
Contents:

3.1 Requirement Engineering
3.2 Tools, Specification and Design Techniques
3.3 Design Methods
3.4 Real-Time Languages
Only few structured design methods directly relate to common hard real-time activities, such as periodic or sporadic processes.

**Burns and Wellings (1994):** real-time design method must provide

- the explicit recognition of the types of activities/objects
  cyclic and sporadic activities

- the integration of appropriate scheduling paradigms with the design process

- the explicit definition of the application timing requirements for each object

- the explicit definition of the application reliability requirements for each object

- the support for different modes of operation
  e.g. takeoff, cruising, and landing for an aircraft
• the explicit definition and use of resource control objects

• the decomposition to a software architecture that is amenable to processor allocation, schedulability and timing analysis

• facilities and tools to allow the schedulability analysis to influence the design as early as possible in the overall design process

• tools to restrict the use of the implementation language so that worst case execution time analysis can be carried out

• tools to perform the worst-case execution time and schedulability analysis

For hard real-time systems the significant disadvantage is that timing problems will not be recognized before testing

⇒ software engineering problem
3.1 Requirement Engineering

3.1.1 Structured Analysis

Aim

– structuring synthetic systems
– calculating future behavior
  this provides a planning basis for predictions

Stepwise refinement ...

its components and the relations between them

... similar to SADT, based on graphics: diagrams
**Modeling functions and data**

each diagram: represents a limited part of the system, its functions, connections between functions, data flows

**Modeling control:** State transition diagrams

specify the conditions when the system changes from one state to another control flow diagrams

**Hierarchy of diagrams** general aspects --- detailed aspects

Upper level signals: external signals enter the r-t system

Signals created by the r-t system are sent to the environment

Lower level signals: between parts of the r-t system
3.1.2 Example [case study]: Filling of paint cans

Real-time system (grey, filling control system) and the environment (technical system)
Environment: has five technical processes

• operator: supervises the process
• filling station: gets signal to start/stop filling
• tara scales: gives the weight of the empty can
• gross scales: sends the weight of the filled can
• separator gate: the filled can leaves the filling system

Real-time system:

➢ receives status data from the technical process,
  scales ready signals

➢ sends control signals to the technical processes
  filling change signal, switch signal
Filling control system: First level decomposition

Four independently working processes:

- **Left two processes** triggered by scales ready
- **Operator dialog** triggered by commands
- **Monitor filling** triggered whenever a new net weight arrives

**Get tara weight**
- tara weight
- mean value
- gross weight
- scales ready

**Calculate net-weight**
- planned filling data
- mean value
- scales ready

**Operator dialog**
- commands
- planned filling data
- start filling
- continue filling

**Monitor filling**
- net weight
- filling change signal
- switch signal
- done
- alarm

**Statistics**
- gross weight
- net weight

**Filling data**
Process monitor filling: state transition diagram and main control loops:

- **states**: drawn in rectangles
- **transition between states**: arrows

**lower case labels**: conditions and events necessary to cause a state transition

**UPPER CASE LABELS**: signals generated when the transition takes place
3.2 Tools, Specification and Design Techniques

Real-time development methods have to deal with temporal correctness and complexity.

This requires

- precise definition
- software engineering concepts
  modularization, abstraction, information hiding, and others
- support of a rigorous approach during the entire life-cycle
  requirements engineering, design, implementation, testing, maintenance.
3.2.1 Petri Nets as System Models

A Petri net consists of nodes, edges, markings, and is described as a 4-tuple \( N = (S, T; F, M) \)

Two types of nodes:
- finite set \( S = \{S_1, S_2, \ldots\} \) of places (or positions)
- finite set \( T = \{T_1, T_2, \ldots\} \) of transitions

places and transitions represent parts of the system

Oriented edges: set \( F \subseteq (S \times T) \cup (T \times S) \)

\( F \) describes a "flow relation" between places and transitions

Places can be marked with tokens
- describe the states of parts of the system

States = values specifying cases, situations, etc.

Finite state set \( M = \{M_1, M_2, \ldots\} \)

\( M_i \in \{0, 1\} \) is the state of place \( S_i \)
Principles

- **Decomposition principle**
  Parts (positions, transitions) can be subdivided into subsets

  \[ s \in S \rightarrow N_s = (S_s, T_s ; F_s, M_s) \]

  \[ t \in T \rightarrow N_t = (S_t, T_t ; F_t, M_t) \]

- **Causality principle**
  Changes of part states are triggered by the states of other parts:

  
  | if       | all places leading to a transition are marked, |
  |          | and all places leaving the transition are free, |
  | then     | the tokens are removed from the first places, |
  |          | and the places following the transition are marked |

  "the transition fires"

- **Temporal principle**
  for modeling temporal sequences of operations: "timed nets"
Special situations in Petri nets have practical counterparts:

- safety (no place in the net has more than one token)
  corresponds to the condition that each production cell has at most one work piece

