A Tool Suite for Concurrent Programming

Marian V. Iordache
School of Engineering and Eng. Tech.
LeTourneau University

October 21, 2011
CS Seminar
LeTourneau University

... Blessed be the name of God for ever and ever: for wisdom and might are his: ... he giveth wisdom unto the wise, and knowledge to them that know understanding (Daniel 2:20-21).

Then answered Jesus and said unto them, Verily, verily, I say unto you, The Son can do nothing of himself, but what he seeth the Father do: for what things soever he doeth, these also doeth the Son likewise (John 5:19).

All things were made by him; and without him was not any thing made that was made (John 1:3).

And he is before all things, and by him all things consist (Colossians 1:17).

... Christ *came*, who is over all, God blessed for ever. Amen (Romans 9:5).

Motivation

- Concurrent programming is difficult because of shared resources.
- These can cause *race conditions*.
- Race conditions denote the situations in which the result depends on the order of execution of concurrent operations.

PROCESS #1

$$\begin{array}{lcl} \vdots & & \\ (1) \quad u & = & x \\ (2) \quad u & = & u + 10 \\ (3) \quad x & = & u \\ \vdots & & \end{array}$$

PROCESS #2

$$\begin{array}{lcl} \vdots & & \\ (a) \quad v & = & x \\ (b) \quad v & = & v + 20 \\ (c) \quad x & = & v \\ \vdots & & \end{array}$$

- Initial value of shared variable x is $x = 30$.
- Correct result: $x = 60$.
- Possible results:
  - $x = 40$ for operation sequence (1), (a), (2), (b), (c), (3).
  - $x = 50$ for operation sequence (1), (a), (2), (3), (b), (c).
  - $x = 60$ for operation sequence (1), (2), (3), (a), (b), (c).

- *Locks* can be used to ensure that critical operation sequences are not interrupted.
  - Generally, it is difficult to find the best way to use locks in a program.
  - Improper use of locks may result in
    - deadlocks;
    - sequential execution of parallel code.
- Concurrent programs are hard to debug.

A Concurrency Tool Suite

A Concurrency Tool Suite (ACTS): software that generates automatically the concurrency control code.

Based on an input specification, the software generates files containing user code, embedded concurrency control code, and supervision code.

The output files are written in C.

Compared to other software for concurrent programming:

- Permits general interprocess synchronizations.
- Permits concurrency constraints that can be described in terms of linear inequalities (such as generalized mutual exclusion and fairness constraints).
- Provides a deadlock prevention tool. If the specification allows reaching deadlock states, this tool can be used to prevent deadlock.
- ...
- It is free open-source software.

Specifications

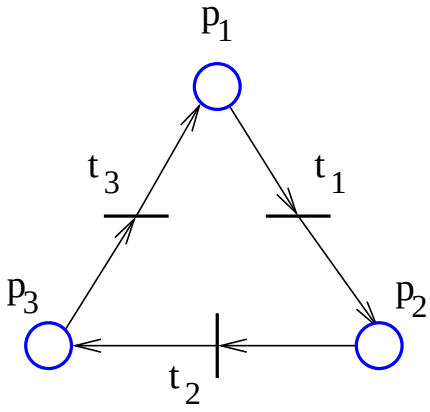
The specification describes the concurrent entities as state machines.

The specification is in terms of sets of concurrent entities.

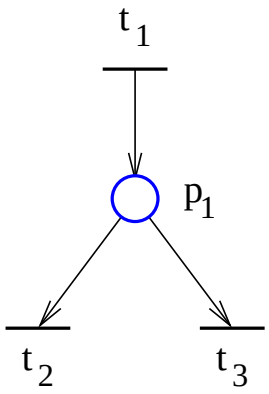
A set may have any number of elements (zero included). This number is not fixed and may change during the execution of the generated program.

User code is associated with each state of a state machine.

```
1. process prc {  /* set of processes */
2.    places:  p1 p2 p3
3.    transitions:  t1 t2 t3
4.    (p1, t1, p2); (p2, t2, p3); (p3, t3, p1)
5. }
```




```
1. thread thr {  /* set of threads */
2.    places:  p1
3.    transitions:  t1 t2 t3
4.    (t1, p1); (p1, t2);
5.    (p1, t3) { j >= 1; }  /* conditional tran.  */
6. }
```



Each state of a state machine can be associated with user code.

```
1. prc.p2 {  /* User code for state 'p2' of any process in the set 'prc' */
2.     i = i + 2;
3.     v = read_data(f, i);
4. }
```

The initial number of elements of a set and the initial state of each element can be specified:

```
initialize: prc(p1:1, p2:2)  /* In the beginning 'prc' will have three processes: one in the state 'p1' and two in the state 'p2'. */
```

Synchronization:

```
sync   prc.t1   thr.t3   gr.t2  /* requests that all transitions in the list be synchronized. */
```

Safety constraints:

```
prc.p2 + thr.p1 <= 1  /* mutual exclusion: the number of processes 'prc' in the state
'p2' plus the number of threads 'thr' in the state 'p1' must not exceed 1. */

2*prc.p1 - 3*thr.p1 <= 5  /* any linear inequality may be included */
```

Liveness:

```
live: prc.t1 gr.t0  /* all transitions in the list must be "live", that is, all potential deadlocks
affecting them must be eliminated. */
```

See example and documentation directories for other features.

Data Parallelism: natural, specifications are in terms of sets of threads/processes.

Fork/Join: Synchronization involving a source/sink transition of the forked/joined process or thread.

Barrier: Implemented with a synchronization instruction.

Barriers have one limitation:

- A synchronization involving two terms synchronizes some element of the first set with some element of the second set. There is no distinction between the elements of a set.

- If a specific element of set A should be synchronized with another specific element of set B, it is necessary to write the specs such that no other elements of A/B are available at the time of the synchronization.

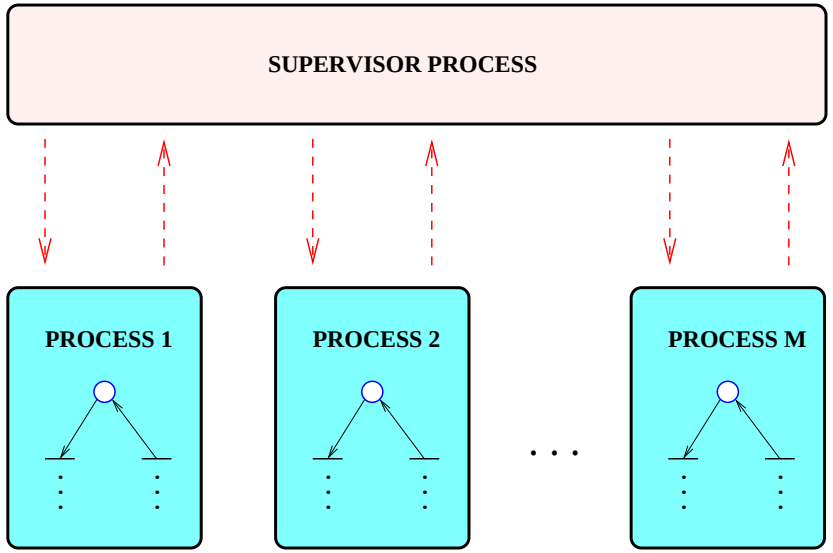
Code Generation

The software generates C code for each specified thread or process set. It contains the user code with embedded supervision code.

