Scheduling of Concurrent Reactive Objects for Embedded Real-Time Systems

Per Lindgren, Professor Embedded Systems
Luleå University of Technology
Sweden

...using Microsoft Powerpoint
EISLAB
Luleå University of Technology

Dept. of Computer Science, Electrical and Space Engineering

- Div. EISLAB 13 Faculty, 33 PhD students, 5 Research Areas
  - Embedded Systems
  - Electronics Design
  - Sensor Systems
  - Electro-Magnetic Compatibility (EMC)
  - Electronics Production

- Embedded Systems (2 seniors, 4 PhD students)
  - Focus on Embedded Real-Time Software
  - Modeling and programming language Timber
    www.timber-lang.org
Why yet another model?

Zillions of models & tools out there….

– Models/Frameworks
  • (Real-Time) UML Profiles
  • (Real-Time) CORBA, Ptolemy
  • TTA, Actors, etc.

– Languages
  • Domain Specific Languages, Erlang (telecom), etc.
  • Synchronous Languages (Giotto, SCADE), etc.
  • Actor languages (Axum, Erlang?), etc.

– Verification
  • Lots of methodologies and tools for WCET/Schedulability/Safety/Liveness, etc…BUT
Why yet another model?

Existing approaches, arguably…

– Difficult to use, model/program
– Timing & functionality often treated separately
– Hardware interaction often omitted
– Not suitable to resource constrained platforms
  CPU/Memory/Power
What we all would want…

Ideally a dish offering

– Independent jobs and concurrent interaction with the environment

– Simple, intuitive programming model

– Real-Time Model & Implementation

– Compile time verification w.r.t., Deadlock, Schedulability, Memory, etc.

– Feasible on Small Systems (resource constrained Memory/CPU)
The free lunch?

On today’s menu

- Independent jobs and concurrent interaction with the environment
  **Independency of jobs through implicit concurrency**

- Simple, intuitive programming model
  **Object/component model (OO oriented)**

- Real-Time Model & Implementation
  **Timing visible in model, preserved by Implementation**

- Compile time verification w.r.t., Deadlock, Schedulability, Memory, etc.
  **Automatic schedulability analyses based on Stack Resource Policy (SRP) exploiting kernel design (OH) and application WCETs**

- Feasible on Small Systems (resource constrained Memory/CPU)
  **Light weight kernel for ARM7, down to 8 bit controllers**
  **Efficient Deadline Monotonic (DM) scheduling under SRP**
Concurrent Reactive Framework

Event driven; natural property of Embedded Real-Time Systems
– Timed Constrained Reactions (messages)

Concurrent Reactive Objects (CRO)
– Abstracts system state and interactions
– CRO hierarchy through Concurrent Reactive Components (CRC)

Model, synthesize & analyze Embedded Real-Time Software

Interface the environment
– Hardware Interaction
– External libraries & Legacy code

[Paper submitted to RTiS 2011]
Concurrent Reactive Objects (CROs)

Stateful
- Complete State Encapsulation, built in!

Concurrent
- May execute in parallel, but
- Only one method per Object at each time

Communicates through messages
- SYNC(Object, Method, Arguments)
- ASYNC(Object, Method, Baseline, Deadline, Arguments)
Framework tool-suit

- XML based storage
- IDE/GUI for design, simulation, synthesis & verification
- Generates executable C-code + run-time system (kernel)
- Kernel design
  - Resources management under Stack Resource Policy (SRP)
  - Efficient implementation through Deadline Monotonic (DM) Scheduling
- Ongoing work on analysis (schedulability, memory, etc.)