- conflict (a token has two possibilities to leave a position)
  corresponds to two ways of reaction in a hazardous situation

- reachability (a place can never be marked)
  corresponds to a useless device

- lifelock: its absence ensures continuous functioning of the system

- deadlock: corresponds to a blocked state, due to an erroneous design
Other Net Types and Their Practical Reference

Colored nets
Individualized "colored" tokens
different processes can run concurrently on the same net

Timed (determined) nets
The time of a token waiting on a place is restricted or depends on the switching behavior of a transition
for modeling time-out conditions, watch-dog processes

Timed stochastic nets
The switching of a transition behaves in a time-stochastic manner
Firing delay behaves according to a probability distribution function (e.g. Poisson distribution)
for modeling interrupt-controlled requirements of a process-controlling computer system
Example:

From: A Train Control System Case Study in Model-Based Real Time System Design, A. Zimmermann and G. Hommel, T. U. Berlin, WPDRTS '03
There are open questions regarding the power of (timed) Petri nets

- There is no powerful hierarchical abstraction concept
  in which dynamic properties on a lower layer are formal refinements of specific properties on a higher layer
  A formal behavior analysis of real-size system design project is practically intractable

- Wedde (2000): Petri nets suffer from a conceptual point of view:
  it seems to be impossible to model soft real-time situations

\[ <\text{deadline}> \quad t' \quad t \quad <\text{time interval}> \]

ERROR
3.2.2 Petri Net Based Specification of Concurrent Processes


Starting point: trace-based specification of a concurrent system

Trace: is a sequence of states of a process

Method: a stochastic Petri net model is built in a modular and systematic way to:

- specify each process separately
- model interactions of processes in terms of feasible sequences of events

Overview:

- Step 1: Informal system description
- Step 2: Definition of the trace-based specifications
- Step 3: Domain transformation
- Step 4: Integration
- Step 5: Adding external assumptions
- Step 6: Temporization
♦ Step 1: Informal system description

An informal description is derived from the system requirements documents in natural language, must contain all information to identify the communication sequential processes (CSP) and their traces (in the next step).

Designer has to specify a list of components, and for each component (process):

(a) a list of events in which the process can be involved

(b) a list of all possible states

state transition:

events that cause transition into a given state have to be identified
(c) a list of assumptions

   internal assumptions: precedence or logical constraints between process states

   external assumptions: concern events that are involved in more than one process

(d) temporal specifications: consist of a list of time consuming events and their related temporal parameters
Step 2: Definition of the trace-based specifications

A formal CSP specification is derived from the informal description

− Each component is identified as a process and is assigned a name \((P)\)

− To each event is assigned a name \((e)\)

  Set of events of a process: "alphabet" of the process, \(\alpha P\)

− Formal definition of states: Each state gets a name \((S)\)

− Traces: trace \(t\) := sequence of events

\[
\langle \rangle := \text{empty trace} \quad \#t := \text{length of } t \\
\langle e_1 \rangle := \text{trace containing only one event} \quad t' := \text{trace deprived of its last event} \\
\langle e_1, e_2 \rangle := \text{trace containing a sequence of two events (ordered)} \quad t'' := \text{last event of } t \\
\{\langle e_1, e_2 \rangle\}^* := \text{infinite set of traces:} \\
\{\langle \rangle, \langle e_1 \rangle, \langle e_1, e_2 \rangle, \langle e_1, e_2, e_1 \rangle, \langle e_1, e_2, e_1, e_2 \rangle, \ldots\} \quad \text{traces}(P) := \text{set of all possible traces of } P
\]
Furthermore:

\[ E_S(P) := \text{subset of events of } \alpha P \text{ that cause a transition of } P \text{ into state } S \]

\[ L_S(P) := \text{set of traces } t \text{ whose last event } t'' \text{ belongs to } E_S(P) \]

\( P \text{ may get in state } S \text{ iff the last event of } t \text{ belongs to } E_S(P) \)

\[ T(P|S) := \text{set of events that may occur in state } S \text{ (supposed } E_S(P) \neq \emptyset) \]

Internal assumptions: \( \text{int} \)

\[ P.\text{int} := \text{true if } \exists t: t' \in L_S(P) \text{ and } t'' \in T(P|S); \text{ otherwise } \text{false} \]

Interpretation: \( P \) can be in \( S \), and a successor event from \( T(P|S) \) exists

From the internal assumptions and the state definitions,

feasible sequences of events of \( \alpha P \) from the initial state to a final state are determined
External assumptions: express constraints on sequences of events belonging to different processes

logical constraints (such as precedences or mutual exclusions) are included in the formal Petri net model

Once states and assumptions have been formally defined, a set of legal traces \(\text{traces}(P)\) can be generated

Example: \(\text{traces}(P) = \{\langle\rangle, \langle e_1, e_2 \rangle, \langle e_1, e_2, e_1 \rangle, \langle e_1, e_3 \rangle, \langle e_1, e_3, e_4 \rangle\}\)

after the first event \((e_1)\), \(P\) can evolve in two different ways: \((e_2)\) or \((e_3)\)
Step 3: Domain transformation

Set of traces of each process is translated in its equivalent Petri subnet. This process can be performed automatically.

