U \le 1$?