Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205
Resource-efficient programming

4 Concurrency

4.1 Parallel programming

4.2 MPI

4.3 Performance metrics

4.4 Robotics middleware

4.5 Concurrency theory

4.6 Process models

Material: **ros-nodes-howto.zip**.

Norwegian University of Life Sciences

ROS 2 package creation

A ROS2 C++ **package** for compilation supported by **cmake** can be created by

e.g. **cpp_srvcli**

ros2 pkg create --build-type ament_cmake *prjname* --dependencies rclcpp …

… for the example,[1] add **example_interfaces** here

This creates a **package XML file** and an input file for cmake.

XSD metadata schema http://download.ros.org/schema/package_format3.xsd

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd"
                              schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>prjname</name>
  …
  <license>CC BY-NC-SA</license>
  <buildtool_depend>ament_cmake</buildtool_depend>
  <depend>rclcpp</depend>
   …     example:1 <depend>example_interfaces</depend>
</package>
```

package.xml

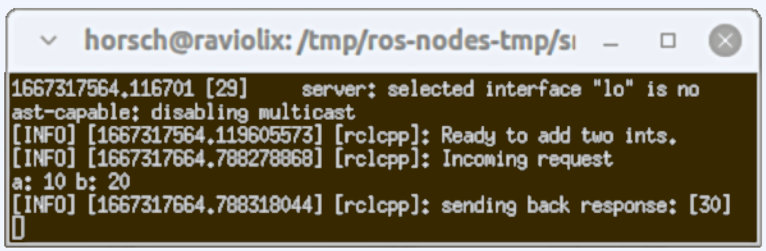
[1] http://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Service-And-Client.html

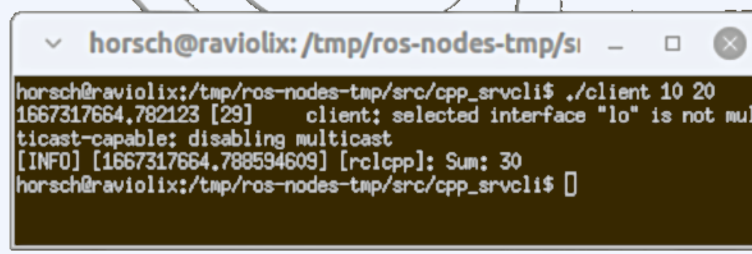
Material: **ros-nodes-howto.zip**.

Example[1]

How to test the **ros-nodes-example**:

- Compile the client and server codes using "colcon" (which calls cmake).
  - You may need to install cmake first.
- Run "server" on one terminal (or one computer in the network).
- Run "client x y" on another.
- They should interact, and the addition x+y should be performed.





Disclaimer: If you use ROS 2 for your work and it leads to a publication (or master thesis), include a citation to the reference S. Macenski *et al.*, *Science Robotics* **7**(66): eabm6074, doi:10.1126/scirobotics.abm6074, **2022**.

[1]http://docs.ros.org/en/rolling/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Service-And-Client.html

Collective communication

Send/receive is done from *one sender* process to *one recipient* process.
In a **collective communication** step, *all the MPI ranks participate* jointly.

- **Broadcast:** MPI_Bcast(buffer, count, type, root, handle)
  After the broadcast, *all processes' buffers* contain the value that used to be in the buffer of the root process. Rank 0 is often used as the root process.

- **Scatter:** MPI_Scatter(content, count, type, buffer, count, type, root, handle)
  Like broadcast, but *content* is *split (scattered) over the recipients' buffers*.

- **Reduce:** MPI_Reduce(content, buffer, count, type, *operation*, root, handle)
  Content from all the processes is *aggregated* into the buffer of the root process. For example, add up all the values (with *MPI_SUM* as *operation*).

- **Gather:** MPI_Gather(content, count, type, buffer, count, type, root, handle)
  The gather operation is the *opposite of scatter*. Split content from all processes is written into one big buffer at the root process.

Collective communication

Gathering operation (all ranks to the root rank):

- MPI_Gather(local_chunk, 3, MPI_CHAR, content, 3, MPI_CHAR, 0, ...)

Scatter operation (all ranks to the root rank):

- MPI_Reduce(local_chunk, reduced, 3, MPI_BYTE, MPI_MAX, 0, ...)