The following translation rules will be used in the later case study:

Rule 1: Sequence. \( \text{traces}(P) = \{ \langle \rangle, \langle e_1 \rangle, \langle e_1, e_2 \rangle \} \)

A trace of maximum size is translated:

- \textbf{events} are represented by transitions, \textbf{states} by places

- empty trace ... initial state of \( P \) ... initial place
- trace \( \langle e_1, e_2 \rangle \) ... final state of \( P \) ... last place
Rule 2: Recursion. $traces(P) = \{ \langle e_1, e_2 \rangle \}^*$

The sequence is cyclically closed by introduction of a dummy event corresponds to a transition from the final state to the initial state.
Rule 3: Concurrence (a). \( \text{traces}(P \parallel Q) = \{\langle e_2, e_1 \rangle\}^* \)

The alphabets of processes \( P \) and \( Q \) have a non-empty intersection \( \{e_2, e_1\} \)

Steps: separate translation of \( \text{traces}(P) \) and \( \text{traces}(Q) \)
identification and extraction of the common subnet
Rule 4: Concurrence (b). \( \text{traces}(P \parallel Q) = \text{interleaved}(\text{traces}(P), \text{traces}(Q)) \)

The alphabets of \( P \) and \( Q \) have empty intersection

Steps: separate translation of \( \text{traces}(P) \) and \( \text{traces}(Q) \)
parallel composition of the two Petri nets
add an initial place, and two dummy transitions
Rule 5: Synchronous Communication

Based on the semantics of a synchronous communication between two processes: **Two corresponding events are collapsed into a single one**
Step 4: Integration

Translation rules ... can be considered as a set of functions from the trace-based domain to the Petri net domain

Rules 1 and 2: translate the trace sets of individual processes

Rules 3, 4 and 5: compose pairs of processes

Compound process  = composition of two or more processes

processes P and Q are compound to P \ast Q

System with N concurrent processes: at most N – 1 compositions are needed to result in a Petri net skeleton for the system
Step 5: Adding external assumptions
External assumptions describe dependencies between states of processes
By these, the set of feasible traces is reduced

Classes of constraints:

- Precedence constraints on the order of events
  modeled by adding a place between the two involved events
  the first is a send event and the second a receive event
  the place represents a buffer where some message is stored

- Resource contention constraints
  concurrent access to a shared resource – can be described by a critical region
  in the Petri net, a critical region is modeled by adding a place
- Inhibition constraints
  depending on the current state of another process Q, the evolution of a process P is inhibited
  in terms of Petri nets, this is modeled by an inhibiting edge from that current state

- Constraints of non-deterministic behavior
  if a non-deterministic choice between a number of send events exists
  in terms of Petri nets: firing a send transition enables only one transition, and disables the other ones
Step 6: Temporization

In the temporization phase, the Petri net model is enhanced by introducing time and adding proper timing parameters.

Temporization can be introduced by associating delays with transitions, places, or tokens.

Timed transitions can be defined:
- either by a fixed time interval \((min, max)\)
- or by a random variable representing the firing delay

stochastic Petri nets
3.2.3 Case Study

... modeling a railway crossing system by a structured approach

**Steps:** definition of system requirements, system analysis,
Petri net specification, considering constraints, temporization

**Informal system description**
Three non-terminating processes: train (T), control mechanism (C), level crossing gate (P)

**Goal:** modeling a safe system
the train cannot be in the crossing while the gate is up

The requirements specify interactions between the controller and the train,
when the train approaches, enters or leaves the crossing
Meaningful Events

E1 synchronization between train and control (T, C):
train signals arrive at the control mechanism

E2 synchronization between train and control (T, C):
train informs the control that it has crossed the gate

E3 synchronization between control mechanism and gate (C, P):
the control mechanism tells the gate to lower the barrier

E4 synchronization between control mechanism and gate (C, P):
control mechanism tells the gate to raise the barrier

E5 train passing (T)

E6 moving barrier down (P)

E7 moving barrier up (P)
States (for each process)

**train:**

TS1  train is approaching the gate := there has not been any synchronization with the control mechanism yet (initial state)

TS2  train is leaving := its second synchronization has just taken place

TS3  train starts crossing := its first synchronization with the control mechanism has just taken place

TS4  train ends crossing := it has just passed

**control mechanism:**

CS1  control mechanism is waiting for the approaching signal from train := there has not been any synchronization with the train yet (initial state)

