Embedded Systems
Petri Nets
Computing changes of markings

- “Firing” transitions $t$ generate new markings on each of the places $p$ according to the following rules:

$$M'(p) = \begin{cases} 
M(p) - W(p,t), & \text{if } p \in \bullet t \setminus t^* \\
M(p) + W(t,p), & \text{if } p \in t^* \setminus \bullet t \\
M(p) - W(p,t) + W(t,p), & \text{if } p \in \bullet t \cap t^* \\
M(p) & \text{otherwise}
\end{cases}$$

When a transition $t$ fires from a marking $M$, $w(p, t)$ tokens are deleted from the incoming places of $t$ (i.e. from places $p \in \bullet t$), and $w(t, p)$ tokens are added to the outgoing places of $t$ (i.e. to places $p \in t^*$), and a new marking $M'$ is produced.
Activated transitions

- Transition $t$ is "activated" iff

$$(\forall p \in \bullet t : M(p) \geq W(p,t)) \land (\forall p \in t^\bullet : M(p) + W(t,p) \leq K(p))$$

Activated transitions can "take place" or "fire", but don't have to. The order in which activated transitions fire is not fixed (it is non-deterministic).
Boundedness

- A place is called \textit{k-safe} or \textit{k-bounded} if it contains in the initial marking $m_0$ and in all other reachable from there markings at most $k$ tokens.

- A net is \textit{bounded} if each place is bounded.

- \textbf{Boundedness: the number of tokens in any place cannot grow indefinitely}
- \textbf{Application: places represent buffers and registers (check there is no overflow)}

- A Petri net is inherently bounded if and only if all its reachability graphs (i.e. reachability graphs with all possible starting states) all have a \textit{finite number of states}. 
Liveness

- A transition $T$ is live if in any marking there exists a firing sequence such that $T$ becomes enabled.
- An entire net is live if all its transitions are live.
- Important for checking deadlock.
Liveness (more precisely)

- A **transition** \( t \) is **dead** at \( M \) if no marking \( M' \) is reachable from \( M \) such that \( t \) can fire in \( M' \).
- A **transition** \( t \) is **live** at \( M \) if there is no marking \( M' \) reachable from \( M \) where \( t \) is dead.
- A **marking** is **live** if all transitions are live.
- A **P/T net** is **live** if the initial marking is live.

**Observations:**
- A live net is deadlock-free.
- No transition is live if the net is not deadlock-free.
Deadlock

- A **dead marking (deadlock)** is a marking where no transition can fire.
- A Petri net is **deadlock-free** if no dead marking is reachable.
Shorthand for changes of markings

Firing transition:

\[ M'(p) = \begin{cases} 
M(p) - W(p, t), & \text{if } p \in t^* \setminus t^* \\
M(p) + W(t, p), & \text{if } p \in t^* \setminus t^* \\
M(p) - W(p, t) + W(t, p), & \text{if } p \in t^* \cap t^* \\
M(p) & \text{otherwise}
\end{cases} \]

Let

\[ t(p) = \begin{cases} 
-W(p, t) & \text{if } p \in t^* \setminus t^* \\
+W(t, p) & \text{if } p \in t^* \setminus t^* \\
-W(p, t) + W(t, p) & \text{if } p \in t^* \cap t^* \\
0 & \text{otherwise}
\end{cases} \]

\[ \forall p \in P: M'(p) = M(p) + t(p) \]

\[ M' = M + t \quad +: \text{ vector add} \]
Matrix $N$ describing all changes of markings

$N(p,t) = \begin{cases} 
-W(p,t) & \text{if } p \in \cdot t \setminus \cdot \\
+W(t,p) & \text{if } p \in t \setminus \cdot t \\
-W(p,t)+W(t,p) & \text{if } p \in t \cap \cdot t \\
0 & \text{otherwise}
\end{cases}$

Def.: Matrix $N$ (incidence matrix) of net $N$ is a mapping $N: P \times T \rightarrow Z$ (integers)

such that $\forall t \in T: N(p,t) = t(p)$

Component in column $t$ and row $p$ indicates the change of the marking of place $p$ if transition $t$ takes place.
Incidence matrix