```
Scattering content[15] to local_chunk[3].
      rank 0: 'a' 'b' 'c'
      rank 1: 'd' 'e' 'f'
      rank 2: 'g' 'h' 'i'
      …
Gathering using MPI_Gather.
      rank 0: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 1: '' '' '' '' '' '' '' '' '' '' '' '' '' '' ''
      rank 2: '' '' '' '' '' '' '' '' '' '' '' '' '' '' ''
      …
Reducing local chunks into 'reduced' using MPI_Reduce with MPI_MAX.
      rank 0: 'm' 'n' 'o'
      rank 1: '' '' ''
      rank 2: '' '' ''
      …
```

Name	Meaning
MPI_MAX	**maximum**
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value, location
MPI_MINLOC	min value, location

Example file: collective-communication.zip

Collective communication

Gathering operation (all ranks to all ranks):

- **MPI_Allgather(local_chunk, 3, MPI_CHAR, content, 3, MPI_CHAR, …)**

Scatter operation (all ranks to all ranks):

- **MPI_Allreduce(local_chunk, reduced, 3, MPI_BYTE, MPI_MAX, …)**

```
Scattering content[15] to local_chunk[3].
      rank 0: 'a' 'b' 'c'
      rank 1: 'd' 'e' 'f'
      rank 2: 'g' 'h' 'i'
      …
Gathering using MPI_Allgather.
      rank 0: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 1: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      rank 2: 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o'