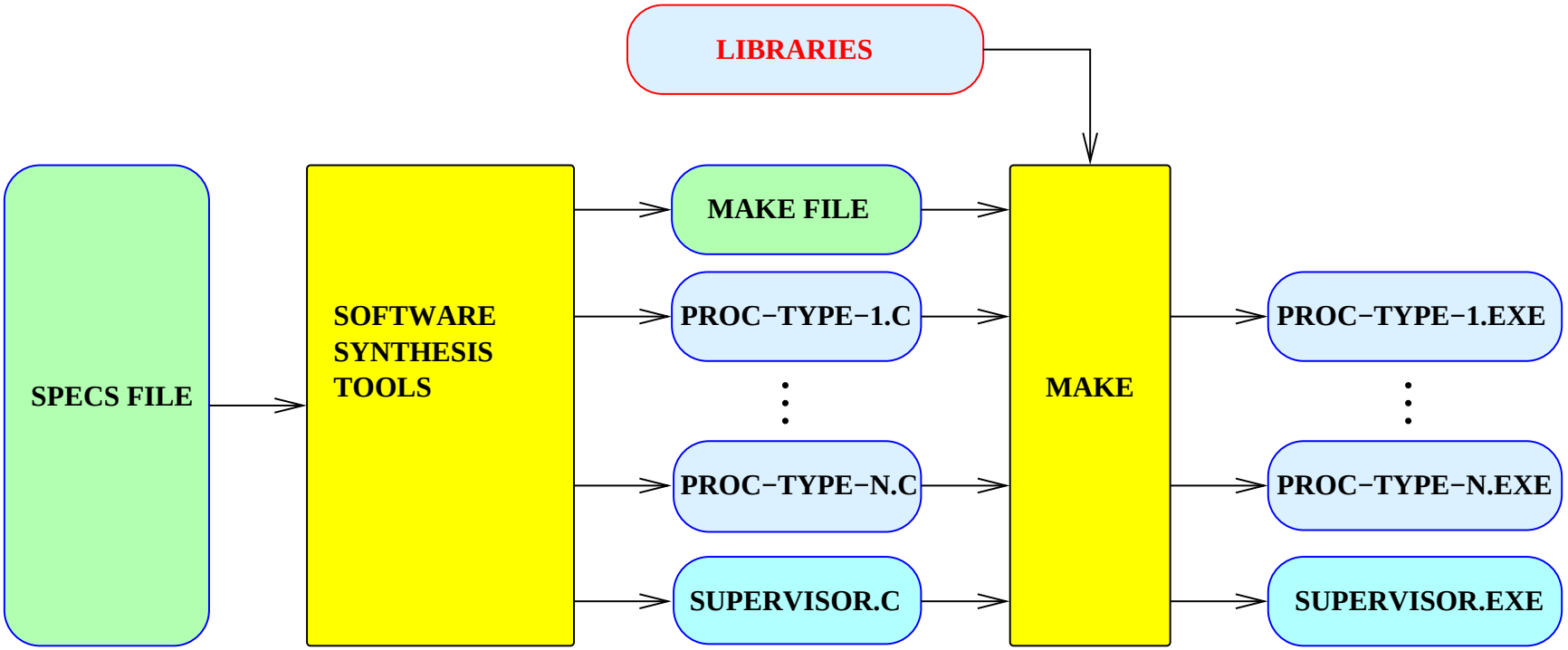
The software generates also the C code of a supervisor process.

Why a supervisor process and not locks? So that more general coordination constraints can be implemented.

The supervisor communicates with the specified threads and processes by means of unnamed pipes.

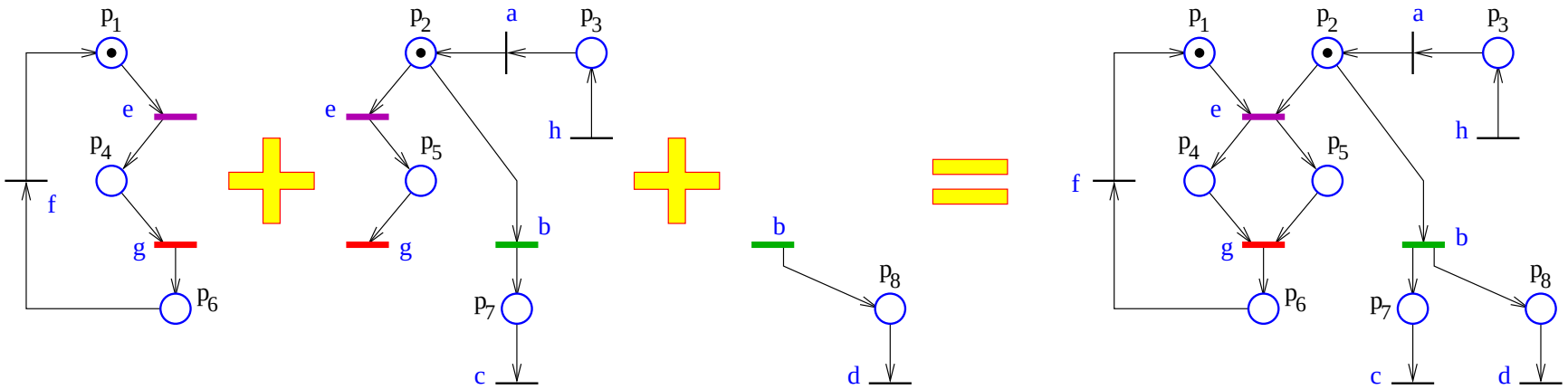


Code Generation



Note: Thread source files are compiled and linked together with the supervisor file.

State machines synchronized on transitions make up a Petri net.



TYPE A TYPE B TYPE C COMPOSITION

Petri nets model concurrent systems.

Petri net models have been used in: *distributed algorithms*, *flexible manufacturing*, *program specification*, *communication protocols*, and others.

An ***ordinary* Petri net structure** is a tuple $\mathcal{N} = (P, T, F)$ where:

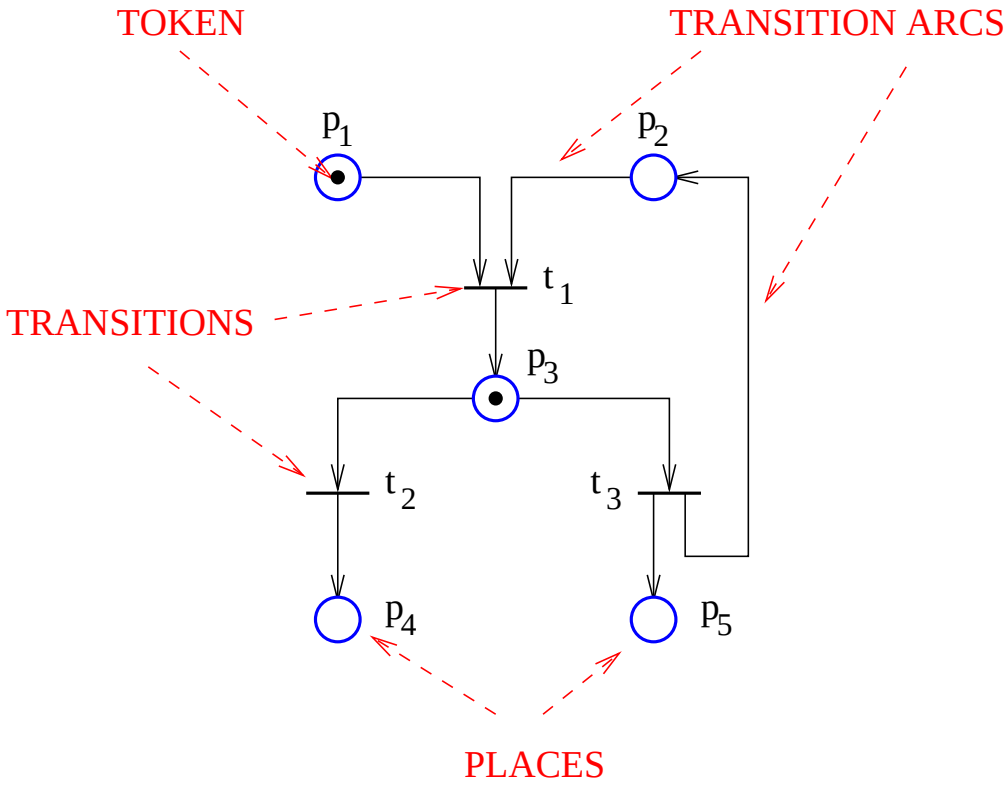
P - is the **set of places**

T - is the **set of transitions**

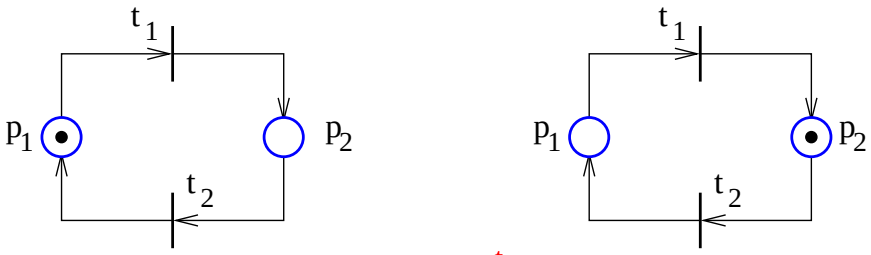
F - is the **set of transition arcs**

A **marking** μ is a vector containing the number of tokens for each place.

A Petri net structure $\mathcal{N}$ of **initial marking** μ_0 is denoted as $(\mathcal{N}, \mu_0)$.



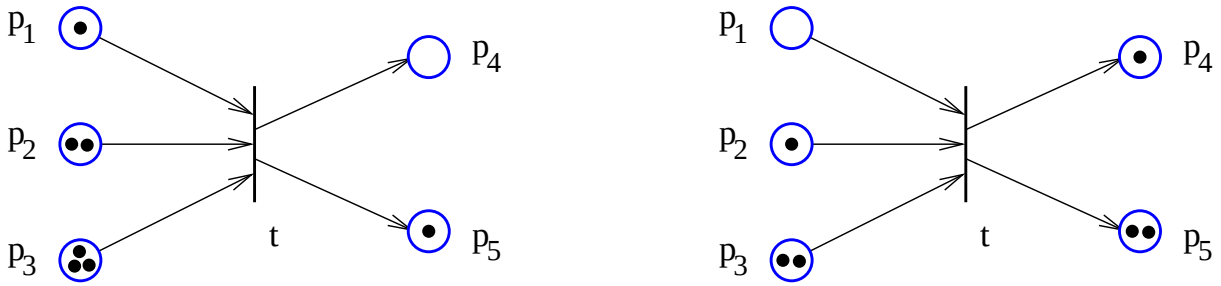
t_1 – fires, $\mu_1 \xrightarrow{t_1} \mu_2$



t_2 – fires, $\mu_2 \xrightarrow{t_2} \mu_1$

$\mu_1 = [1\ 0]^T$ $\mu_2 = [0\ 1]^T$

t – fires, $\mu_1 \xrightarrow{t} \mu_2$



t – enabled t – not enabled

$\mu_1 = [1\ 2\ 3\ 0\ 1]^T$ $\mu_2 = [0\ 1\ 2\ 1\ 2]^T$

A **supervisor** enforces constraints on the operation of a Petri net.

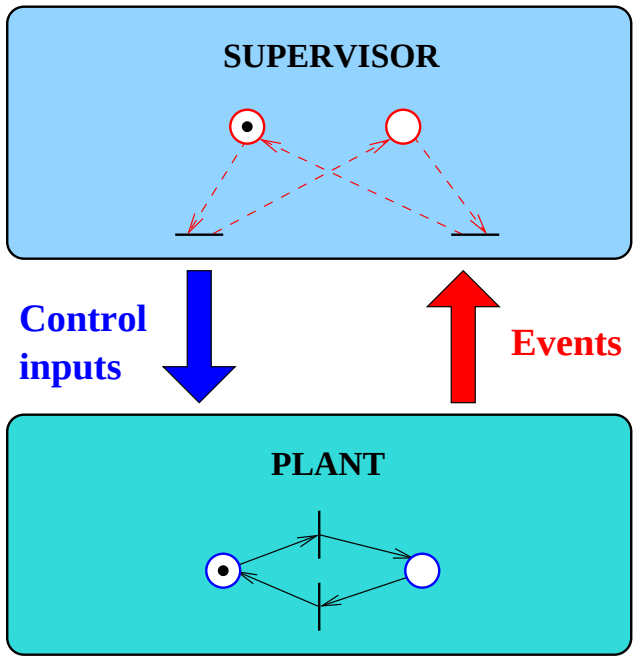
A transition is **uncontrollable** if a supervisor cannot inhibit its firing.

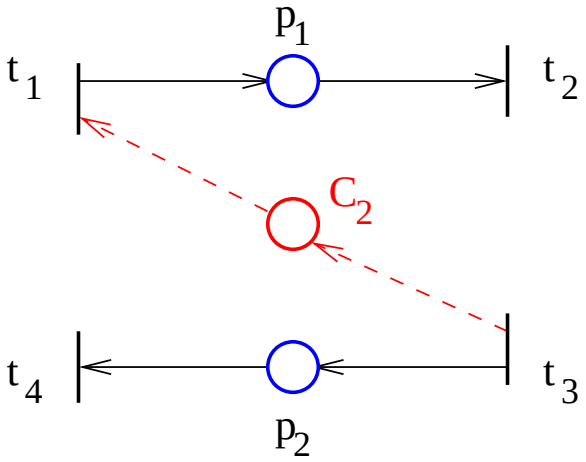
A transition is **unobservable** if a supervisor cannot observe its firing.

Given a plant Petri net and a specification on the operation of the plant Petri net, design a supervisor represented also as a Petri net, so that the plant Petri net satisfies the specification when connected together with the supervisor in closed loop.

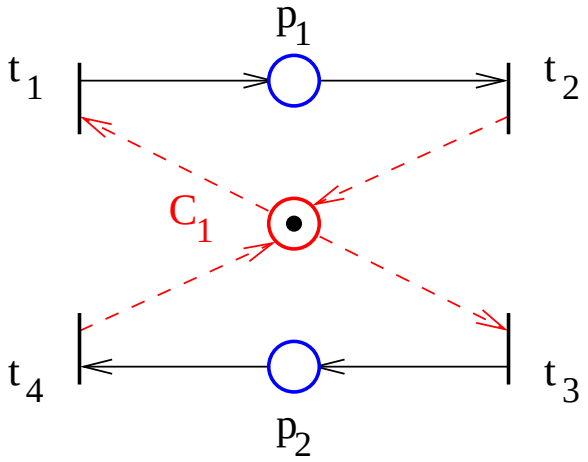
Plant + Supervisor = Closed-loop

Control places: the places of the supervisor.