Incidence matrix $N$ of a pure (no elementary loops) place/transition-net:

$$N_{p,t} := \begin{cases} 
-W(t, p), & \text{arc from } p \text{ to } t \\
+W(t, p), & \text{arc from } t \text{ to } p \\
0, & \text{otherwise}
\end{cases}$$
**Example:** $N = \begin{array}{|c|c|c|c|c|c|c|c|}
\hline
 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} \\
\hline
p_1 & 1 & & & & & & & & & -1 \\
p_2 & -1 & 1 & & & & & & & & \\
p_3 & & -1 & 1 & & & & & & & \\
p_4 & & & -1 & 1 & & & & & & \\
p_5 & & & & -1 & 1 & & & & & \\
p_6 & & & & & -1 & 1 & & & & \\
p_7 & & & & & & -1 & 1 & & & \\
p_8 & & & & & & & -1 & & & \\
p_9 & & & & & & & & 1 & 1 & \\
p_{10} & & & & & & & & & -1 & 1 \\
p_{11} & & & & & 1 & & & & & \\
p_{12} & & & & & & 1 & & & & \\
p_{13} & & & & & & & 1 & & & \\
\hline
\end{array}
State equation

\[ N_{t+1} = p_1 N_t + p_2 N_{t-1} + p_3 N_{t-2} \]

\[ m' = m_0 + N_{t+1} \]

\[ n' = \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & -1 & -1 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ 0 \\ 0 \end{bmatrix} \]

\[ \begin{bmatrix} v_0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \]
State equation

reachability graph
We are interested in subsets $R$ of places whose number of labels remain invariant under firing of transitions:

- e.g. the number of trains commuting between Amsterdam and Paris (Cologne and Paris) remains constant

Important for correctness proofs
Place - invariants

Standardized technique for proving properties of system models

For any transition $t_j \in T$ we are looking for sets $R \subseteq P$ of places for which the accumulated marking is constant:

$$\sum_{p \in R} t_j(p) = 0$$

Example:
Characteristic Vector

\[ \sum_{p \in R} t_{-j}(p) = 0 \]

Let:
\[ c_R(p) = \begin{cases} 
1 & \text{if } p \in R \\
0 & \text{if } p \notin R 
\end{cases} \]

\[ \Rightarrow \sum_{p \in R} t_{-j}(p) = t_{-j} \cdot c_R = \sum_{p \in P} t_{-j}(p) c_R(p) = 0 \]

Scalar product
Condition for place invariants

\[ \sum_{p \in R} t_j(p) = t_j \cdot c_R = \sum_{p \in P} t_j(p) c_R(p) = 0 \]

Accumulated marking constant for all transitions if

\[ t_1 \cdot c_R = 0 \]

... ... ...

\[ t_n \cdot c_R = 0 \]

Equivalent to \( N^T c_R = 0 \) where \( N^T \) is the transposed of \( N \)
More detailed view of computations

\[
\begin{pmatrix}
\ell_1(p_1) & \ldots & \ell_1(p_n) \\
\ell_2(p_1) & \ldots & \ell_2(p_n) \\
\vdots & \ddots & \vdots \\
\ell_m(p_1) & \ldots & \ell_m(p_n)
\end{pmatrix}
\begin{pmatrix}
c_R(p_1) \\
c_R(p_2) \\
\vdots \\
c_R(p_n)
\end{pmatrix}
= \begin{pmatrix}
0 \\
0 \\
0 \\
0
\end{pmatrix}
\]

System of linear equations.

Solution vectors must consist of zeros and ones.

Different techniques for solving equation system (Gauss elimination, tools e.g. Matlab, …)
**Application to Thalys example**

\[ N^T c_R = 0, \text{ with } N^T = \]

\[ \begin{array}{cccccccccccc}
  & p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 & p_8 & p_9 & p_{10} & p_{11} & p_{12} & p_{13} \\
 t_1 & 1 & -1 & & & & & & -1 & & & & 1 \\
 t_2 & 1 & -1 & 1 & -1 & & & & & & & & 1 \\
 t_3 & 1 & -1 & 1 & -1 & & & & & & & & 1 \\
 t_4 & & & & & & & & & & & & 1 \\
 t_5 & & & & & & & & & & & & 1 \\
 t_6 & & & & & & & & & & & & 1 \\
 t_7 & & & & & & & & & & & & 1 \\
 t_8 & & & & & & & & & & & & 1 \\
 t_9 & & & & & & & & & & & & 1 \\
 t_{10} & & & & & & & & & & & & 1 \\
\end{array} \]

\[ c_{R,1} = \left( \begin{array}{cccccccccccc}
 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array} \right) \]
Solution vectors for Thalys example