      …
Reducing local chunks into 'reduced' using MPI_Allreduce with MPI_MAX.
      rank 0: 'm' 'n' 'o'
      rank 1: 'm' 'n' 'o'
      rank 2: 'm' 'n' 'o'
      …
```

Name	Meaning
MPI_MAX	**maximum**
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MAXLOC	max value, location
MPI_MINLOC	min value, location

Example file: collective-communication.zip

Collective communication

What MPI operation(s) would we use for the following?

- There are n processes (ranks).
- Each rank generates k = 65536 floating-point random numbers between 0 and 1.
- Now there are $k \cdot n$ random numbers. We would like all of them together to become a **unit vector** $\mathbf{x} = (x_0, \ldots, x_{kn-1})$ such that $\mathbf{x}^2 = 1$.
- We definitely don't want to send all the values to all processes, especially if k becomes even greater, but do this as efficiently as possible.

Discussed MPI operations

MPI_Send	MPI_Isend
MPI_Recv	MPI_Irecv
MPI_Wait	MPI_Test
MPI_Bcast	MPI_Ibcast
MPI_Scatter	MPI_Iscatter
MPI_Reduce	MPI_Ireduce
MPI_Gather	MPI_Igather
MPI_Allgather	MPI_Iallgather
MPI_Allreduce	MPI_Iallreduce

(See **unit-vector-incomplete.cpp**, where the implementation is missing.)

