2. Decomposition Approach

Contents:

2.1 Design Levels
2.2 The Task Model
2.3 Schedules
An Example
2.1 Design Levels

There is no closed design theory for r-t systems available.

Design is an iterative process of assumptions and verifications.

The result of this process can be represented at different refinement levels.

We distinguish:

- global level
- path level
- activity level
- process level
- task level
2.1.1 Global level

Coarse view on the activities that should be performed by the r-t system
5-stage structure:

- **sensors**: devices that inform about the status of the environment
- **sensor elements**: software modules that perform some pre-processing on the measured data
- **evaluation and decision**: validity of pre-processed data is checked and put into relation with historical data
  
  requirements for changes in the environment are computed database, knowledge base
- **actor elements**: required changes are converted into control data for the actors
- **actors**: devices to change the behavior of the environment
Paths of activities:
2.1.2 Path level

At this level, several so-called paths of activities are identified.
A path is a set of activities with given timing conditions for their execution, and where the execution order is restricted by precedences.

- **A path can be**
  - periodic with some given period $t_j$
  - aperiodic, sporadic (asynchronous)

- **can be triggered by**
  - certain constellation of sensed (measured) data,
  - resulting data of other paths
  - the human observer

- **can have a deadline, due date**
  - with an aperiodic or sporadic (asynchronous) behavior,
  - and as well with deadlines or due dates
An example path:

Sensor outputs

preprocessing

postprocessing

The so-called end-to-end scheduling deals with the scheduling of paths

Gerber et al. (1995)
2.1.3 Activity level

Activities (or jobs) ... are programs
Each activity consists of preemptable and non-preemptable segments,
each segment requires certain resources during execution

2.1.4 Process level

Process ... is an activity being in execution
Several types of processes are distinguished:

Periodic process: is executed repeatedly, once in a fixed period of time
A typical example is to read sensor data and update the current state of
internal variables and outputs

Asynchronous process: is executed once
responds to internal or external events
2.1.5 Task level

Task ... is part of an activity (called segment) that has constant processing conditions preemptable or non-preemptable, precedences.

A task may require one or more resources during its execution.

A real-time system consists generally of a huge number of tasks, with different conditions for their processing.

A general requirement is that each task is executed on a single processor, and terminates in finite time.
2.2 The Task Model

A task is defined by its task type \( (\tau_1, \ldots, \tau_k) \)

- segment of an activity i.e. a piece of program code
- Example: graphical display of a temperature

**task** \( (T_j) \):

- execution (instanciation) of an activity segment; has own data
- example: measurement and processing of data from different instruments

**periodic task**

- task \( T_j \) is executed periodically with given periodicity

- \( i \)-th execution of task \( T_j \):
  \[ T_j^i \]
  - example: repeated temperature measurement (e.g. every second)

**aperiodic** (statically) or **asynchronous** (dynamically) tasks:

- are processed only once
Characterization of tasks

- Processing time (execution time, $p_j$)

  depends on

  = speed of processor
  = processor periphery (memory size, channel capacity, ...)
  = availability of additional resources required during task execution
  = display unit, communication channel,
  = delays caused by the execution of other more urgent tasks

  source for the running times is a careful program analysis
  upper execution time bounds

- Ready time (release time, $r_j$)

  earliest time a task $T_j$ can be started on some processor
- Deadline \((d_j)\)
  
  latest allowed completion time of process \(T_j\)
  
  In case that both, release time and deadline are specified: \(p_j \leq d_j - r_j\)

- Periodicity of tasks (length of period \(p_j\))
  
  ready time in the \(i\)-th interval: \(r^i_j := (i - 1)p_j + r^1_j\) \((i = 2, 3, ...)\)
  
  deadline in the \(i\)-th interval: \(d^i_j := ip_j + r^1_j = r^{i+1}_j\)
  
  usually \(d^i_j \leq r^{i+1}_j\) required

  feasibility condition: \(p_j \leq p_j\) or \(p_j \leq d^i_j - r^i_j\)

- Asynchronous tasks
  
  occur at previously unknown times at system run-time,
  
  due to an external or internal event

  Assumption: processing time and deadline are known at arrival time
  
  An asynchronous task may have to be processed at a certain frequency
− Arrival time of a task: the time when the system gets knowledge that the task has to be processed
  usually event-triggered tasks, e.g. asynchronous tasks
− Priority
  expresses the criticality of a task
  higher critical tasks must be processed prior to less critical tasks.
− Preemptability
  Tasks can be interrupted by other tasks that have a higher priority.
  Non-preemptive execution: a task, once started, cannot be interrupted
  Preemptive execution: the task execution can be interrupted.
− Resources (besides processors, and fixed during the execution of a task)
  Non-shared hardware devices except CPU's;
  non-shared software components and data
  I/O channels, communication media, or data, may restrict the concurrent execution of tasks.
Notation: set of $s$ resource types, $R = \{R_1, \ldots, R_s\}$, each with some maximal amount normalized to $1$ (= 100%), and tasks can acquire fractions thereof with a given granularity.

Resource requirement vector of a task: $R(T_j) = (r_{j1}, \ldots, r_{js}) \in [0, 1]^s$

- Precedence relation $T_i \prec T_j$

In this case, $T_j$ cannot be started before $T_i$ has been completed

Determining these parameters is essential in real-time scheduling

Though determining these parameters is essential in real-time scheduling, exact knowledge of these cannot generally be assumed.