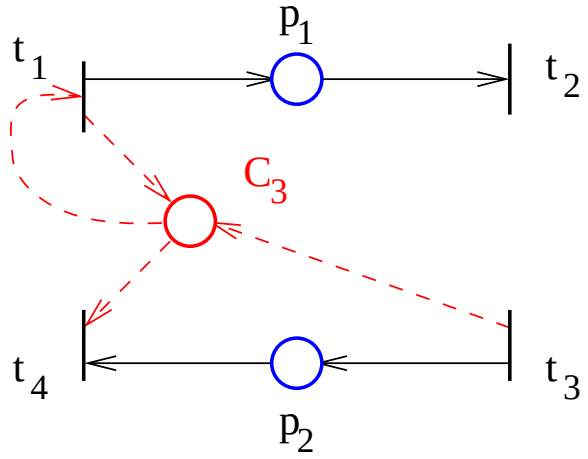


$\mu_1 + \mu_2 \leq 1$



$v_1 - v_3 \leq 0$ (v_i: how many times t_i has fired.)

$q_1 \leq \mu_2$

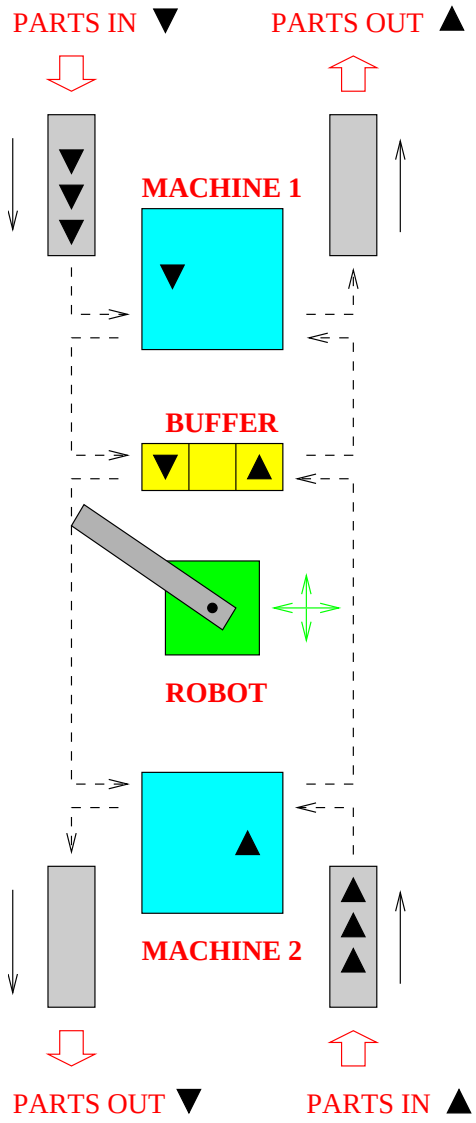


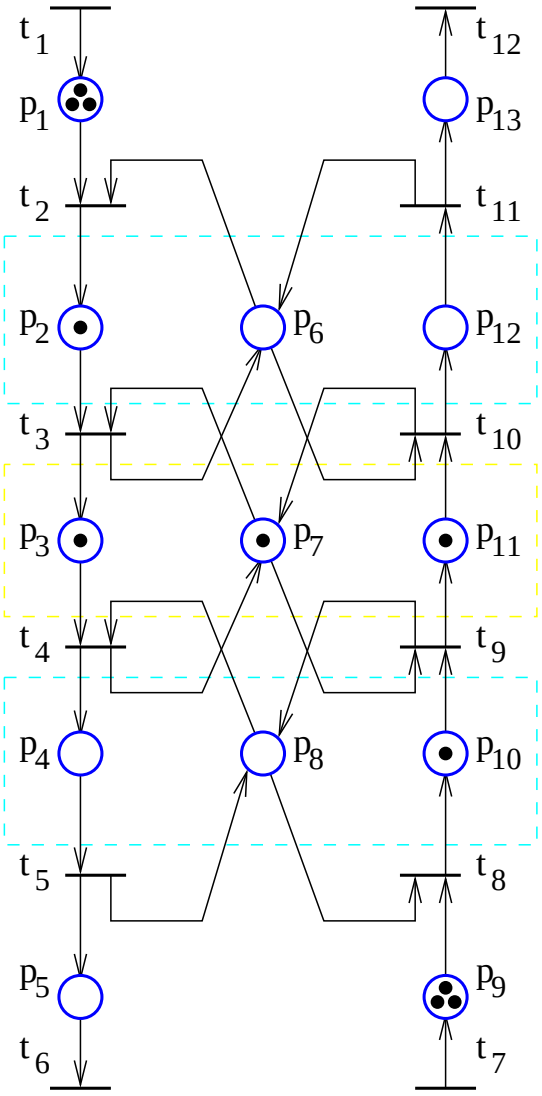
Let's illustrate deadlock prevention on a flexible manufacturing example.

Each machine can only process one part at a time.

The robot moves the parts.

The buffer is the only storage space available to the robot.