Performance in time and in space

Time, in theory:

- Number of steps executed by a *Turing machine*
  (or similar formalisms, such as *random-access machines*)
- Number of statements to be executed when going through the code

Time, in practice:

- CPU time, *i.e.*, number of cores × measured runtime of the program

Space, in theory:

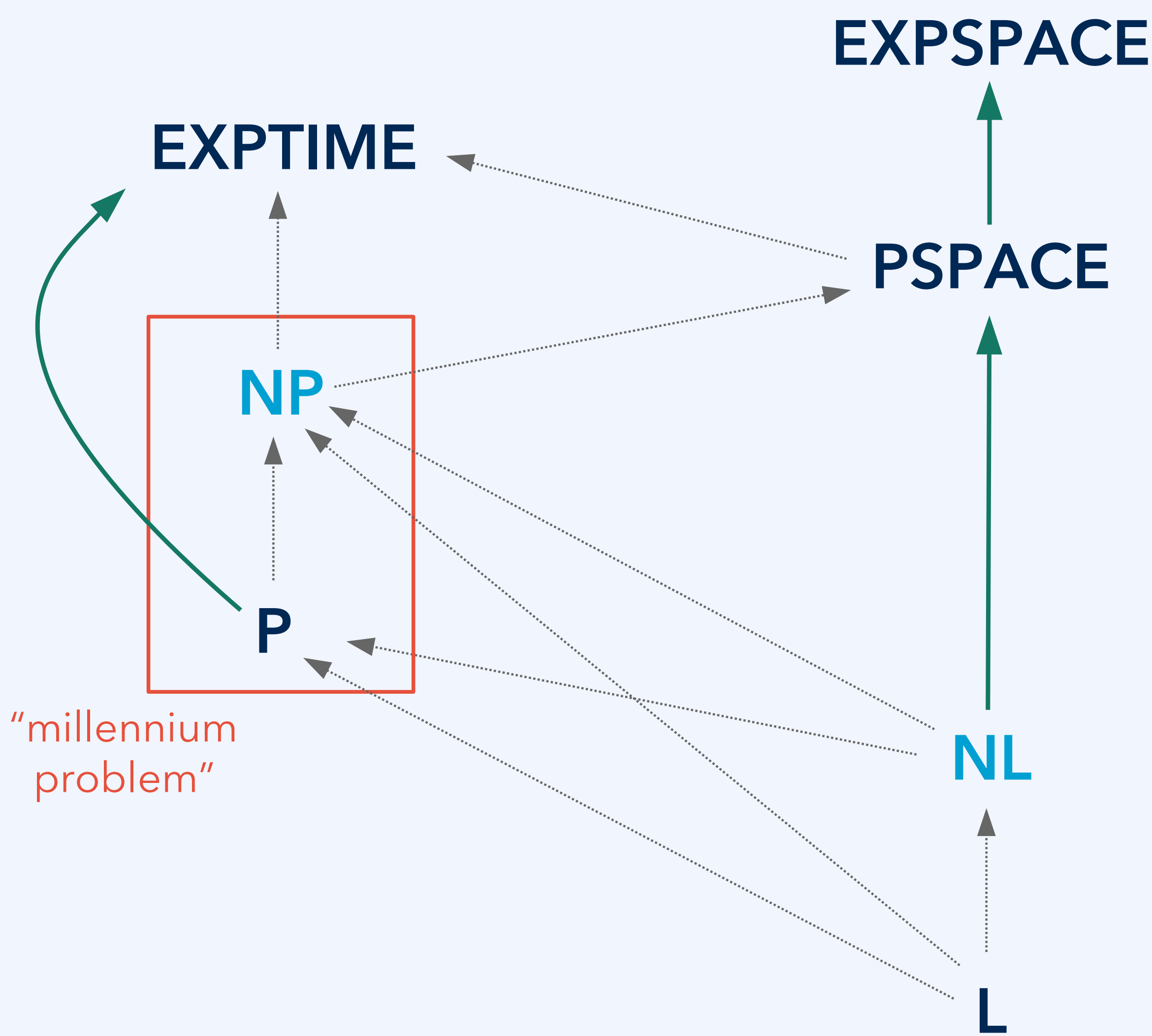
- Legth of tape used by a *Turing machine*
  (or number of registers used by a *random-access machine*)
- Number of elementary variables, or their total size in bytes, in the code

Space, in practice:

- Actual memory use measured during program execution

In **complexity theory**, the theoretical metrics are used to define computational **complexity classes**, such as DTIME($f(n)$) and DSPACE($f(n)$) for deterministic $O(f(n))$ time and space, respectively, as function of the problem size n.

Hierarchy of computational complexity classes



Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

4 Concurrency

4.5 Concurrency theory

4.6 Process models

INF205

11th March 2025

States and transitions (events)

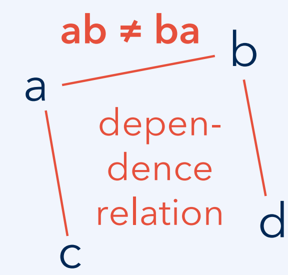