[Paper accepted for publication at SIES 2011]
Example Control System (1)

- XML storage format *(class definitions)*
Example Control System (1)

- XML storage format (class definitions)
- Re-usable objects & components (from repository)
  (e.g., general purpose PID controller and data logger)
Example Control System (1)

- XML storage format (class definitions)
- Re-usable objects & components (from repository) (e.g., general purpose PID controller and data logger)
- Abstractions of environments (platforms, external code, etc.)
Example Control System (1)

- XML storage format (class definitions)
- Re-usable objects & components (from repository)
  (e.g., general purpose PID controller and data logger)
- Abstractions of environments (platforms, external code, etc.)
- Application specific objects & components
Example Control System (1)

- XML storage format (class definitions)
- Re-usable objects & components (from repository) (e.g., general purpose PID controller and data logger)
- Abstractions of environments (platforms, external code, etc.)
- Application specific objects & components
- A system component definition

![Diagram of control system components and interfaces]
Example Control System (2)
System Component Definition

The application is bound to the (chosen) environment.
Example Control System (3)  
Application Component Definition

- **Interfaces**: (Provided, Required)

![Diagram of control system components and interfaces]
Example Control System (3)

Application Component Definition

– Interfaces: (Provided, Required)
– Instances: (adapterinst, contrlinst, loginst)
Example Control System (3)

Application Component Definition

- Interfaces: (Provided, Required)
- Instances: (adapterinst, contrlinst, loginst)
- Communication (interconnections)
Example Control System (3)
Application Component Definition

- Interfaces:
  - Required (control_out, get_feedback)
  - Provided (init, setpoint)
- Should implement a generic PID controller
Example Control System (4)
Controller Class Definition

– Interface:
  • Required (control_out, get_feedback)
  • Provided (init, setpoint)

– Should implement a generic PID controller
  • State (setpoint, P, I, D)
  • Typical periodic implementation

Example Control System (4)
Controller Class Definition

– Interface:
  • Required (control_out, get_feeback)
  • Provided (init, setpoint)

– Should implement a generic PID controller
  • State (setpoint, P, I, D)
  • Typical periodic implementation

Example Control System (5)
Controller Class Definition

- Initialize state
- Deadline 1 MS (effects the jitter)
- After 10 MS

Diagram:
- Init block
- Control block
- Timeline from 0 to 22
Example Control System (5)
Controller Class Definition

- `init`
  - Initialize state

- `control`
  - After 10 MS
  - Deadline 1 MS
  - After 10 MS, effects controller response

Deadlines:
- Deadline 1 MS at 10 MS
- Deadline 1 MS at 19 MS
CRO controller class: interface subset of C types

```c
void setpoint(int s) {
  ...
  ...
}

void init() {
  ...
}

control_out
get_feedback
```
CRO controller class: state subset of C types

```c
int setp, p, i, d;
...

void init()
{
  ..
  ..
}

void setpoint(int s) { ... }
```

control_out

get_feedback
CRO controller class: bindings & code
Subset of C with local namespace per Object

```c
int setp, p, i, d;
...

void setpoint(int s) { setp = s; }

void init() {
    setp = p = i = d = 0;
    ASYNC(ctrl, MS(10), MS(1));
}

void ctrl() {
    int fb = SYNC(get_feedback);
    // compute c_out, p, i, d
    SYNC(control_out, c_out);
    ASYNC(ctrl, MS(10), MS(1));
}
```

```c
void contrl() {
    int fb = SYNC(get_feeback);
    // compute c_out, p, i, d
    SYNC(control_out, c_out);
    ASYNC(ctrl, MS(10), MS(1));
}
```
We choose a System (top component) to instantiate
Code synthesis

From top level
– Traverse and instantiate recursively
– Aggregate instance names, and make bindings
– Output is an object instance tree (with bound names)
– Emit C code (defines, states, methods)

```c
// local name bindings
#define get_feedback ..... 
#define control_out ..... 
#define process ..... 
// method implementation
int controller_process(int arg){
  int fb = SYNC(get_feedback, NO_ARG);
  // Controller state update etc
  SYNC(control_out, outval);
  ASYNC(process, MS(10), MS(1), NO_ARG);
}
#undef get_feedback
#undef control_out
#undef process
```
Lightweight Resource Management and Scheduling

Resource management under SRP
– Allows for (single) shared stack execution
  • Minimize memory requirement
– Allows for bound priority inversion
  • Max 2 “context switches” per job mitigates blocking
– Compile time deadlock test
  • Ensures deadlock free execution
– In combination with Deadline Monotonic (DM) Scheduling
  • Efficient kernel implementation
  • Schedulability test taking kernel OH into consideration
SRP and DM at a glance