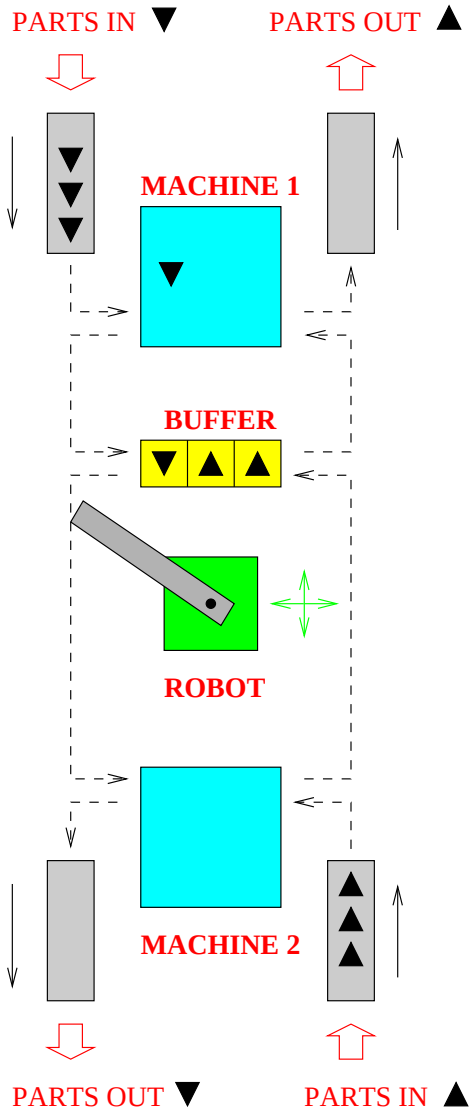


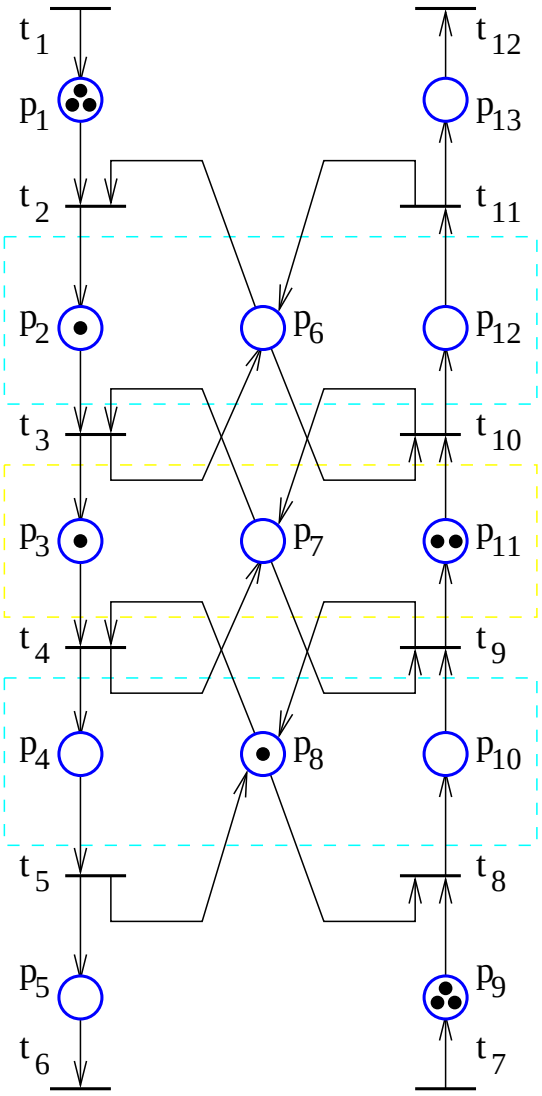


Each machine can only process one part at a time.

The robot moves the parts.

The buffer is the only storage space available to the robot.

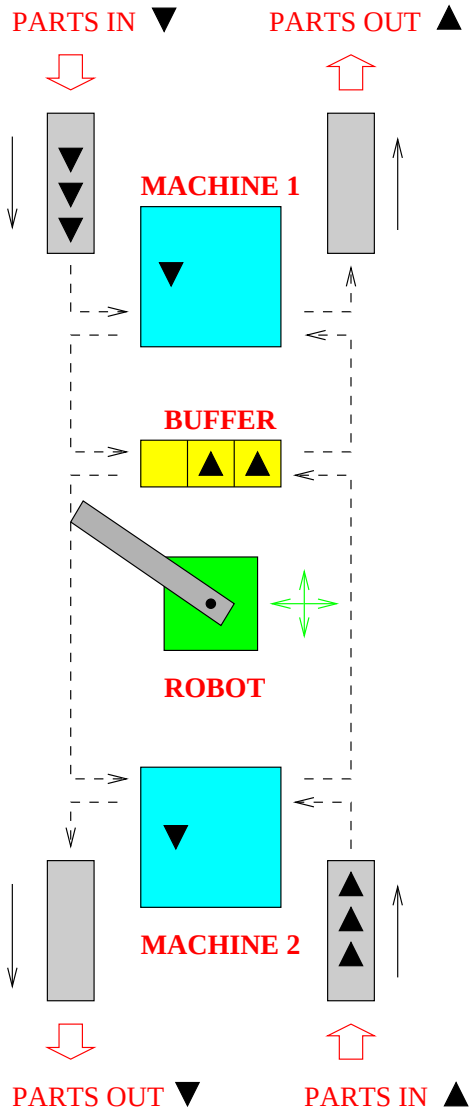


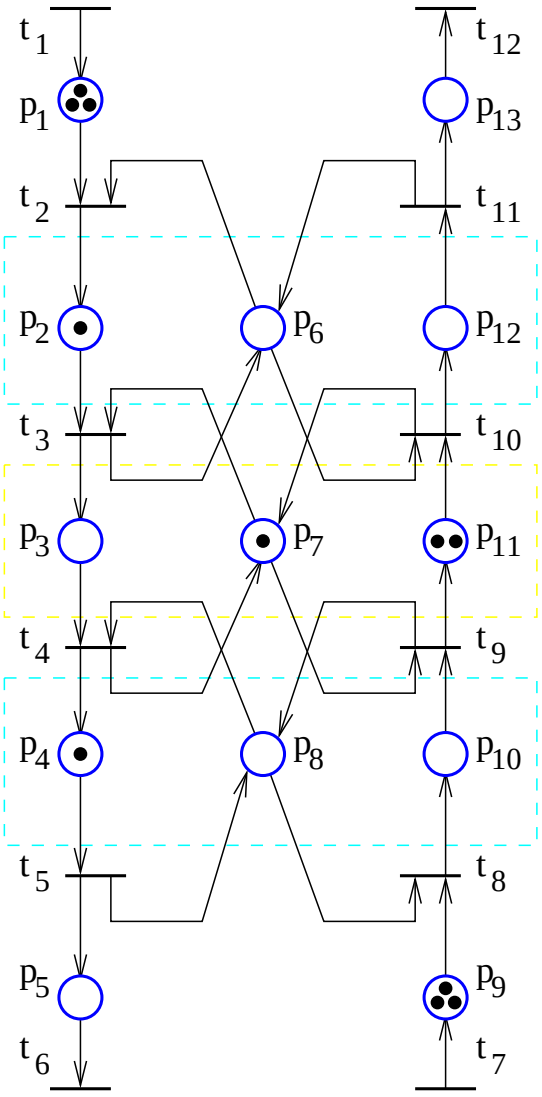


Each machine can only process one part at a time.

The robot moves the parts.

The buffer is the only storage space available to the robot.