CS2  control mechanism is receiving the leaving-signal from train := down-signal synchronization with the gate has just taken place
CS3 control mechanism is sending the down-signal to gate := the first synchronization with the train has just taken place

CS4 control mechanism is sending the up-signal to gate := the second synchronization with the train has just taken place

CS5 control mechanism is resetting if the up-signal synchronization with the gate has just taken place

level gate:

PS1 gate is open if the barrier is up (initial state)

PS2 gate is closed if the barrier is down

PS3 gate is opening if the up-signal from the control mechanism has just arrived

PS4 gate is closing if the down signal from the control mechanism has just arrived
Assumptions

train

TA1 train is in state TS2 iff the previous state was TS4
TA2 train is in state TS3 iff the previous state was TS1
TA3 train is in state TS4 iff the previous state was TS3

control

CA1 control mechanism is in state CS2 iff the previous state was CS3
CA2 control mechanism is in state CS3 iff the previous state was CS1
CA3 control mechanism is in state CS4 iff the previous state was CS2
CA4 control mechanism is in state CS5 iff the previous state was CS4

level gate

PA1 gate is in state PS2 iff the previous state was PS4
PA2 gate is in state PS3 iff the previous state was PS2
PA3 gate is in state PS4 iff the previous state was PS1
PA4 gate is in state PS1 iff the previous state was PS3
Temporal specifications

Events $E_5, E_6, E_7$ are time consuming

Timing parameters: $T_{trans}, T_{down}, T_{up}$

All other events are assumed to be instantaneous

Definition of trace-based specifications

Process names: $P, C, T$; processes are assumed to be non-terminating

Event names:

$E_1$: sync1, $E_2$: sync2, $E_3$: sync3,
$E_4$: sync4, $E_5$: crossing, $E_6$: down, $E_7$: up

send and receive events are already replaced by their related communication events sync1, ..., sync4
**Alphabet of each process:**

\[ \alpha_T = \{ \text{sync1, sync2, crossing} \} \]
\[ \alpha_C = \{ \text{sync1, sync2, sync3, sync4} \} \]
\[ \alpha_P = \{ \text{sync3, sync4, up, down} \} \]

**Hence we see:**

\[ \alpha_T \cap \alpha_C = \{ \text{sync1, sync2} \}, \quad \alpha_C \cap \alpha_P = \{ \text{sync3, sync4} \}, \quad \alpha_T \cap \alpha_P = \emptyset \]

**Initial states:**

\[ \text{TS1: } T(T|\text{approaching}) = \{ \langle \rangle \} \]
\[ \text{CS1: } T(C|\text{waiting}) = \{ \langle \rangle \} \]
\[ \text{PS1: } T(P|\text{open}) = \{ \langle \rangle \} \]
Other states:

\( \text{TS2: } \text{sync2} \in E_{\text{leaving}(T)} \)

\( \text{TS3: } \text{sync1} \in E_{\text{startcross}(T)} \)

\( \text{TS4: } \text{crossing} \in E_{\text{endcross}(T)} \)

\( \text{CS2: } \text{sync3} \in E_{\text{leavsig}(C)} \)

\( \text{CS3: } \text{sync1} \in E_{\text{downsig}(C)} \)

\( \text{CS4: } \text{sync2} \in E_{\text{upsig}(C)} \)

\( \text{CS5: } \text{sync4} \in E_{\text{reset}(C)} \)

\( \text{PS1: } \text{up} \in E_{\text{open}(P)} \)

\( \text{PS2: } \text{down} \in E_{\text{close}(P)} \)

\( \text{PS3: } \text{sync4} \in E_{\text{opening}(P)} \)

\( \text{PS4: } \text{sync3} \in E_{\text{closing}(P)} \)
Internal assumptions: define far each process a total ordering relation in its alphabet

TA1: \( \text{PT.sync2}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync2} \} \) and \( t'' \in T(T|\text{endcross}) \)

TA2: \( \text{PT.sync1}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync1} \} \) and \( t'' \in T(T|\text{approaching}) \)

TA3: \( \text{PT.crossing}: \) true if \( \exists t, \) with \( t' \in \{ \text{crossing} \} \) and \( t'' \in T(T|\text{startcross}) \)

CA1: \( \text{PC.sync3}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync3} \} \) and \( t'' \in T(T|\text{downsig}) \)

CA2: \( \text{PC.sync1}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync1} \} \) and \( t'' \in T(T|\text{waiting}) \)

CA3: \( \text{PC.sync2}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync2} \} \) and \( t'' \in T(T|\text{leavsig}) \)

CA4: \( \text{PC.sync4}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync4} \} \) and \( t'' \in T(T|\text{upsig}) \)

PA1: \( \text{PP.down}: \) true if \( \exists t, \) with \( t' \in \{ \text{down} \} \) and \( t'' \in T(T|\text{closing}) \)

PA2: \( \text{PP.init}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync4} \} \) and \( t'' \in T(T|\text{close}) \)