- **Stack Resource Policy (SRP)**
  - A job may only claim resources in a hierarchical manner
  - A job must execute run-to-end (i.e., non-blocking)
  - A job is admitted (will start) *iff*
    - all resources needed is available
    - it has the highest priority of eligible jobs
    - it does not preempt any lower priority jobs that have claimed a higher priority resource

- **Scheduling**
  - Under Deadline Monotonic (DM) scheduling, priorities are assigned from *relative* job deadlines.
  - Earliest Deadline First (EDF) is an alternative assigning priorities dynamically from *absolute* job deadlines
  - **DM** is less flexible compared to EDF but allows for more efficient implementation onto commonplace hardware

Huge body of research on SRP and DM, e.g.
- [Liu/Leyland 1973 Original work on scheduling]
- [Baker 1990 Original work on SRP]
- [Jansen 2003/2004 EDFI, similar to ours while EDF & no OO]
Mapping from CRO to SRP-DM

A perfect match for our CRO model!!!!!

Informal mapping
– Object = resource
– Method = job
– SYNC = resource request
– ASYNC = job request
  • A job request is a sequence, (resource request → job → resource release)
  • An ASYNC with a baseline-offset is a postponed job request
  • For DM, the priority of the job is given from the job request deadline (for DM)
SRP Analysis (1)

Input:
– Object instance tree
– Entry points (the job requests from the environment)

Output:
– Resource ceilings (ceiling level for each object)
– Preemption levels (priorities) for interrupts

From the output together with the synthesized code we have all information needed to perform SRP based scheduling during run-time.

[Paper Submitted to WORDS2011]
SRP-Deadline Monotonic Kernel (1)

SRP-DM scheduling of external job requests
  – Efficient utilization of interrupt hardware
    System ceiling implemented in interrupt hardware
      • External interrupt requests can (if enough priority levels and interrupt sources be implemented completely in hardware)
  – Simple co-operative hardware/software kernel
    • Resource request is implemented by lowering the interrupt level (release by raising it again)
    • Shared sources are locally managed in the interrupt handler
    • Shared priority levels infers local priority inversion (limits scheduling) but also minimizes stack size

*Future work on memory optimization!*
SRP-DM scheduling of external internal job requests

- If one timer and interrupt source per internal job request priority, then scheduled by completely by hardware

- Else, if # timers insufficient
  - Queue implemented a CRO object, i.e., as a part of the system not as a part of the kernel!
  - Infers only local (bound) priority inversion
SRP Analysis (2)

From the SRP (1) analysis get;
– Resource ceilings (ceiling level for each object)
– Preemption levels (priorities) for interrupts
– Output is an object instance tree (with bound names)

and in addition we can obtain;
– job requests (ASYNC messages in the system)
– set of resource requests for each job request (SYNCs)

[Paper submitted to RTiS 2011]
WCET analysis and/or measurements

From WCET analysis on kernel primitives and CRO methods we get:

– WCET for each job including:
  • WCET for interrupt handling
  • WCET for SYNC operations (constant)
    (used to derive the blocking time for a resource)
  • WCET for ASYNC operations
    execution time for the ENQUEUE operation is bound by
    maximum number of messages in the system
    (blocking time for a timer resource)
Example 1a, priorities

Job preemption levels (priorities) are assigned according to deadlines:

\( j_1 \) has shorter deadline than \( j_2 \) (i.e., more urgent), thus is given a higher priority (0 is highest priority in hardware).

\[
\begin{align*}
 j_1 &= \text{ASYNC}(r_1, m_{11}, DL = 3) \\
 j_2 &= \text{ASYNC}(r_2, m_{21}, DL = 6)
\end{align*}
\]
Example 1b, stack

\[ j_1 = \text{ASYNC}(r_1, m_{11}, DL=3) \]

\[ j_2 = \text{ASYNC}(r_2, m_{21}, DL=6) \]