Terminology related to concurrency is often taken from the domain of **discrete event systems** (for example, *finite automata*). Adopting such an approach:

- A system can be in any of a finite number of **states**.
- Events, or **transitions** between states, are thought of as instantaneous.
- A concurrent **process** is a (partially) temporally ordered set of events.
- Two events or transitions t and t' can be …
  - … **concurrent** whenever they are both enabled (*i.e.*, both can occur), one does not inhibit the other, and $t{\cdot}t'$ has the same outcome as $t'{\cdot}t$; in other words, they are concurrent if we don't say which comes first.
  - … **causally dependent** if they both occur, and it is important to say which comes first, either because only one order is possible or because it will have an impact on the outcome.
- **Limitation:** This model cannot make two transitions strictly synchronous.

Traces:[1] Partially ordered sets of events

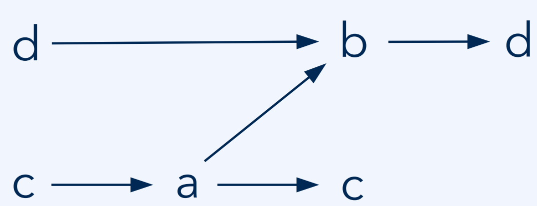
Dependence/independence between actions & events in an enterprise system:

a) Updated raw sensory data ingested into knowledge base
b) Data analysis on raw sensory data, creating aggregated data
c) Read access to raw sensory data by a user
d) Read access to aggregated data by a user