PA3: \( \text{PP.end}: \) true if \( \exists t, \) with \( t' \in \{ \text{sync3} \} \) and \( t'' \in T(T|\text{open}) \)

PA4: \( \text{PP.up}: \) true if \( \exists t, \) with \( t' \in \{ \text{up} \} \) and \( t'' \in T(T|\text{opening}) \)
These assumptions define the precedence constraints between events belonging to the same alphabet

Legal traces of each process:

\[ \text{traces}(T) = \{ \langle \text{sync1, crossing, sync2} \rangle \}^* \]
\[ \text{traces}(C) = \{ \langle \text{sync1, sync3, sync2, sync4} \rangle \}^* \]
\[ \text{traces}(P) = \{ \langle \text{sync3, down, sync4, up} \rangle \}^* \]

Domain transformation

The traces of the three processes are translated into their equivalent Petri subnets. Applying rules 1 and 2 we obtain:
In the subnet for P, the transition for the cyclic closure has been replaced by the event "**up**" because the initial state and the final state are the same.
Integration

Integration of the subnets gives the Petri net skeleton of the system.

First we integrate $P$ and $C$ to get the net for $P*C$

Next, $T*C$ is constructed.

Then both, $P*C$ and $T*C$, are combined to the final Petri net $P*C*T$

Likewise, one could directly combine $P*C$ with $T$. 
Subnet of \( P^*C \):

The set of traces for the compound process \( P^*C \) are:

\[
traces(P^*C) = \{ \langle sync1, sync3, sync2, down, sync4, up \rangle \}^* \\
\cup \{ \langle sync1, sync3, down, sync2, sync4, up \rangle \}^*
\]
Subnet of $C^*T$:

The set of traces for the compound process $C^*T$ are:

$$\text{traces}(C^*T) = \{ \langle \text{sync1, crossing, sync3, sync2, sync4} \rangle \}^* \cup \{ \langle \text{sync1, sync3, crossing, sync2, sync4} \rangle \}^*$$
Petri net of $P^*C^*T$: 

- **open**
- **waiting**
- **approaching**
- **downsig**
- **sync 1**
- **startcross**
- **closing**
- **leavsig**
- **sync 3**
- **up**
- **sync 2**
- **down**
- **close**
- **upsig**
- **sync 1**
- **endcross**
- **opening**
- **reset**
- **leaving**

© 2003 K. H. Ecker, T.U. Clausthal. 3. Specification, ... 3-46
**Adding external assumptions**

There are no constraints involving different processes

**Reachability graph of the final net**

The possible markings of the states: M1, ..., M20 (the starting place is omitted):

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>M8</th>
<th>M9</th>
<th>M10</th>
<th>M11</th>
<th>M12</th>
<th>M13</th>
<th>M14</th>
<th>M15</th>
<th>M16</th>
<th>M17</th>
<th>M18</th>
<th>M19</th>
<th>M20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>approach</strong></td>
<td>1 0 0 0 0 0 0 0 1 0 1 1 0 0 1 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>waiting</strong></td>
<td>1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>open</strong></td>
<td>1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>startcross</strong></td>
<td>0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>downsig</strong></td>
<td>0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>endcross</strong></td>
<td>0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>leavsig</strong></td>
<td>0 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>closing</strong></td>
<td>0 0 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>leaving</strong></td>
<td>0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 1 1 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>upsig</strong></td>
<td>0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>close</strong></td>
<td>0 0 0 0 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>reset</strong></td>
<td>0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>opening</strong></td>
<td>0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Each marking (column) represents a state of the net.

The possible state transitions are as follows:
From an analysis we can state that the system is not safe:

Markings M3 and M6 are high risk states: in M3 the train has passed when the gate was open; in M6 the train passed when the gate was closing.

**Temporization**

Associate time delays with the transitions

- min- and max-values, or stochastic values

With time conditions, the high risk in states M3 and M6 can be avoided.

For example: put time constraints on transitions sync3, down, crossing, so that down can fire before crossing.

The minimum time delay associated with crossing must be greater than the max time needed for the sequence $\langle$sync3, down$\rangle$:

$$\text{min}(\text{crossing}) - \text{max}(\text{sync3}) > K \quad (K = \text{time for closing the barrier})$$

Other timing constraints: depend on train speed, train distance from the gate at the time of synchronization with the controller.
3.2.4 Object Orientation

**Specification and Modeling [Pereira 2000]:**

Object-orientation is a powerful approach to managing complexity

- Object-oriented design (OOD) is based on a small number of basic concepts:
  - objects, classes, operations and relationships
- Some important relationships are inheritance and aggregation

But its use in real-time systems development must be taken carefully:

- concurrency and deterministic time behavior may be difficult to deal with
- many intricate features may remain hidden in the object internals
  - they may not be available when considering the global behavior of the system
A concurrent object-model can be most suitable for developing large real-time systems
• **Understandability and maintainability:**