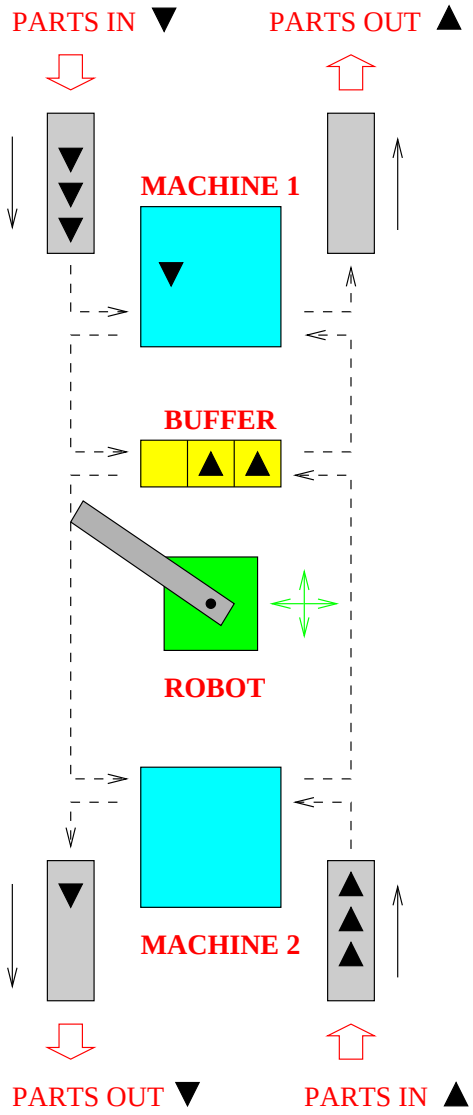


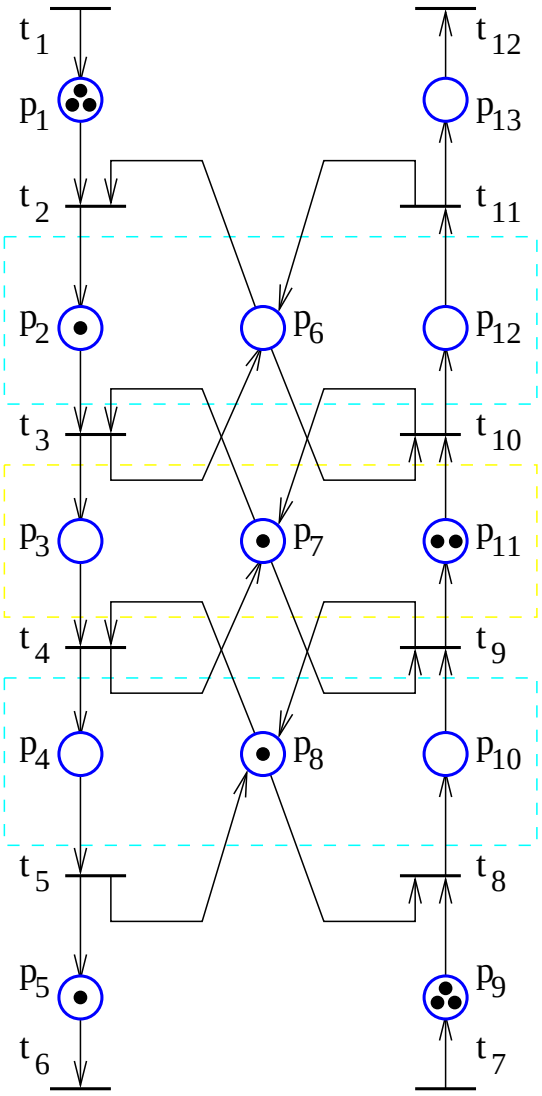


Each machine can only process one part at a time.

The robot moves the parts.

The buffer is the only storage space available to the robot.

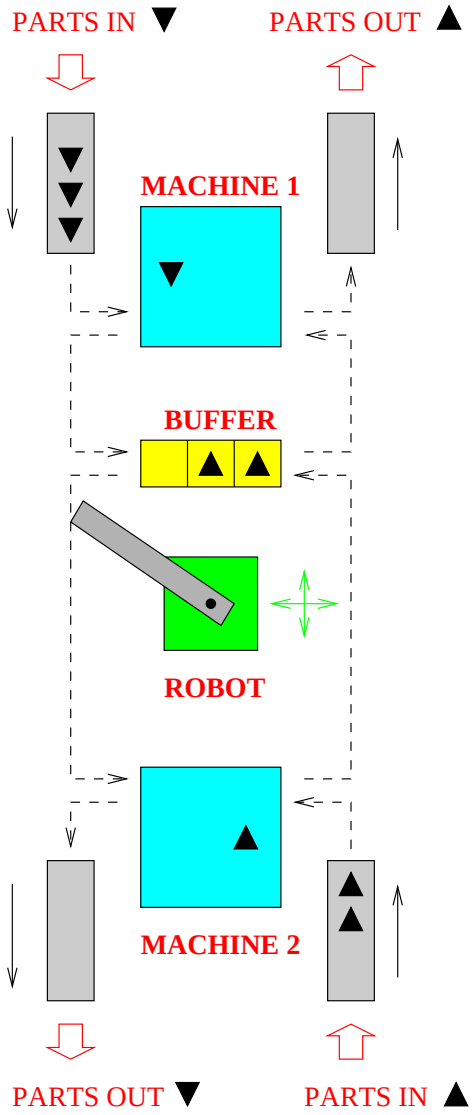


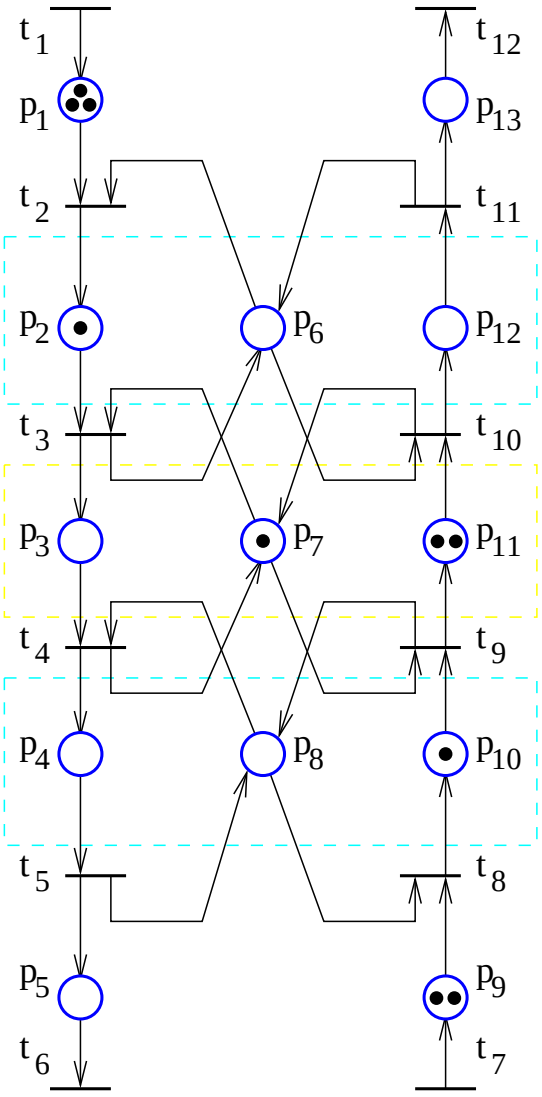


Each machine can only process one part at a time.

The robot moves the parts.

The buffer is the only storage space available to the robot.