Events that are **dependent** can *never* occur *concurrently*.
Events are independent if they are **commutative**: bc = cb.

In a particular execution or process, if it is unsubstantial in what order two events occur, they are **concurrent**: Below, *e.g.*, the first and second c-d pairs:



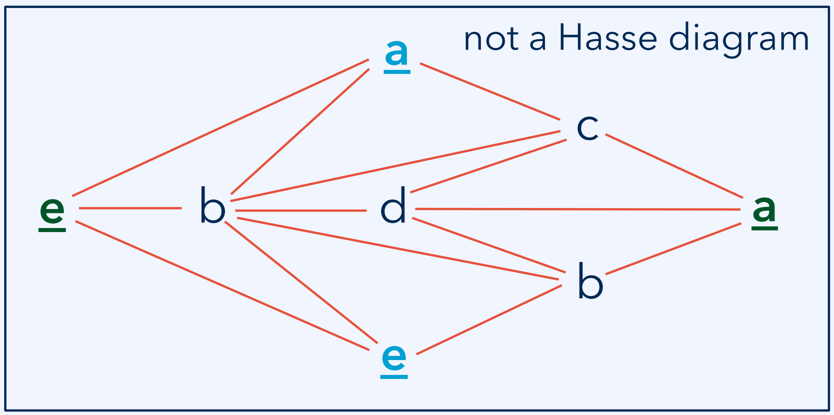
Hasse diagram for the *trace*[1]

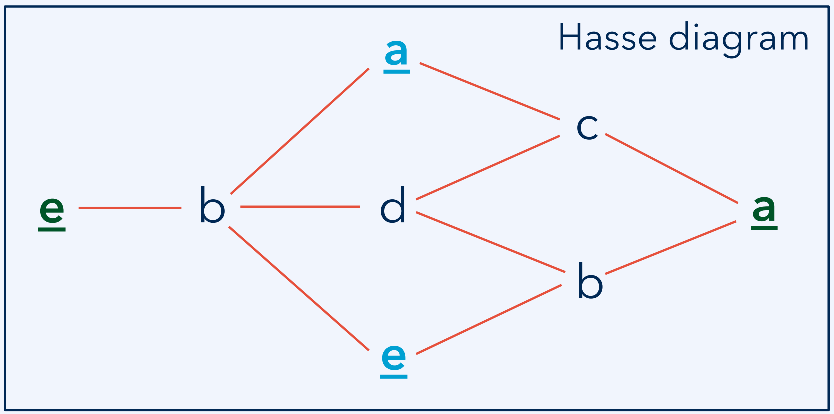
cacdbd = cdacbd = dcabdc = …

[3]Also called **Mazurkiewicz traces** after Polish mathematician Antoni Mazurkiewicz.

Diagrams for partially ordered sets

By convention, **Hasse diagrams** are often used to denote causal dependency of events. These diagrams remove *any **indirect** or **redundant** dependencies*:





Two events are **directly or indirectly causally dependent** if one is specified to occur (conclude) before the other occurs (begins). Above: e and a are indirectly dependent.

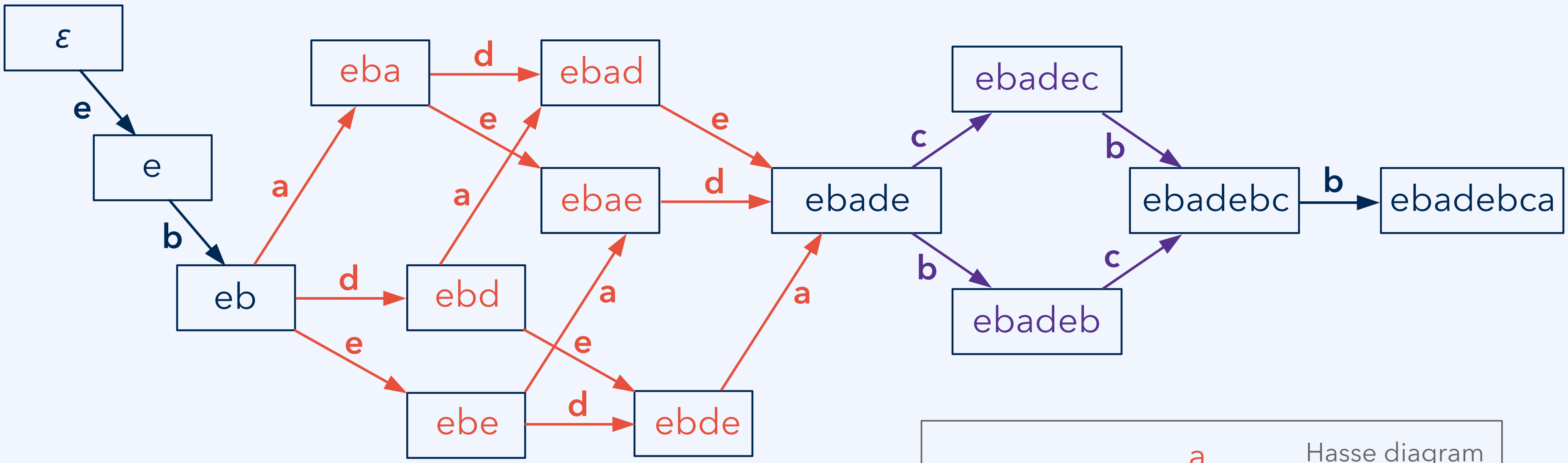
Events are **concurrent** if they are not directly or indirectly causally dependent – it does not matter which occurs first. Above: e and a are concurrent.

Attention

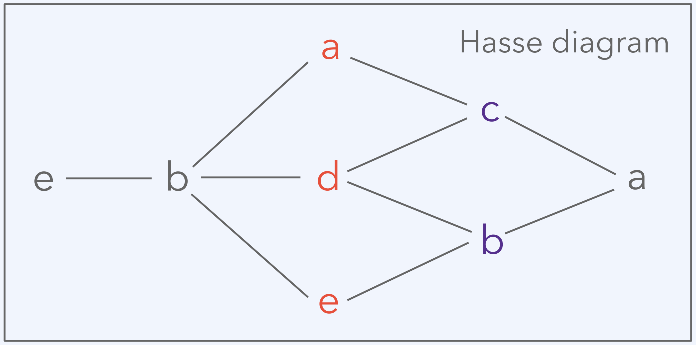