Object-oriented models allow a direct mapping of the problem domain semantics onto the model being built

- Objects in the model can be used to represent and behave like the corresponding elements in the real-world

• **Modularity:**

Identifiable objects/classes provide a logical partitioning of the problem domain

- Each object class represents a self contained piece of the problem domain which can be examined separately

• **Concurrence:**

Real-time systems have to interact with concurrent processes going on in the technical plant

- Objects become natural units for concurrent execution

This allows naturally modeling of the real-world concurrence of applications

• **Distribution:**

In an object-oriented model, the real-time system is organized as
- cooperative collection of objects,
- which may interact by passing messages one to another in a client-server fashion

• **Better management of complexity:**

Object-oriented techniques provide powerful abstraction concepts to deal with system complexity

  The different kinds of object hierarchies (generalization/specialization, aggregation) facilitate the presentation and management of varying levels of details

• **Increased extendability:**

The refinement or extension of a superclass into a subclass is very nicely supported by using inheritance mechanisms
But there are also quite a number of arguments in the opposite direction:

♦ **Timing requirements:**

Usually not considered by object-oriented methodologists are the description of timing requirements

− cyclic activation of object methods, deadlines,
− absolute time constraints between events

C. E. Pereira (1995) presented an object-oriented framework, where timing requirements are described in the scope of

• objects,
• classes,
• and their relationships
Object-orientation:

- obscures predictability, decreases efficiency
- can make the development much harder, because it is based on principles that contradict to real-time systems practice.
- employs abstraction to its limits while real-time systems clearly need more insight into the details and more explicit presentation, less abstraction

There is a fundamental difficulty in applying object-oriented concepts at the specification and design level of real-time systems development

Zalewski (2000)
3.3 Design Methods

3.3.1 Real-time HOOD

Objective of HRT-HOOD (Hard Real-Time HOOD):

- structured design method for hard real-time systems
- guaranteeing deterministic timing behavior
- based on preemptive priority scheduling theory.

Basic object types: PASSIVE, ACTIVE, PROTECTED, CYCLIC, SPORADIC
CYCLIC objects represent periodic activities  
they may spontaneously invoke operations in other objects,  
the only operations they have are requests which demand immediate attention

SPORADIC objects represent sporadic activities  
they may spontaneously invoke operations in other objects

ACTIVE objects control execution of their operations  
they may spontaneously invoke operations in other objects.

PROTECTED objects control execution of their operations  
they may not invoke operations in any other object

PASSIVE operations contain only sequential code which does not need to synchronize with any other object

Burns and Wellings (1994)
3.3.2 Real Time Specification Language RAISE

RAISE (Rigorous Approach to Industrial Software Engineering) is a formal method in the development of software systems

RAISE was developed at CRI Denmark as part of an ESPRIT project

RAISE consists of:
− the RAISE Specification Language (RSL) [RLG92]
− a comprehensive development method [RMG95]

The RAISE language supports the specification, design and implementation stages of the software development process.

The method is based on the stepwise refinement paradigm
The RAISE Specification Language RSL

supports the following specification styles

- concurrent processes which communicate with each other via synchronous channels
- polymorphism, inheritance and encapsulation

RSL is not executable

- There are currently no compilers or interpreters for RSL

RSL specifications can be very similar to imperative code

no explicit notion of time!
3.4 Real-Time Languages

3.4.1 Evolution of Programming Languages

- Assembly languages
- Basic
- Cobol
- Fortran
- Algol 60
- Algol 68
- Pascal
- Simula
- Smalltalk 80
- Fortran 77
- C
- C++
- FP
- Prolog
- Logo
- Lisp
- Hope
- Languages with RT-constructs
- Ada 9X
- Occam 2
- Chill
- Modula-2
- Concurrent Pascal
- Pearl
- Forth
- Modula-2
- PL1
- Concurrent Pascal
- Ada
The imperative, functional and logical paradigms offer different ways of problem description

- functional and logical: mainly subject to scientific discussion
- imperative: universal; used in practice

Within each group: Four main steps of development

- 60's: Procedural languages
- 70's: Structured programming
- 80's: Modularization
- 90's: Object oriented programming

Real-time programming:
- Timing conditions were first realized by assembler constructs
Three tendencies of development:

(1) Integration of real-time constructs in existing languages
   definition of timing conditions, execution of parallel processes
   Real-Time Fortran
   Real-Time Basic
   Concurrent Pascal
   
   Advantage: Host language is already accepted
   Drawback: Enhancing an existing language may lead to conceptual inconsistencies

(2) Development of special real-time languages
   e.g. Pearl, Forth
   
   Advantage: well suited for real-time applications
   Drawback: special language for - the relatively small area of - real-time applications
(3) Development of universal languages with integrated rt-constructs
e.g. Chill, Ada
... avoids the drawbacks of the first two approaches

