## Models of computation for embedded systems

<table>
<thead>
<tr>
<th>Communication/ local computations</th>
<th>Shared memory</th>
<th>Message passing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Synchronous</td>
</tr>
<tr>
<td>Communicating finite state machines</td>
<td>StateCharts, StateFlow</td>
<td>SDL, MSCs</td>
</tr>
<tr>
<td>Data flow model</td>
<td></td>
<td>SDL</td>
</tr>
<tr>
<td>Computational graphs</td>
<td></td>
<td>C, C++, Java</td>
</tr>
<tr>
<td>Von Neumann model</td>
<td>C, C++, Java</td>
<td>SDL</td>
</tr>
<tr>
<td>Discrete event (DE) model</td>
<td>VHDL, Simulink</td>
<td>SDL</td>
</tr>
</tbody>
</table>
Place/transition nets

Def.: \((P, T, F, K, W, M_0)\) is called a place/transition net (P/T net) iff

1. \(N=(P,T,F)\) is a net with places \(p \in P\) and transitions \(t \in T\)
2. \(K: P \rightarrow (\mathbb{N}_0 \cup \{\omega\}) \setminus \{0\}\) denotes the capacity of places
   (\(\omega\) symbolizes infinite capacity)
3. \(W: F \rightarrow (\mathbb{N}_0 \{0\})\) denotes the weight of graph edges
4. \(M_0: P \rightarrow \mathbb{N}_0 \cup \{\omega\}\) represents the initial marking of places

In the following: assume initial marking is finite, capacity \(\omega\).

Unbounded Petri net

Defaults:
- \(K = \omega\)
- \(W = 1\)
Invariants & boundedness

- A net is **covered** by place invariants iff every place is contained in some invariant.
- **Theorem 4:**
  - a) If $R$ is a place invariant and $p \in R$, then $p$ is bounded.
  - b) If a net is covered by place invariants then it is bounded.

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.
Structural properties: deadlock-traps

- A place set $S$ is a **(static) deadlock** if every transition that adds token from $S$ also removes token from $S$.
- A place set $S$ is a **trap** if every transition that removes token from $S$ also adds token to $S$.

Empty structural deadlocks and marked traps

- Empty structural deadlocks are never re-marked;
- Marked traps are never emptied.
Sufficiently marked places

A place is called sufficiently marked if there are enough token for one of the outgoing transitions:

- Define
  \[ W(p) = \min \{ W(p,t) \mid (p,t) \in F \} \]
  if there exists a \((p,t) \in F\) and 0 otherwise

- Place \(p\) is **sufficiently marked** in marking \(M\), if \(M(p) \geq W(p)\)

- A set of places is sufficiently marked if it contains a sufficiently marked place.

Deadlock-Trap Property

- A P/T has the **deadlock-trap property**, if every (static) deadlock contains a trap that is sufficiently marked in \(M_0\).
Deadlock-Trap Property

Theorem 5:
Every homogeneous P/T net with non-blocking weights that has the deadlock-trap property is deadlock-free.

Homogeneous: For each place, all outgoing edges have the same weight.
Non-blocking weights: $W^-(p) \geq W^+(p)$
- $W^-(p) = \min \{ W(p,t) \mid (p,t) \in F \}$ if there exists a $(p,t) \in F$ and 0 otherwise
- $W^+(p) = \min \{ W(t,p) \mid (t,p) \in F \}$ if there exists a $(t,p) \in F$ and 0 otherwise

Fairness

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).
- 2 forks needed!
Fairness

Let $N$ be a Petri net and $w$ an execution of $N$.

- $w$ is **impartial** with respect to a set of transitions $T$ iff every transition in $T$ occurs infinitely often in $w$.
- $w$ is **just** with respect to a set of transitions $T$ iff every transition in $T$ that is enabled in all except finitely many markings occurs infinitely often in $w$.
- $w$ is **fair** with respect to a set of transitions $T$ iff every transition in $T$ that is enabled in infinitely many markings occurs infinitely often in $w$.