\( j_1 \) arrives (interrupt entry) and starts executing, system ceiling = 1, (previous value = infinity)
Example 1b, stack

\[ j_2 \text{ arrives and pends (in hardware), since system ceiling = 1} \]

\[ j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3) \]

\[ j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6) \]
Example 1b, stack

\[ j_1 \text{ ends (interrupt exit) and pending } j_2 \text{ is released, system ceiling = 2} \]

\[ j_1 = \text{ASYNC}(r_1, m_{11}, DL = 3) \]

\[ j_2 = \text{ASYNC}(r_2, m_{21}, DL = 6) \]
Example 1b, stack

\[ j_1 = \text{ASYNC}(r_1, m_{11}, DL = 3) \]

\[ j_2 = \text{ASYNC}(r_2, m_{21}, DL = 6) \]

\[ j_2 \text{ ends (interrupt exit)} \]

\[ \text{system ceiling} = \infty \]

A single execution stack suffices.
Example 1c, stack (preemption)

$j_2$ arrives and starts executing, system ceiling = 2

$j_1$ arrives and preempts $j_2$, system ceiling = 1

$j_1$ ends and resumes $j_2$, system ceiling = 2

A single execution stack suffices

$j_1 =$ ASYNC($r_1$, $m_{11}$, DL = 3)

$j_2 =$ ASYNC($r_2$, $m_{21}$, DL = 6)
Example 1d, inter-arrival

\[ j_1 \text{ arrives and preempts } j_2 \]

\[ j_1 \text{ arrives and preempts } j_2 \text{ again!!!} \]

\[ j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3) \]

\[ j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6) \]
Schedulability test under SRP-DM

The information given by SRP (1,2) and WCET analysis is sufficient to perform schedulability test.

Necessary condition:
– CPU Utilization < 1 (cheap to check)

Sufficient condition
– Each job meets its deadline (response time analysis)
Example 1, Utilization

Utilization = \frac{2}{3} + \frac{3}{6} > 1 \text{ FAIL!!}

\begin{align*}
R & \quad J & \quad \pi \\
\text{r}_1 & \quad j_1 & \quad 1 \\
\text{r}_2 & \quad j_2 & \quad 2
\end{align*}

\text{t}_1 = 3, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3)

\text{t}_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6)
Example 1, Response Time

\[
R \quad J \quad \pi
\]

- \( r_1 \) \( j_1 \) 1
- \( r_2 \) \( j_2 \) 2

Response time = 2 < 3 OK
Response time = 2*2+3 > 6 FAIL!!!

\[
t_1=3, j_1=\text{ASYNC}(r_1, m_{11}, \text{DL}=3)
\]

\[
t_2=6, j_2=\text{ASYNC}(r_2, m_{21}, \text{DL}=6)
\]
Example 2, Utilization

Utilization = $1/3 + 3/6 = 5/6 \leq 1$ Pass!!!

$t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3)$

$t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6)$
Example 2, Response Time

\[ R \quad J \quad \pi \]

\begin{align*}
  r_1 & \quad j_1 & = 1 \\
  r_2 & \quad j_2 & = 2 \\
\end{align*}

Response time = 2 < 3 OK

Response time = 1*2+3=5 < 6 Pass!!

\[ t_1=6, j_1=ASYNC(r_1, m_{11}, DL=3) \]

\[ t_2=6, j_2=ASYNC(r_2, m_{21}, DL=6) \]
Example 3, Utilization

Utilization = 1/3 + 4/6 = 6/6 <= 1 OK

$t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3)$

$t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6)$
Example 3, Response Time

Response time = 2 < 3 OK

Response time = 1*2+4=6 <= 6 Pass!