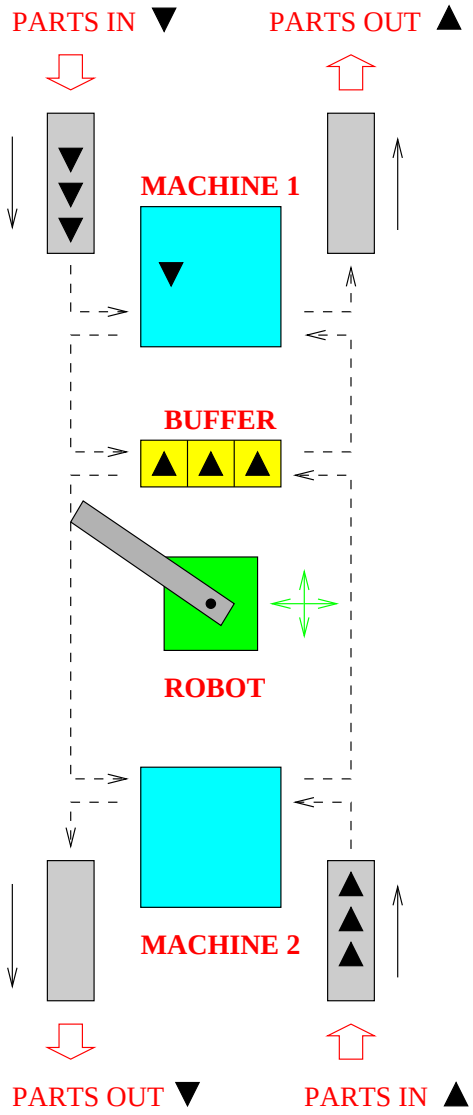


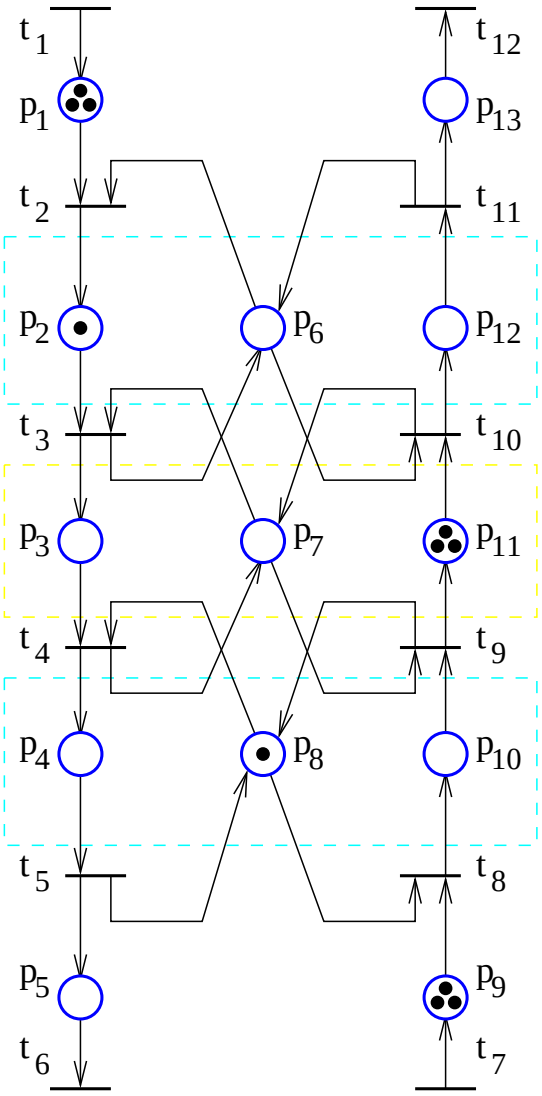


Each machine can only process one part at a time.

The robot moves the parts.

The buffer is the only storage space available to the robot.

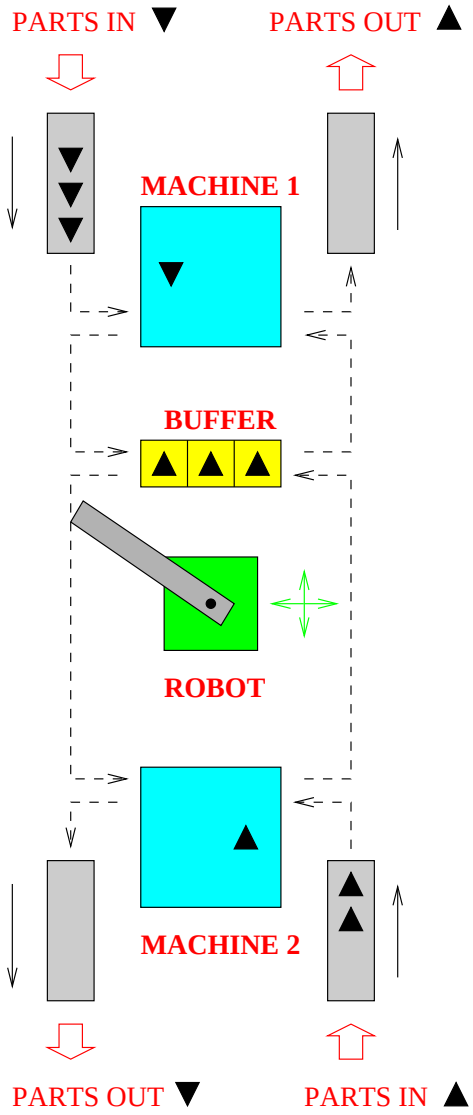


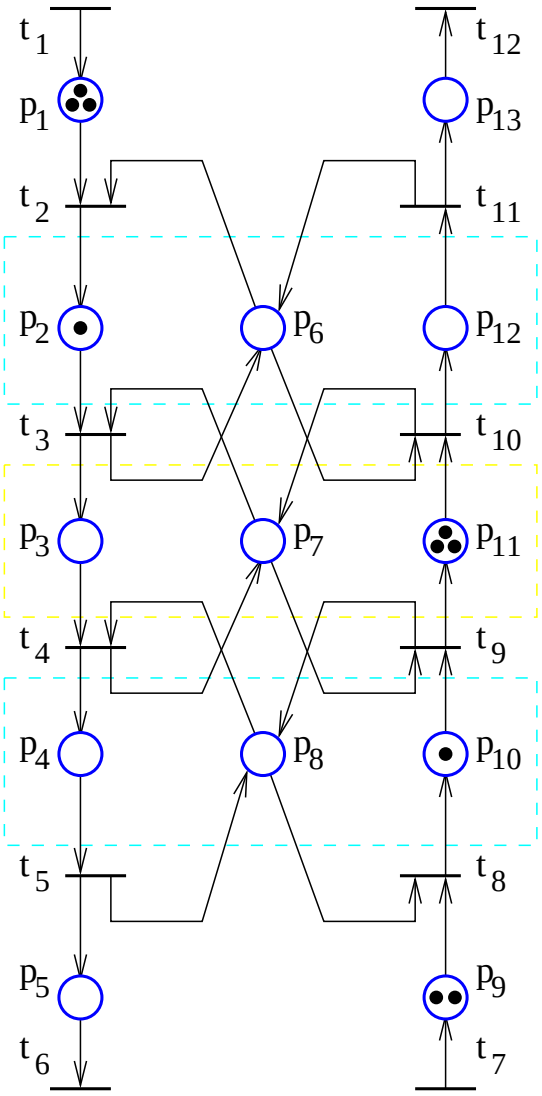


Each machine can only process one part at a time.

The robot moves the parts.

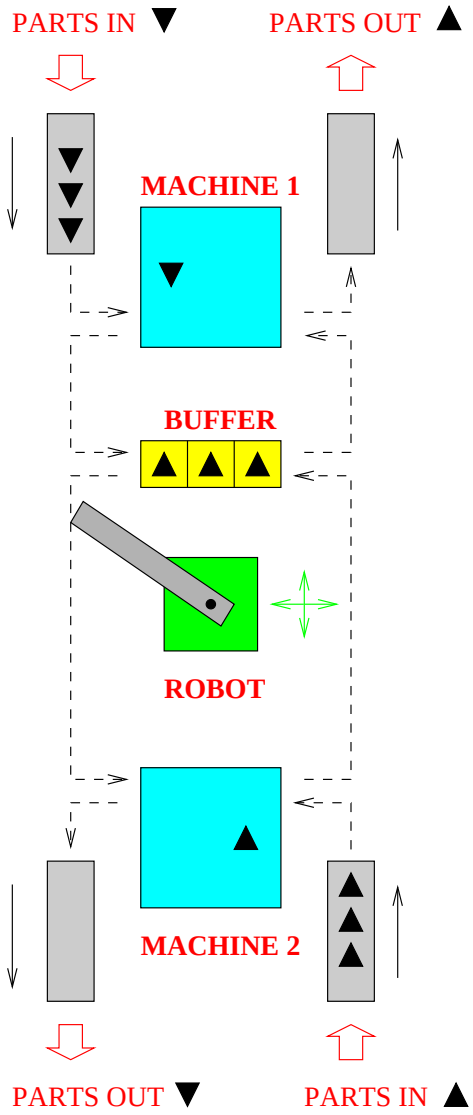
The buffer is the only storage space available to the robot.