- $w$ is impartial $\Rightarrow$ $w$ is fair
- $w$ is fair $\Rightarrow$ $w$ is just

Persistent nets

**Theorem 7:** If the net is persistent, then every just execution is fair.
Theorem 8: Let $N$ be a bounded net, $t$ a live transition, and $w$ a state-fair execution of $N$. Then $t$ occurs infinitely often in $w$.

Extensions: finite capacities

- $K(p)=4$
Extensions: Petri nets with priorities

- $t_1 \prec t_2 : t_2$ has higher priority than $t_1$.

- Petri nets with priorities are Turing-complete.

Predicate/transition model of the dining philosophers problem

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

Token: individuals.
Semantics can be defined by replacing net by equivalent condition/event net.
Model can be extended to arbitrary numbers.
Summary Petri nets

Pros:
- Appropriate for distributed applications,
- Well-known theory for formally proving properties,
- Initially theoretical topic, but now widely adapted in practice due to increasing number of distributed applications.

Cons (for the nets presented):
- problems with modeling timing,
- no programming elements,
- no hierarchy.

Extensions:
- Enormous amounts of efforts on removing limitations.

Data Flow Models
Data flow modeling

- **Def.**: The process of identifying, modeling and documenting how data moves around an information system.

Data flow modeling examines
- *processes* (activities that transform data from one form to another),
- *data stores* (the holding areas for data),
- *external entities* (what sends data into a system or receives data from a system, and
- *data flows* (routes by which data can flow).

Data flow as a “natural” model of applications

[Diagram: Registering for courses and Video on demand system]


www.ece.ubc.ca/~irenek/techpaps/vod/vod.html
Process networks

Many applications can be specified in the form of a set of communicating processes.

Example: system with two sensors:

```
<table>
<thead>
<tr>
<th>Temperature sensor</th>
<th>Humidity sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>mux</td>
<td>FIFO</td>
</tr>
</tbody>
</table>
```

Alternating read

```
loop
read_temp; read_humidity
until false;
```

of the two sensors
not the right approach.

The case for multi-process modeling in imperative languages

```
MODULE main;
TYPE some_channel =
   (temperature, humidity);
some_sample : RECORD
   value : integer;
   line : some_channel
END;
PROCESS get_temperature;
VAR sample : some_sample;
BEGIN
LOOP
sample.value := new_temperature;
IF sample.value > 30 THEN ....
sample.line := temperature;
to_fifo(sample);
END
END get_temperature;

PROCESS get_humidity;
VAR sample : some_sample;
BEGIN
LOOP
sample.value := new_humidity;
sample.line := humidity;
to_fifo(sample);
END
END get_humidity;
BEGIN
get_temperature; get_humidity;
END;
```

- Blocking calls new_temperature, new_humidity
- Structure clearer than alternating checks for new values in a single process

How to model dependencies between tasks/processes?
**Dependences between processes/tasks**

- Get_temperature
- Get_humidity
- FIFO

**Task graphs**

- **Def.:** A **dependence graph** is a directed graph $G=(V,E)$ in which $E \subseteq V \times V$ is a partial order.
- If $(v_1, v_2) \in E$, then $v_1$ is called an **immediate predecessor** of $v_2$ and $v_2$ is called an **immediate successor** of $v_1$.
- Suppose $E^*$ is the transitive closure of $E$. If $(v_1, v_2) \in E^*$, then $v_1$ is called a **predecessor** of $v_2$ and $v_2$ is called a **successor** of $v_1$.