$R \quad J \quad \pi$

$r_1 \quad j_1 \quad 1$

$r_2 \quad j_2 \quad 2$

$t_1=6, j_1=\text{ASYNC}(r_1, m_{11}, \text{DL}=3)$

$t_2=6, j_2=\text{ASYNC}(r_2, m_{21}, \text{DL}=6)$
Example 4, Utilization

$j_2$ uses $r_1$ for 1t

Utilization = $1/3 + 4/6 = 6/6 =< 1$ OK

$t_1=6, j_1=\text{ASYNC}(r_1, m_{11}, \text{DL}=3)$

$t_2=6, j_2=\text{ASYNC}(r_2, m_{21}, \text{DL}=6)$
Example 4, Response Time

\( j_2 \) uses \( r_1 \) for 1t

\[
R \quad J \quad \pi
\]

\[
\begin{array}{ccc}
\text{r}_1 & \text{j}_1 & 1 \\
\text{r}_2 & \text{j}_2 & 2 \\
\end{array}
\]

Response time = 3 \( \leq \) 3 OK

Response time = 1*2+4=6 \( \leq \) 6 Pass!

\[
t_1=6, j_1=\text{ASYNC}(r_1, m_{11}, DL =3)
\]

\[
t_2=6, j_2=\text{ASYNC}(r_2, m_{21}, DL=6)
\]

\[
m_{\mathbf{11}} \quad m_{\mathbf{12}}
\]

\[
m_{\mathbf{21}} = \text{SYNC}(r_1, m_{12})
\]
Example 5, Utilization

$j_2$ uses $r_1$ for $2t$

Utilization = $1/3 + 4/6 = 6/6 = < 1$ OK

$t_1=6, j_1=$ASYNC($r_1, m_{11}, DL=3$)

$t_2=6, j_2=$ASYNC($r_2, m_{21}, DL=6$)
Example 5, Response Time

$j_2$ uses $r_1$ for 2t

$R \quad J \quad \pi$

$r_1 \quad j_1 \quad 1$

$r_2 \quad j_2 \quad 2$

Response time = 4 > 3 Fail!!

Response time = 1*2+4=6<= 6 Pass!

$t_1=6, j_1=ASYNC(r_1, m_{11}, \text{DL}=3)$

$t_2=6, j_2=ASYNC(r_2, m_{21}, \text{DL}=6)$
WCET (1)

Lots of readily available methods and tools for WCET. We (initially) take an empirical approach:
– For each external job request, measure WCET
Lots of readily available methods and tools for WCET. We initially take an empirical approach:

– For each external job request, measure WCET
– Add OH for scheduling (interrupt entry/exit)
WCET/Blocking Times (3)

\[ j_2 = \text{ASYNC}(r_2, m_{21}, \ldots) \]

\[ m_{21} = \text{SYNC}(r_1, m_{12}) \]

– For each external job request, measure WCET
For each external job request, measure WCET

Add OH for scheduling (interrupt entry/exit)

Add OH for scheduling (resource request, release)

\[ j_2 = \text{ASYNC}(r_2, m_{21}, \ldots) \]

\[ m_{21} = \text{SYNC}(r_1, m_{12}) \]

\[ m_{21} = \text{measured WCET} \]

\[ m_{12} = \text{measured WCET} \]

\[ m_{21} = \text{measured WCET} \]

\[ \text{interrupt entry} \]

\[ \text{resource request} \]

\[ \text{resource release} \]

\[ \text{interrupt exit} \]

\[ \text{total WCET} \]
WCET/Blocking Times (5)

\[ j_2 = \text{ASYNC}(r_2, m_{21}, \ldots) \]

- \( c_2 \) WCET for \( j_2 \)
- \( l_2 = \{(r_1, b_1)\} \) list of (requests, blocking times) derived from the (SYNCs) occurring in \( j_2 \)

This is ALL we need for schedulability test!
Or… what about internal job requests?

Assume a system with 2 entry points, $j_1$ and $j_2$

$t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL}=3)$

$t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL}=6)$

$m_{21} = \{ \ldots \text{ASYNC}(r_1, m_{12}); \ldots \}$
Internal job requests are extracted by the SPR(1) analysis

SRP(1) renders a system with $j_1$, $j_2$ and $j_3$. $j_3$, inherits deadline and inter-arrival from $j_2$.