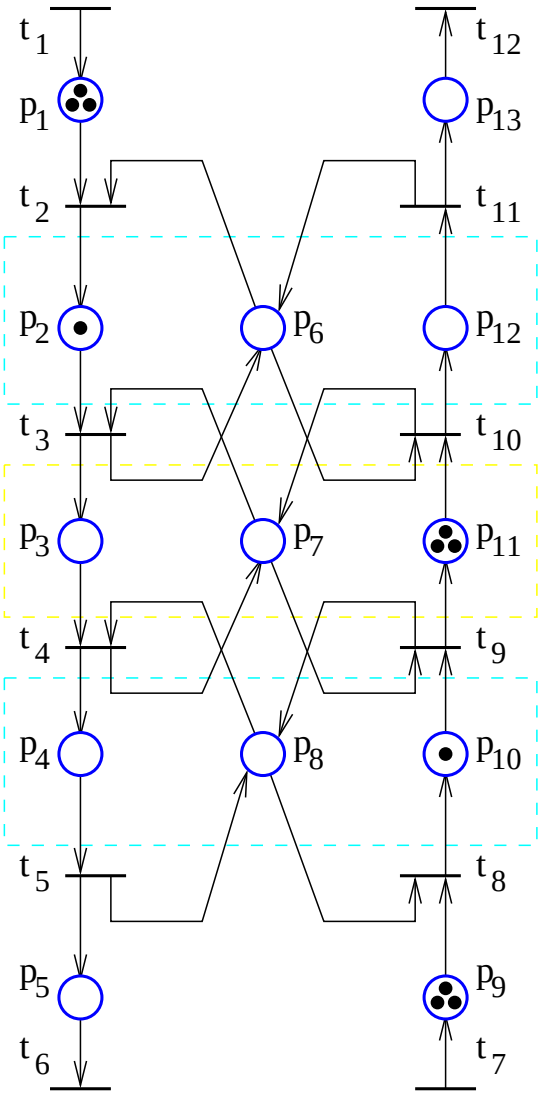




Deadlock, as no continuation is possible from the current state.

Problem: *How to constrain the markings to ensure deadlock cannot occur?*



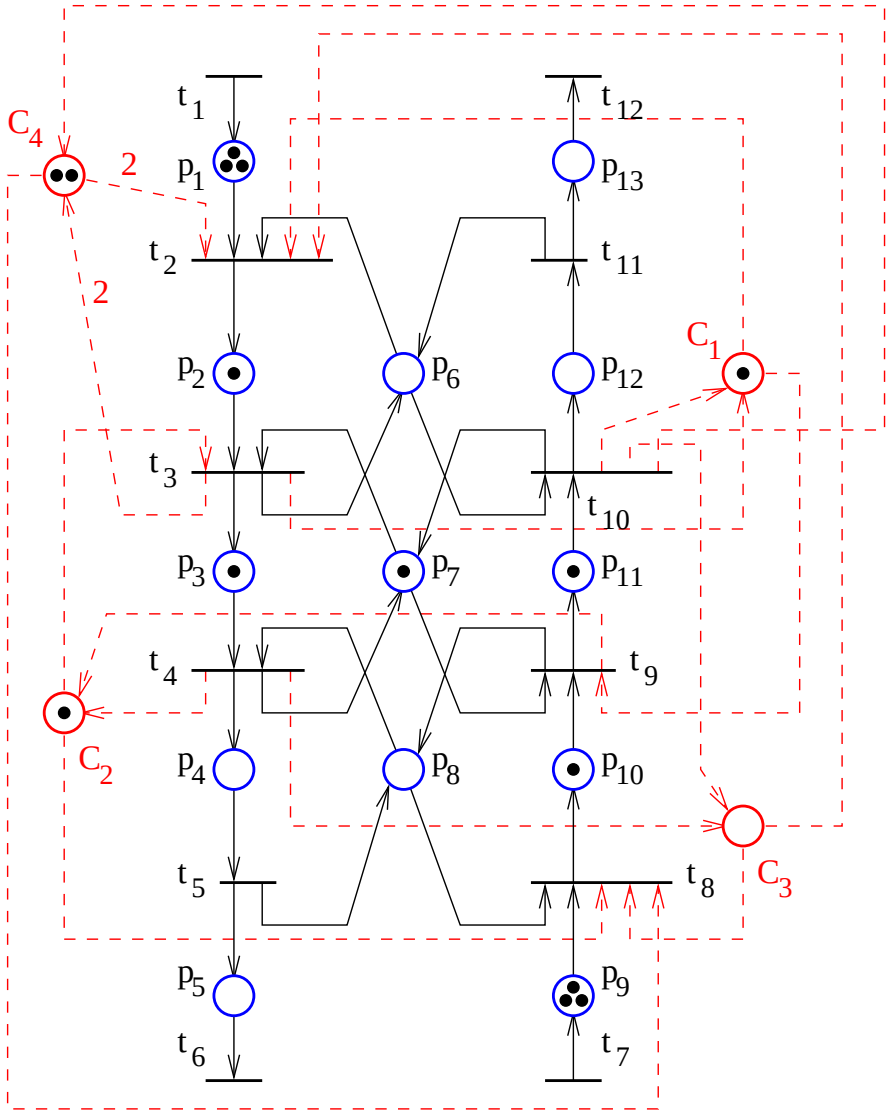


The inequalities to be enforced:

$$\begin{aligned}
\mu_3 + \mu_6 + \mu_7 + \mu_{12} &\geq 1 \\
\mu_4 + \mu_7 + \mu_8 + \mu_{11} &\geq 1 \\
\mu_4 + \mu_6 + \mu_7 + \mu_8 + \mu_{12} &\geq 1 \\
2\mu_3 + \mu_4 + 2\mu_6 + 2\mu_7 + & \\
\mu_8 + \mu_{11} + 2\mu_{12} &\geq 3
\end{aligned}$$

In addition to the inequalities above, the initial marking μ_0 of the plant must satisfy:

$$\begin{aligned}
\mu_{0,2} + \mu_{0,6} + \mu_{0,12} &\geq 1 \\
\mu_{0,3} + \mu_{0,7} + \mu_{0,11} &\geq 1 \\
\mu_{0,4} + \mu_{0,8} + \mu_{0,10} &\geq 1
\end{aligned}$$



For Future Work

- Specification language.
  - *Problem: For complex enough protocols specifications can be large.*
  - For future work: Add high level instructions.
- The deadlock prevention tool generates constraints that guarantee a deadlock-free operation of the Petri net representing the software.
  - *Problem: There may be deadlocks that are not visible in the Petri net model.*
  - For future work: Implement responsiveness algorithms [Iordache, 2010 CDC].
  - *Problem: The current deadlock prevention procedure is not guaranteed to converge and has high computational complexity. At this time it is not known whether these issues are likely to manifest themselves in practice.*
  - For future work: Implement other algorithms or develop new algorithms.
  - *Problem: Deadlock prevention assumes (partially) repetitive structures.*
  - For future work: Generalize the deadlock prevention approach.