Nodes are assumed to be a „program“ described in some programming language, e.g. C or Java.
Reference model for data flow:
Kahn process networks (1974)

For asynchronous message passing:
communication between tasks is buffered

Special case: Kahn process networks:
executable task graphs;
Communication via infinitely large FIFOs

For asynchronous message passing:
communication between tasks is buffered
Properties of Kahn process networks (1)

- Each node corresponds to one program/task;
- Communication is only via channels;
- Channels include FIFOs as large as needed;
- Channels transmit information within an unpredictable but finite amount of time;
- Mapping from $\geq 1$ input sequence to $\geq 1$ output sequence;
- In general, execution times are unknown;
- Send operations are non-blocking, reads are blocking.
- One producer and one consumer; i.e. there is only one sender per channel;

Properties of Kahn process networks (2)

- There is only one sender per channel.
- A process cannot check whether data is available before attempting a read.
- A process cannot wait for data for more than one port at a time.
- Therefore, the order of reads depends only on data, not on the arrival time.
- Therefore, Kahn process networks are deterministic (!), for a given input, the result will always the same, regardless of the speed of the nodes.

This is the key beauty of KPNs!
A Kahn Process

process f(in int u, in int v, out int w)
{
    int i; bool b = true;
    for (;;)
    {
        i = b ? wait(u) : wait(w);
        printf("%dn", i);
        send(i, w);
        b = !b;
    }
}


A Kahn Process

process f(in int u, in int v, out int w)
{
    int i; bool b = true;
    for (;;)
    {
        i = b ? wait(u) : wait(w);
        printf("%dn", i);
        send(i, w);
        b = !b;
    }
}

A Kahn Process

process g(in int u, out int v, out int w)
{
    int i; bool b = true;
    for(;;) {
        i = wait(u);
        if (b) send(i, v); else send(i, w);
        b = !b;
    }
}

Process reads from u and alternately copies it to v and w

A Kahn System

- Prints an alternating sequence of 0’s and 1’s

Emits a 1 then copies input to output

Emits a 0 then copies input to output
**Definition: Kahn networks**

A **Kahn process network** is a directed graph \((V,E)\), where

- \(V\) is a set of **processes**, 
- \(E \subseteq V \times V\) is a set of **edges**, 
- associated with each edge \(e\) is a **domain** \(D_e\) 
- \(D^{\omega}\): finite of countably infinite sequences over \(D\)

\(D^{\omega}\) is a complete partial order where 
\(X \leq Y\) iff \(X\) is an initial segment of \(Y\)

\[ \text{Definition: Kahn networks} \]

- associated with each process \(v \in V\) with incoming edges 
  \(e_1, ..., e_p\) and outgoing edges \(e_1', ..., e_q'\)
  is a continuous **function**
  \(f_v: D_{e_1}^{\omega} \times ... \times D_{e_p}^{\omega} \rightarrow D_{e_1'}^{\omega} \times ... \times D_{e_q'}^{\omega}\)

(A function \(f: A \rightarrow B\) is **continuous** if \(f(\lim_A a) = \lim_B f(a)\))
**Semantics: Kahn networks**

A process network defines for each edge $e \in E$ a **unique** sequence $X_e$.

$X_e$ is the least fixed point of the equations

$$ (X_{e_1}, \ldots, X_{e_q}) = f_v(X_{e_1}, \ldots, X_{e_q}) $$

for all $v \in V$.

Result is independent of scheduling!

---

**Scheduling Kahn Networks**

![Diagram of Kahn Network]

Problem: run processes with finite buffer
Scheduling may be impossible

\[ \text{A (Two a's for every b)} \quad \text{B (Alternates between receiving a and b)} \]

Parks’ Scheduling Algorithm (1995)

- Set a capacity on each channel
- Block a write if the channel is full
- Repeat
  - Run until deadlock occurs
  - If there are no blocking writes → terminate
  - Among the channels that block writes, select the channel with least capacity and increase capacity until producer can fire.
Example

A
(always produces token)

C
(only consumes from A)

B
(always produces token)

D
(always consumes token)

Parks’ Scheduling Algorithm

- Whether a Kahn network can execute in bounded memory is undecidable
- Parks’ algorithm does not violate this
- It will run in bounded memory if possible, and use unbounded memory if necessary

Disadvantages:
- Requires dynamic memory allocation
- Does not guarantee minimum memory usage
- Scheduling choices may affect memory usage
- Data-dependent decisions may affect memory usage
- Relatively costly scheduling technique
- Detecting deadlock may be difficult
**Synchronous data flow (SDF)**

- Asynchronous message passing: tasks do not have to wait until output is accepted.
- Synchronous data flow: all tokens are consumed at the same time.