$t_1=6, j_1=\text{ASYNC}(r_1, m_{11}, \text{DL} = 3)$
$t_2=6, j_2=\text{ASYNC}(r_2, m_{21}, \text{DL} = 6)$
$t_3=6, j_3=\text{ASYNC}(r_1, m_{12}, \text{DL} = 6)$

$m_{21} = \{ \ldots \text{ASYNC}(r_1, m_{12}); \ldots \}$
Example 6, Utilization

Utilization = $\frac{1}{3} + \frac{1}{6} + \frac{3}{6} = \frac{6}{6} = 1 \leq 1$ OK

$R \quad J \quad \pi$

- $r_1 \quad j_1 \quad 1$
- $r_1 \quad j_3 \quad 1$
- $r_2 \quad j_2 \quad 2$

$t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, DL=3)$

$t_3 = 6, j_3 = \text{ASYNC}(r_1, m_{12}, DL=6)$

$t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, DL=6)$

$m_{21} = \{ \ldots \text{ASYNC}(r_1, m_{12}); \ldots \}$
### Example 6, Response Time

<table>
<thead>
<tr>
<th>$R$</th>
<th>$J$</th>
<th>$\pi$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td>$j_1$</td>
<td>1</td>
</tr>
<tr>
<td>$r_1$</td>
<td>$j_3$</td>
<td>1</td>
</tr>
<tr>
<td>$r_2$</td>
<td>$j_2$</td>
<td>2</td>
</tr>
</tbody>
</table>

**Response time for each task:**

- $t_1=6, j_1=$ ASYNC($r_1, m_{11}$, DL =3)
- $t_3=6, j_3=$ ASYNC($r_1, m_{12}$, DL =6)
- $t_2=6, j_2=$ ASYNC($r_2, m_{21}$, DL =6)

**Response time calculation:**

- $t_1 = 1 \times 2 + 1 \times 1 + 3 = 6 \leq 6$ OK
- $t_3 = 1 \times 2 + 1 \times 1 + 3 = 6 \leq 6$ OK
- $t_2 = 3 \leq 3$ OK

**Diagrams:**

- Task $r_1$ with jobs $j_1$ and $j_3$.
- Task $r_2$ with jobs $j_1$ and $j_3$.

**Message:** $m_{21} = \{ \ldots$ ASYNC($r_1, m_{12}$); $\ldots \}$
Or... what about postponed internal job requests?

Assume a system with 2 entry points, \( j_1 \) and \( j_2 \)

\[
\begin{array}{c|c|c}
R & J & \pi \\
\hline
r_1 & j_1 & 1 \\
\hline
r_2 & j_2 & 2 \\
\end{array}
\]

\[ t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL}=3) \]

\[ t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL}=6) \]

\[ m_{21} = \{ \text{ASYNC}(r_1, m_{12}, \text{BL}=2, \text{DL}=6); \ldots \} \]
SRP(1) renders a system with \( j_1, j_2 \) and \( j_3 \)
\( j_3 \), inherits inter-arrival from \( j_2 \) but has explicit BL & DL

\[
\begin{array}{ccc}
R & J & \pi \\
\hline
r_1 & j_1 & 1 \\
r_1 & j_3 & 1 \\
r_2 & j_2 & 2 \\
\end{array}
\]

\( t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3) \)
\( t_3 = 6, j_3 = \text{ASYNC}(r_1, m_{12}, \text{DL} = 6) \)
\( t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6) \)

\[
m_{21} = \{ \\
\quad \text{ASYNC}(r_1, m_{12}, \text{BL= 2, DL = 6}) \\
\}
\]

Or... what about postponed internal job requests?
Example 7, Utilization

Utilization = \( \frac{1}{3} + \frac{1}{6} + \frac{3}{6} = \frac{6}{6} = 1 \) OK

\[
R \quad J \quad \pi \\
r_1 \quad j_1 \quad 1 \\
r_1 \quad j_3 \quad 1 \\
r_2 \quad j_2 \quad 2
\]

\( t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3) \)
\( t_3 = 6, j_3 = \text{ASYNC}(r_1, m_{12}, \text{DL} = 6) \)
\( t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6) \)