Especially the occurrence of asynchronous tasks cannot be predicted because this depends on particular constellations of data.
<table>
<thead>
<tr>
<th></th>
<th>task set</th>
<th>task parameters</th>
<th>arrival times</th>
</tr>
</thead>
<tbody>
<tr>
<td>static scheduling</td>
<td>known</td>
<td>known</td>
<td>known</td>
</tr>
<tr>
<td>dynamic scheduling</td>
<td>known</td>
<td>known</td>
<td>unknown</td>
</tr>
</tbody>
</table>

Of influence are further the processing capabilities of the processors:

Processors may be

– identical, so that the task processing times are the same on each processor,
– uniform, if they have different speeds
– heterogeneous, if processors require different times for each task
2.3 Schedules

A schedule specifies completely the task allocations to the processors
can be constructed for the set of non-asynchronous tasks
usually: periodic tasks

Important parameters in schedules:

- Task start time \((s_j)\)
  time at which process \(T_j\) starts

- Task completion time \((c_j)\)
  time at which process \(T_j\) completes

\[ s_j + p_j = c_j \] (non-preemptive process)
\[ s_j + p_j \leq c_j \] (preemptive process)
Three phases in real-time scheduling are distinguished:

(i) Schedulability checks

- Feasibility check
  Given a set of tasks with hard timing constraints, is there a schedule where each task meets its deadline?
  
- Optimization problem:
  
  = minimize number of late tasks
  
  = minimize number of arriving tasks that cannot be processed on time (rejected tasks)
(ii) Schedule construction

**Feasible schedule:**

- each processor executes at most one task at a time,
- each task $T_j$ is assigned to exactly one processor for $p_j$ units of time
  (non-preemptive case),
- or each task $T_j$ is assigned to processors during non-overlapping time intervals of total length $p_j$  (preemptive case),
- all timing restrictions, resource requirements and precedences are obeyed.

**Explicit schedule:** complete and detailed specification (off-line scheduling)

- start time and processor are specified for each task
- Explicit specification can also be represented graphically by a Gantt chart
- Explicit schedules are used in pre-run-time scheduling.

**Implicit schedule:** specification of a scheduling rule (on-line scheduling)

- e.g. by means of priorities: tasks of higher priority are given preference.
(iii) Schedule execution (run-time system)

- non-preemptive scheduling: context switch at the end of each task only

- preemptive scheduling: context switches between tasks are initiated either by an external/internal event or by the operating system system clock, changing priorities, etc.

Context switches cause delays:

    total delays are unpredictable if the number of context switches is unknown

The problems we are confronted with are manifold:

- How many processors are required, and how should the code be distributed

- Find a feasible and safe schedule, in which also communication delays between pairs of dependent tasks are taken into account

- Which strategy should the run-time system follow in order to ensure correct behavior
An Example: A simple system with two paths $A'$ and $A''$
nodes represent activities (for simplicity: each activity has one task)
edges represent conditions for the execution order of the activities

Assumptions on the paths and activities:
- $A'$ is periodic with period 20; the first interval starts at time 0
- under certain conditions $A'$ triggers the sporadic path $A''$
- after being triggered, $A''$ is completed within 18 time units

The activities (tasks) may have the following properties:

<table>
<thead>
<tr>
<th>$p_j$</th>
<th>T'1</th>
<th>T'2</th>
<th>T'3</th>
<th>T'4</th>
<th>T'5</th>
<th>T'6</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

| preempt | yes | yes | yes | no  | yes | yes |

| $R(T'_j)$ | | | | | | |

<table>
<thead>
<tr>
<th>program code</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C3</th>
</tr>
</thead>
</table>

| path $A''$: |
|-----------------|-----|-----|-----|-----|-----|-----|-----|
| $T''_1$ | $T''_2$ | $T''_3$ | $T''_4$ | $T''_5$ | $T''_6$ | $T''_7$ |
| 2    | 1    | 4    | 3    | 5    | 4    | 2    |

| preempt | yes | no | no | no | yes | yes | no |

| $R(T''_j)$ | | | | | | |

| program code | C2 | C4 | C6 | C7 | C3 | C5 | C8 |

Assumptions on the processor environment:
Three identical processors: \( P_1, P_2, P_3 \)

each with local memory which carries the code executed locally

Installed code:

- \( P_1 \) has the code of \( C_1, C_2, C_4, C_6, C_8 \)
- \( P_2 \) has the code of \( C_1, C_3, C_4, C_5, C_6 \)
- \( P_3 \) has the code of \( C_2, C_4, C_5, C_7, C_8 \)

**Question:** Is it possible to schedule the paths properly and safely?

Assume communication delays between pairs of dependent tasks
e.g. unit time delay, if dependent tasks are processed on different processors

**Is it then still possible to get a feasible solution?**
Summary of Chapter 2

Decomposition approach allows to distinguish different levels of design
  global level, path level, activity level, task level

Tasks are characterized by properties
  periodicity, processing time, release time, deadline, priority,
  preemption, precedence constraints

Task properties are the basis for the construction of schedules for the
activities and paths, and hence for the complete system