We proved that:
- the number of trains serving Amsterdam, Cologne and Paris remains constant.
- the number of train drivers remains constant.

\[
\begin{align*}
C_{R,1} &= (1 1 1 1 1 1 0 0 0 0 0 0 0 0 0)
\\
C_{R,2} &= (0 0 0 0 0 0 1 1 0 0 0 1 0)
\\
C_{R,3} &= (0 0 0 0 0 0 0 0 1 1 0 0 1)
\\
C_{R,4} &= (1 0 0 0 1 1 0 0 1 1 1 0 0)
\end{align*}
\]
Solution vectors for Thalys example

It follows:

• each place invariant must have at least one label at the beginning, otherwise “dead”
• at least three labels are necessary in the example
\[ N^T c_R = 0, \text{ with } N^T = \]

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1</td>
<td>φ</td>
<td>φ</td>
<td>-1</td>
<td>-1</td>
<td>φ</td>
</tr>
<tr>
<td>T2</td>
<td>φ</td>
<td>1</td>
<td>φ</td>
<td>φ</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>T3</td>
<td>-1</td>
<td>φ</td>
<td>1</td>
<td>1</td>
<td>φ</td>
<td>φ</td>
</tr>
<tr>
<td>T4</td>
<td>φ</td>
<td>-1</td>
<td>1</td>
<td>φ</td>
<td>φ</td>
<td>1</td>
</tr>
<tr>
<td>T5</td>
<td>φ</td>
<td>φ</td>
<td>-1</td>
<td>φ</td>
<td>-1</td>
<td>φ</td>
</tr>
<tr>
<td></td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
<td>P5</td>
<td>P6</td>
</tr>
<tr>
<td>---</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>T1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>T2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>T3</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T4</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>T5</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Place - invariants
Predicate/transition nets

- Goal: compact representation of complex systems.
- Key changes:
  - Tokens are becoming individuals;
  - Transitions enabled if functions at incoming edges true;
  - Individuals generated by firing transitions defined through functions
- Changes can be explained by folding and unfolding C/E nets
Example: Dining philosophers problem

- $n > 1$ philosophers sitting at a round table;
- $n$ forks,
- $n$ plates with spaghetti;
- philosophers either thinking or eating spaghetti (using left and right fork).

How to model conflict for forks?

How to guarantee avoiding starvation?

2 forks needed!
Condition/event net model of the dining philosophers problem

- Let \( x \in \{1..3\} \)
- \( t_x: x \) is thinking
- \( e_x: x \) is eating
- \( f_x: \) fork \( x \) is available

Model quite clumsy.
Difficult to extend to more philosophers.
Predicate/transition model of the dining philosophers problem (1)

- Let $x$ be one of the philosophers,
- let $l(x)$ be the left spoon of $x$,
- let $r(x)$ be the right spoon of $x$.

- Tokens individuals
- Edges can be labeled with variables and functions
Predicate/transition model of the dining philosophers problem (1)
Predicate/transition model of the dining philosophers problem (2)

- Model can be extended to arbitrary numbers of people.
- No change of the structure.
Time and Petri Nets

- e.g.: Petri nets tell us that ""a new request can be issued only after the resource is released"

- Nothing about time

- In literature, time has been added to PNs in many different ways (notion of temporal constraints for: transitions, places, arcs) → TPN
Timed Petri Nets

- TPN
  - Each transition is defined precisely based on connectivity and tokens needed for transition
  - Given an initial condition, the exact system state at an arbitrary future time $T$ can be determined

- Timed Petri Nets becomes a 7-tuple system
  - $PN = (P, T, F, W, K, M_0, \tau)$
  - $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is a finite set of deterministic time delays to corresponding $t_i$
Time and Petri Nets (TPN)

- adding (quantitative) time to PNs is to introduce temporal constraints on its elements:
  - e.g., a transition must fire after 5 msec
Production system - Top level petri net
magazine/depot
NC axis
Abbildung 5.7: E1.7 – MPS (Teil 1)

Abbildung 5.8: E1.7 – SIMULATOR (Teil 1)
Evaluation

- **Pros:**
  - Appropriate for distributed applications,
  - Well-known theory for formally proving properties,

- **Cons:**
  - PN problems with modeling timing (extensions in TPN)
  - no programming elements, no hierarchy (extensions available)

- **Extensions:**
  - Enormous amounts of efforts on removing limitations.