This notation only shows the **transitions** (events). The **states** (configurations) of the system are not shown.

State-transition diagrams

In a **state-transition diagram**, *two concurrent transitions* give rise to *"diamond" patterns*. *More than two concurrent transitions* lead to *(hyper-)cube patterns*:



Observation: With n concurrent events, we obtain 2^n states, making it prohibitively expensive to explore the whole state space. ("**State explosion problem**".)



Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

4 Concurrency

4.5 Concurrency theory

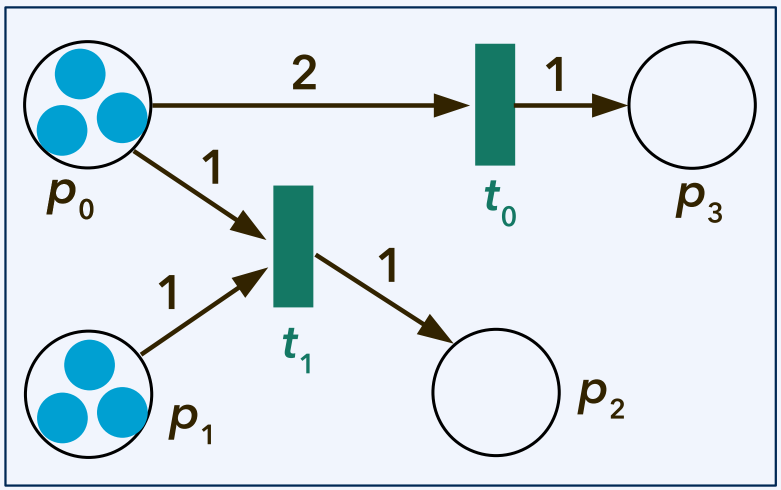
<u>4.6 Process models</u>

INF205

11th March 2025

Petri nets

Components of a Petri net: **places** **transitions** **tokens** **arc**

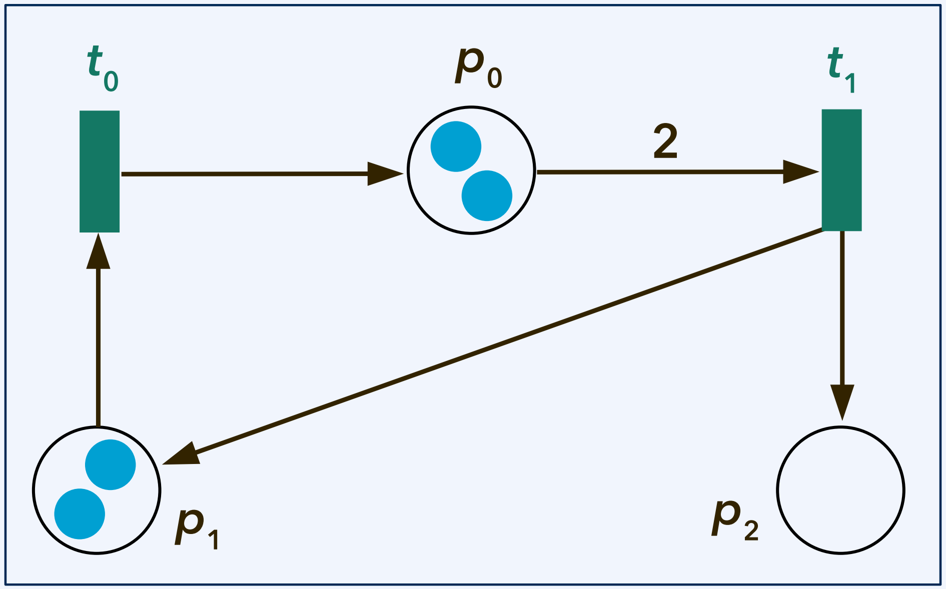


Semantics of this net:

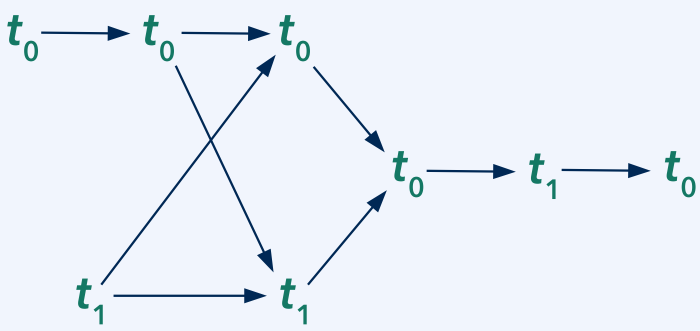
Transition t_0 can only be **fired** if place p_0 contains at least two tokens. Firing t_0 will take away two tokens from p_0 and add one token to p_3.

Transition t_1 can only be fired if both p_0 and p_1 each contain at least one token. It removes one token from each, and adds one token to place p_2.

Petri nets: Example



firing sequence

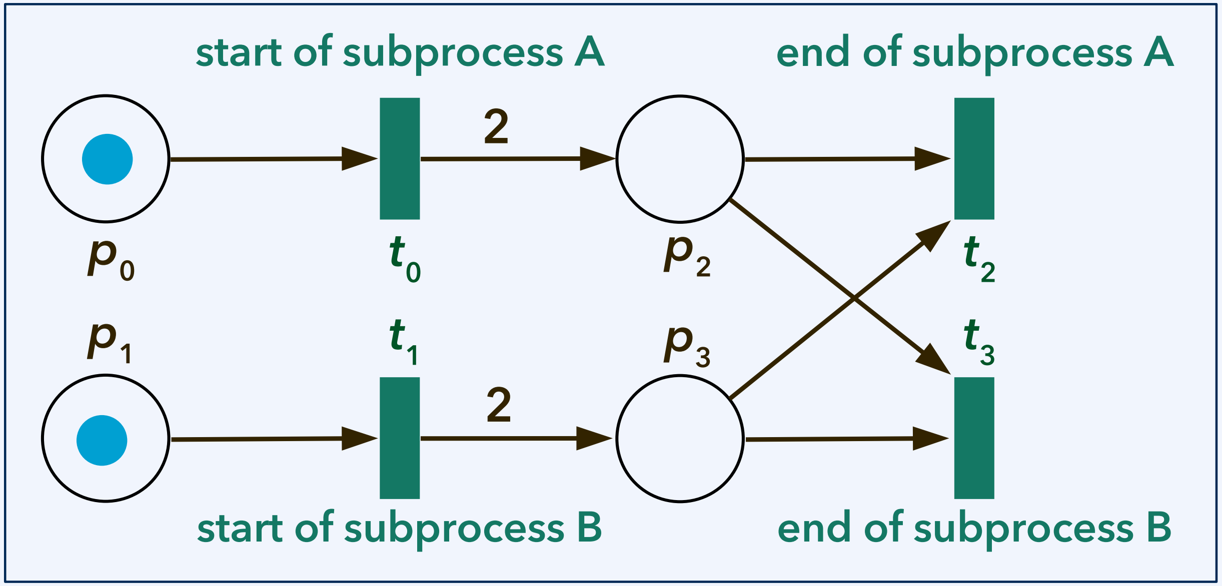