\( m_{21} = \{ \ldots \text{ASYNC}(r_1, m_{12}, \text{BL} = 2, \text{DL} = 6); \ldots \} \)
Example 7, Response Time

\[ R \quad J \quad \pi \]

- \( r_1 \) \( j_1 \) 1
- \( r_1 \) \( j_3 \) 1
- \( r_2 \) \( j_2 \) 2

\[ t_1 = 6, j_1 = \text{ASYNC}(r_1, m_{11}, \text{DL} = 3) \]
\[ t_3 = 6, j_3 = \text{ASYNC}(r_1, m_{12}, \text{DL} = 6) \]
\[ t_2 = 6, j_2 = \text{ASYNC}(r_2, m_{21}, \text{DL} = 6) \]

\[ m_{21} = \{ \ldots \text{ASYNC}(r_1, m_{12}, \text{BL} = 2, \text{DL} = 6); \ldots \} \]

Response time = 3 \( \leq \) 3 OK
Response time = 1*2+1=3 \( < \) 6 OK
Response time = 1*2+1*1+3=6 \( \leq \) 6 OK
Hardware based scheduling also for internal ASYNCs with just one timer

- Timer queue(s) is/are treated as part of the application
- **Kernel** boils down to primitives **4 simple primitives**:  
  - SYNC pre-/postlude + enqueue/dequeue postludes

(a) $t_1 = 60, j_1 = \text{ASYNC}(o_1, m_{11}, 0, 40)$  
$m_{11} = \{ \ldots M_1 = \text{ASYNC}(o_2, m_{21}); \ldots \}$

(b) $i_2 = 60, i_j = \text{ASYNC}(o_2, m_{21}, 0, 40)$  
$m_{21} = \{ \ldots \text{SYNC}(o_1, m_{12}); \ldots \}$

[Details in paper submitted to RTiS 2011]
Example 8, ASYNCs

(a) \( t_1 = 60, j_1 = \text{ASYNC}(o_1, m_{11}, 0, 40) \)
\( m_{11} = \{ \ldots M_1 = \text{ASYNC}(o_2, m_{21}); \ldots \} \)

(b) \( i_t_2 = 60, i_j_2 = \text{ASYNC}(o_2, m_{21}, 0, 40) \)
\( m_{21} = \{ \ldots \text{SYNC}(o_1, m_{12}); \ldots \} \)

[Details in paper submitted to RTiS 2011]
Did we get the free lunch?

What was served

- Independency of jobs through implicit concurrency
  Independent jobs and concurrent interaction with the environment
  Objects operate Concurrently with full state-encapsulation

- Object/component model (OO oriented)
  Simple, intuitive programming model (nothing “unexpected” there)

- Timing visible in model, preserved by Implementation
  Real-Time Model & Implementation

- Automatic schedulability analyses based on Stack Resource Policy
  (SRP) exploiting kernel design (OH) and application WCETs
  Compile time verification w.r.t., Deadlock, Schedulability, Memory, etc.

- Light weight kernel for ARM7, down to 8 bit controllers
  Efficient Deadline Monotonic (DM) scheduling under SRP
  Feasible on Small Systems (resource constrained Memory/CPU)
Ongoing & Future Work

– Ongoing
  • Finalize Kernel for ARM7 & AVR-5 (8 bit)
  • WCET tool integration (for now semi-manual)
  • Integrate Schedulability Analysis

– Future opportunities & open problems (tentative…)
  • Explore offset based scheduling (6 months)
  • Develop SRP based memory analysis (6 months), and multi dimensional scheduling (power, communication, etc.) (1-2 years)
  • Formalize model and kernel (1 year)
    – Proof of safety, liveness, purpose (e.g., that it actually implements SRP)
  • Target Timber compiler to CRO model (1-2 years?)
    – Static object structure, static memory
  • Debugger integrated in the IDE offering (1-2 years?)
    – Co-simulation with e.g., SIMULINK
    – Low level, GDB integration for simulation & target systems
    – High level, visualizing scheduling & communication