Approach (3) is most promising, but experiences decreasing interest

Instead: C and C++ gain increasing interest, because

• they offer useful tools
  editors, debugger, integrated development environments etc.
• and there exist standards
  open systems, communication in networks

But the required real-time features are missing
  timing conditions, synchronization and scheduling of processes,
... must be included as library functions or by means of system calls

⇒ lack of understandability
3.4.2 Requirements on Real-Time Languages

• General requirements
  – bit manipulation operations
    access to registers (status, data registers) of peripheral devices
  – strict data typing of data structures
    recognition of type errors by the compiler
  – program development tools
    translation and debugging of program parts,

• Real-time requirements
  – constructs for the definition of timing conditions
  – predictability of real-time properties

• Process administration
  – definition and generation of parallel processes
  – synchronization of parallel processes
  – definition and alteration of process priorities
3.4.3 Parallel Processes in Ada

Processes in Ada

Ada Programming System Environment (APSE)
... contains a Real-Time Environment (RTE)
  responsible for parallel execution and synchronization of processes

Processes in Ada: "tasks" similar to light weight processes (threads)

After an Ada program (program unit, task) is started:
  a task can create new tasks in two ways, static or dynamic
Static generation: a declared task is assigned to one or more task objects
As soon as the task containing the declaration is executed: all defined task objects are executed

Example

```
TASK TYPE buffer IS -- specification
  ...
END buffer;
TASK BODY buffer IS -- body
  ... -- declarations
BEGIN
  ... -- instructions
END buffer;
```

Static generation of 10 task objects:

```
PROCEDURE system IS
  ... -- declaration of buffer etc.
  b: ARRAY (1..10) OF buffer;
  ...
BEGIN
  ... -- instructions
END system;
```
Dynamic generation of task objects:
Dynamic declaration needs a task declaration and a special access object (pointer to the task declaration)
New task objects are generated during execution of the body of the task containing the declaration (by applying a NEW-clause)
this allows to generate exactly as many new task objects as required in the particular application.

Example

```plaintext
PROCEDURE system IS  
  ... -- declaration of buffer etc.
  TYPE zbuffer IS ACCESS buffer;
  zb: ARRAY (1..10) OF zbuffer;
  ...  
BEGIN  
  ...  
  zb(5) := NEW buffer;  
  ...  
END system
```
General task execution rule in Ada

A task is completed as soon as

(i) all the instructions of its block are executed, and
(ii) all task objects generated in this block are completed.

Additional concepts important in the context of real-time processing are remote calls, synchronization, timing instructions, communication

Realization of remote calls of tasks

- a client task issues the remote (entry-) call
- a server task accepts the remote call
Example: producer-consumer

TASK TYPE buffer IS-- declaration of a server task
ENTRY inbuffer(c: IN character);-- specification
ENTRY outbuffer(c: OUT character); -- of ENTRY calls
END buffer;

TASK BODY buffer IS
  n: CONSTANT integer := 128;
  count: integer RANGE 0..n;
  i, j: integer RANGE 1..n;
  cstring: ARRAY(1..n) OF character;
BEGIN
  count := 0; -- number of used positions
  i := 1;     -- next free position
  j := i;     -- last used position
  LOOP
    SELECT
      WHEN count < n => -- buffer not full
        ACCEPT inbuffer(c: IN character) DO
          cstring(i) := c; -- instructions
        END inbuffer;
        count := count + 1;
  END LOOP;
i := (i MOD n) + 1;
OR
WHEN count > 0 => -- buffer not empty
  ACCEPT outbuffer(c: OUT character) DO
    c := cstring(j);
  END outstring;
  count := count – 1;
  j := (j MOD n) + 1;
END SELECT;
END LOOP;
end buffer;

If \( b \) is a generated task object of type \texttt{buffer}, a client task may access the service of the server by issuing e.g. \( b.\text{inbuffer}(x) \) "rendezvous": the client waits for the server to accept and execute the call. Likewise, the server waits for a call from the client

One property of this solution is that a client may have to wait an undefined amount of time for service.
Timing constructs

The **DELAY** statement allows to define an upper bound for the waiting time

```plaintext
SELECT
  ...
END SELECT
```

If `timing_condition` is true then the **SELECT** instruction is terminated after `delta_t` time units.

`delta_t` is of predefined type `duration`

Predefined packet `calendar` contains useful declarations and operations to handle time

- Predefined type `TIME`; unit: second; required accuracy: 50 ms
- Function `clock`: returns the current time
Example: periodic execution of a sequence of instructions

Task poll waits for $\text{delta}_t$ time units and then executes the instruction sequence again:

```plaintext
WITH calendar ; USE calendar
TASK BODY poll IS
    nextexec: TIME;
    delta_t: CONSTANT duration: 1.0; -- one second
BEGIN
    nextexec := clock + delta_t;
    LOOP
        DELAY nextexec - clock;
        ... -- sequence of instructions
        nextexec := nextexec + delta_t;
    END LOOP;
END;
```
User-defined process preemptions

A preemption statement specifies the process to be started, and the position in the interrupt vector:

FOR entry_name USE AT entry_address

Example. *ir_example* is a task that processes two kinds of interrupts:
one is caused by the periodically generated clock pulses,
and the other by externally signals.