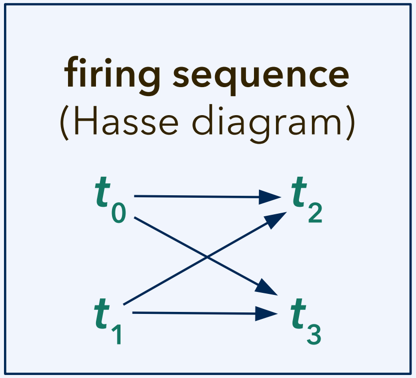
- Transitions can be fired in the following order: $t_0t_0t1t_0t1t_0t1t_0$, $t_0t_0t1t1t_0t_0t1t_0$, $t_0t1t_0t_0t1t_0t1t_0$, $t_0t1t_0t1t_0t_0t1t_0$, $t1t_0t_0t_0t1t_0t1t_0$, and $t1t_0t_0t1t_0t_0t1t_0$. At that point, respectively, a deadlock is reached.
- The net is bounded: There is a limit to the number of tokens per place.

Petri nets and synchronous processes

Two subprocesses are synchronous (also, "coupled") if it is specified that they must overlap temporally, *i.e.*, they must at least in part run at the same time.

Petri net representing two synchronous subprocesses A and B

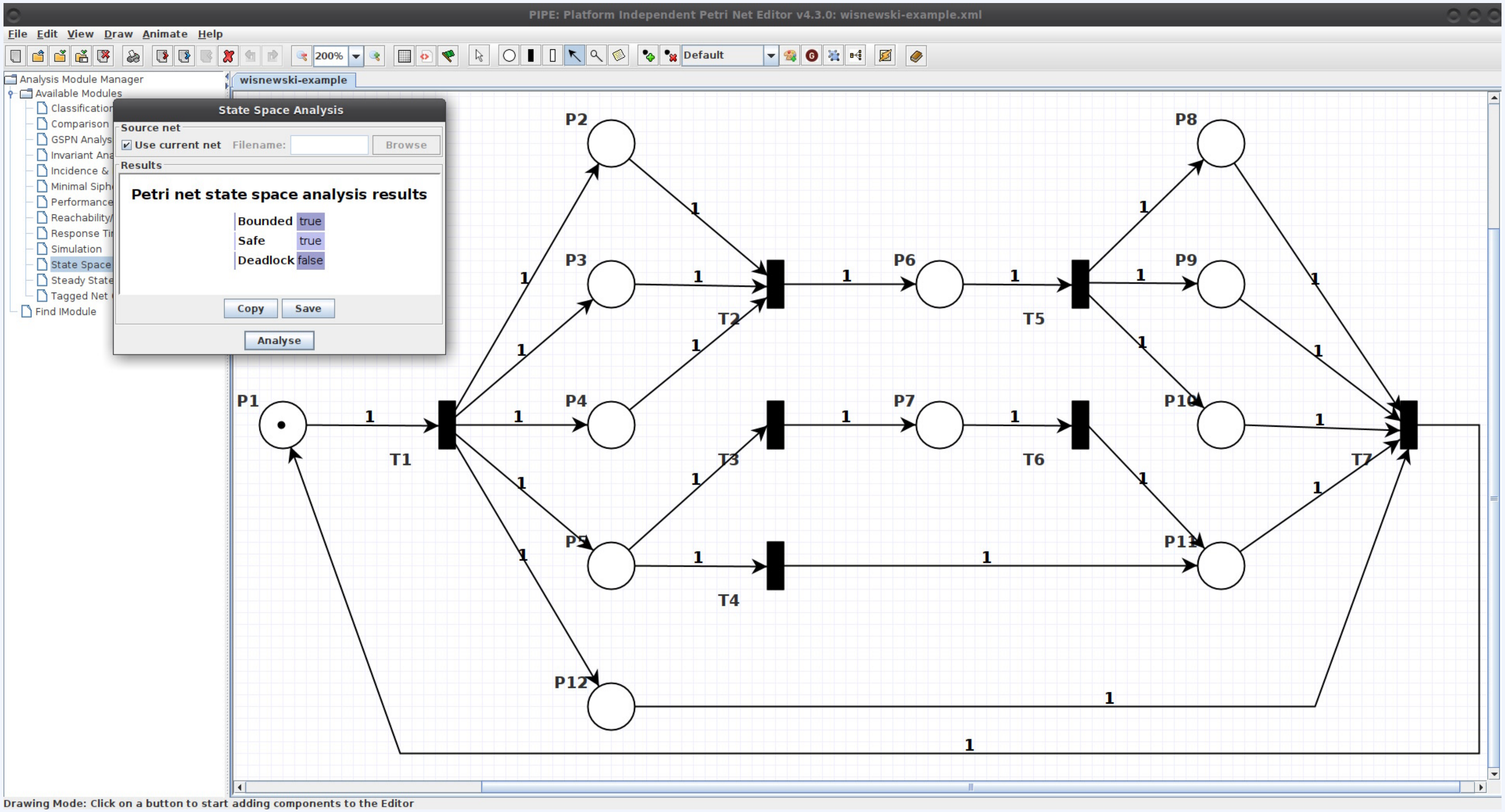




Note: **Synchronicity** ("**coupling**" – subprocesses must overlap) vs. **direct causal dependency** ("**linking**" – may not overlap) vs. **concurrency** (order unspecified).

Petri net editor

PIPE tool for editing/simulating Petri nets: http://pipe2.sourceforge.net/



Norges miljø- og biovitenskapelige universitet

Institutt for datavitenskap

Digitalisering på Ås

INF205

Resource-efficient programming

4 Concurrency

4.1 Parallel programming

4.2 MPI

4.3 Performance metrics

4.4 Robotics middleware

4.5 Concurrency theory

4.6 Process models