- **Remark:**
  - A FSM can be represented by a subclass of Petri nets, where each transition has exactly one incoming edge and one outgoing edge.
Summary

- Petri nets: focus on causal dependencies
  - Condition/event nets
    - Single token per place
  - Place/transition nets
    - Multiple tokens per place
  - Predicate/transition nets
    - Tokens become individuals
    - Dining philosophers used as an example
  - Extensions required to get around limitations
SDL - Specification and Description Language
SDL - Specification and Description Language

- Used here as a (prominent) example of a model of computation based on asynchronous message passing communication.
- Appropriate also for distributed systems
- Language designed for specification of distributed systems.
  - Dates back to early 70s,
  - Formal semantics defined in the late 80s,
  - Defined by ITU (International Telecommunication Union): Z.100 recommendation in 1980
- Another acronym SDL (“System Design Languages”)
SDL - Specification and Description Language

- Provides textual (tool processing) and graphical formats (user interaction)

- Ability to be used as a wide spectrum language from requirements to implementation

- Just like StateCharts, it is based on the CFSM (Communicating FSM) model of computation; each FSM is called a process.

- With SDL the protocol behaviour is completely specified by communicating FSM.

- The formal basis of SDL enables the use of code generation tool chains, which allows an automated implementation of the specification.
SDL - Specification and Description *Language*

- However, it uses *message passing* instead of shared memory for communications
- SDL supports operations on data
- object oriented description of components.
Structuring SDL designs

SDL systems can be structured in various means:

- A system consists of a number of blocks connected by channels, each block may contain a substructure of blocks or it may contain process sets connected by signals.

- Processes execute concurrently with other processes and communicate by exchanging signals; or by remote procedure calls.
Specifying behaviour

1. The behaviour of a process is described as an extended FSM: When started, a process executes its start transition and enters the first state. (triggered by signals)

2. In transitions, a process may execute actions.

3. E.g.: Actions can assign values to variable attributes of a process, branch on values of expression, call procedures, create new processes, send signal to other processes.
SDL-representation of FSMs/processes
Communication among SDL-FSMs

- Communication between FSMs (or “processes“) is based on **message-passing**, assuming a **potentially indefinitely large FIFO-queue**.

- Each process fetches next entry from FIFO,
- checks if input enables transition,
- if yes: transition takes place,
- if no: input is ignored (exception: SAVE-mechanism).
Determinate?

- Let tokens be arriving at FIFO at the same time:
  - Order in which they are stored, is unknown:

All orders are legal: simulators can show different behaviors for the same input, all of which are correct.
Operations on data

- Variables can be declared locally for processes.
- Their type can be predefined or defined in SDL itself.
- SDL supports abstract data types (ADTs). Examples:

```plaintext
DCL
Counter Integer;
Date String;
```

```plaintext
Counter := Counter + 3;
```

Diagram:

```
(1:10)  (11:30)  ELSE
```

CS - ES
Process interaction diagrams

- Interaction between processes can be described in process interaction diagrams (special case of block diagrams).
- In addition to processes, these diagrams contain channels and declarations of local signals.
- Example:
Designation of recipients

1. **Through process identifiers:**
   Example: OFFSPRING represents identifiers of processes generated dynamically.

2. **Explicitly:**
   By including the channel name.

3. **Implicitly:**
   If signal names imply channel names (B → Sw1)
Hierarchy in SDL

- Process interaction diagrams can be included in **blocks**. The root block is called **system**.

Processes cannot contain other processes, unlike in StateCharts.
Hierarchy of a SDL specification
Timers

- Timers can be declared locally. Elapsed timers put signal into queue (not necessarily processed immediately).
- RESET removes timer (also from FIFO-queue).
SDL application

The semantics of SDL defines the state space of the specification. This state space can be used for various analyses and transformation techniques, e.g.:

- state space exploration, simulation
- checking the SDL-specification for deadlocks/lifelocks
- deriving test cases automatically
- code generation for an executable prototype or end system
Summary

- MoC: finite state machine components
  + non-blocking message passing communication

- Representation of processes

- Communication & block diagrams

- Timers and other language elements

- Excellent for distributed applications (e.g., *Integrated Services Digital Network* (ISDN))

- Commercial tools available from SINTEF, Telelogic, Cinderella ([//www.cinderella.dk](http://www.cinderella.dk))