SDF model allows static scheduling of token production and consumption. In the general case, buffers may be needed at edges.

---

**SDF: restriction of Kahn networks**

An **SDF graph** is a tuple \((V, E, \text{cons}, \text{prod}, d)\) where

- \(V\) is a set of nodes (activities)
- \(E\) is a set of edges (buffers)
- \(\text{cons}: E \rightarrow \mathbb{N}\) number of tokens consumed
- \(\text{prod}: E \rightarrow \mathbb{N}\) number of tokens produced
- \(d: E \rightarrow \mathbb{N}\) number of initial tokens

\(d: \) “delay” (sample offset between input and output)
**Multi-rate SDF System**

- DAT-to-CD rate converter
- Converts a 44.1 kHz sampling rate to 48 kHz

![Diagram of a multi-rate SDF system with nodes labeled 1, 2, 3, 7, 8, 5, and 1, showing connections between upsampler and downsampler.]

**SDF Scheduling Algorithm**

*Lee/Messerschmitt 1987*

1. Establish **relative execution rates**
   - Generate balance equations
   - Solve for smallest positive integer vector $c$

2. Determine **periodic schedule**
   - Form an arbitrarily ordered list of all nodes in the system
   - Repeat:
     - For each node in the list, schedule it if it is runnable, trying each node once
     - If each node has been scheduled $c_n$ times, stop.
     - If no node can be scheduled, indicate deadlock.

Source: Lee/Messerschmitt, Synchronous Data Flow (1987)
Example 1

\[
\begin{align*}
A &: 0 < D \\
B &: 4 < 2 \\
C &: 1 < 7 \\
D &: 1 < 3
\end{align*}
\]

\[
\begin{pmatrix}
A \\ B \\ C \\ D
\end{pmatrix} = 
\begin{pmatrix}
1 \\ 4 \\ 1 \\ 2
\end{pmatrix}
\]

\[
\begin{pmatrix}
B \\ C \\ D \\ A
\end{pmatrix} = 
\begin{pmatrix}
-1 \\ 7 \\ -3 \\ -2
\end{pmatrix}
\]

\[
\begin{pmatrix}
C \\ B \\ A
\end{pmatrix} = 
\begin{pmatrix}
2 \\ -3 \\ 3
\end{pmatrix}
\]

c_c = 2 < c_3

c_B = 4 < c_3

c_A = 7 < c_B

\text{simplified: } ABAABABDDCC

d(\text{CA}) = 6

Example 2

\[
3c_c = c_D
\]

\[
\begin{align*}
c_A &= c_B \\
2c_c &= c_A \\
c_D &= \frac{2}{3}c_A \\
c_A &= 0, c_B = 0, c_c = 0
\end{align*}
\]

\text{inconsistent!}
\rightarrow \text{no schedule.}
Example 3

\[ c_b (c_3 - 2) - c_0 = 0 \]
\[ -2c_c + 2c_b = 0 \]
\[ 2c_b = 3c_c \]

\[ c_0 = c_A \]
\[ 3c_c = 2c_D \]

Relative rate between A and C is undefined!

Example 4

\[ c_A = c_D \]
\[ c_A = 1 \]
\[ c_D = 1 \]

Deadlock!
(missing delay)
Observations

- Consistent, connected systems have one-dimensional solution
- Disconnected systems have higher-dimensional solution
- Inconsistent systems have 0-schedule; otherwise infinite accumulation of tokens
- Systems may have multiple schedules

Summary dataflow

- Communication exclusively through FIFOs
- Kahn process networks
  - blocking read, nonblocking write
  - deterministic
  - schedulability undecidable
  - Parks’ scheduling algorithm
- SDF
  - fixed token consumption/production
  - compile-time scheduling: balance equations