The corresponding tasks to be performed upon these signals are *cp_handler* and *sig_handler*

```plaintext
TASK ir_example IS
ENTRY cp_handler;
ENTRY sig_handler;
FOR cp_handler USE AT 16#BC#;
FOR sig_handler USE AT 16#C8#;
END ir_example;
TASK BODY ir_example IS
...
```
i := integer := 0;  -- number of clock pulses
j := integer := 0;  -- number of signals
BEGIN
  LOOP
    SELECT
      ACCEPT cp_handler DO
      i := i + 1;  -- next clock pulse
      END cp_handler;
      IF i > ...  -- end of a given period
      THEN ...  -- do something
    OR
      ACCEPT sig_handler DO
      j := j+1; -- next signal arrived
      END sig_handler;
      ...  -- do something
      END SELECT;
    END LOOP;
END ir_example;
Priorities of tasks objects

If more than one task are to be processed on one processor, the order of execution is generally undefined and hence open.

To enable predictability, the execution order can be controlled by the introduction of priorities: **PRAGMA-clause**

```
TASK buffer IS
    ...
    PRAGMA priority(9);
    ...
END buffer;
```

If a client task has higher priority than the server task (*buffer*), a call to one of the server's service tasks will increase the server's priority temporarily (priority inheritance)
Real-time extensions of Ada: Ada 9X

Properties of Ada (real-time and others):

+ strict typing
+ support of program development
+ parallel programming (rendezvous concept)
  - weak real time concepts (only special data structure and basic functions)
  - delays: only lower bounds, no upper bounds for delays


• precise periodic processor assignments
• recognition of violated deadlines
• asynchronous communication
• transmission of continuous data (multimedia applications)
• dynamic priorities
• general priority inheritance
• improved support of distributed program executions
3.4.4 Process Language PEARL

PEARL = Process and Experiment Automation Realtime Language

Development started in the late 60's in Germany

Some features:

– structured programming concepts

– program partition in modules, separate translation

– interfaces to the technical system and to the computer system are defined by logical names

– processes, called tasks, can be assigned a numerical priority
  smaller value $\equiv$ higher priority

– tasks can be declared to be "resident"
  remain in the main memory
Declaration scheme:

```c
task_example: TASK PRIO 10 RESIDENT
... /* declaration of local variables */
... /* instructions */
END; /* TASK task_example */
```

- task synchronization: by counting semaphores
  variables of type `SEMAPHOR` must be globally declared
  and initialized with `PRESET`

  operations on semaphores:
  - P-operation: `REQUEST`
  - V-operation: `RELEASE`
Task management
Runtime system manages the states of tasks and causes transitions between states:

existent: the task is known to the runtime system
active: the task is ready to be processed or running
scheduled: the task waits to enter the state active
    transition from existent to scheduled by the activate command
delayed: a task can delay itself
    – by calling suspend:
        another task has to call continue to bring the task back to active
    – by calling resume the task can define time or event for its continuation
State transitions can be
- time driven
times for state transitions are entered in a wakeup list
if a wakeup time is reached, the runtime system performs the corresponding transition
- event driven
the runtime system maintains an event list

Scheduling of tasks
- direct transition from existent to active:
  ACTIVATE task_example PRIO 5;
- event driven transition:
  transition from existent to scheduled:
  WHEN in2 ACTIVATE task_example PRIO 1;
  arrival of event in2 causes state transition form scheduled to active
- time driven transitions:
  
at point:
  
  AT 12:0:0 ACTIVATE task_example;
  AFTER 10 SEC ACTIVATE task_example;

  periodic:
  
  ALL 10 SEC ACTIVATE task_example;
  ALL 10 SEC UNTIL 12:0:0 ACTIVATE task_example;
  ALL 10 SEC DURING 3600 SEC ACTIVATE task_example;

- additional possibilities:
  
  termination of a task by TERMINATE
  PREVENT causes a transition from scheduled to existent
exists \rightarrow scheduled

PREVENT

ACTIVATE

time or event driven

TERMINATE

active

running \rightarrow running

PRIO

state transition diagram
Summary of Chapter 3

Useful tools for specification and design of especially large systems are

- Petri nets
- object orientation

but these concepts also their drawbacks

⇒ reason that they are not widely accepted

Several design methods have been developed, such as

- Real-Time HOOD
- RAISE
- Chapter 7 also gives a short introduction to OSEK / VDX, complete specification and design system

On the implementation level, languages should support

- the realization of timing requirements
- and process administration