A close-up of a logo

Description automatically generated

Modelling Of Software

Intensive Systems

Assignment 3: Petri-Net

1st Master computer science

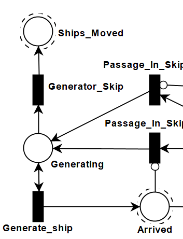
2024-2025

Liam Leirs

Robbe Teughels

Petri-Net Construction:

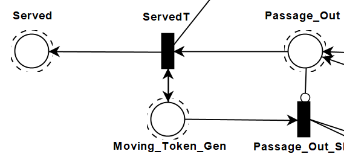
**Generator:**



**When The generator receives a token (generating), it can choose between Generator\_ ship/skip to generate as many/few as non-deterministically chosen.**

**Sink:**

**When moving out the passage, ships enter the sink, either a node to act as a counter or they can be deleted.**



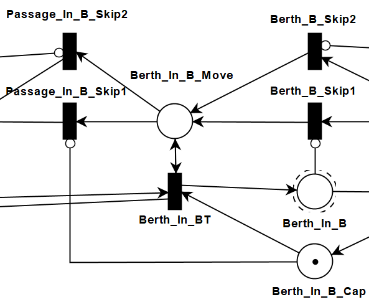
**Capacity constraints:**

**A diagram of a diagram of a circle and a circle with arrows

Description automatically generatedTo make sure that the capacity of any state is respected, we keep track of the remaining capacity of the state. Entering a place consumes a capacity token and leaving places it back.**

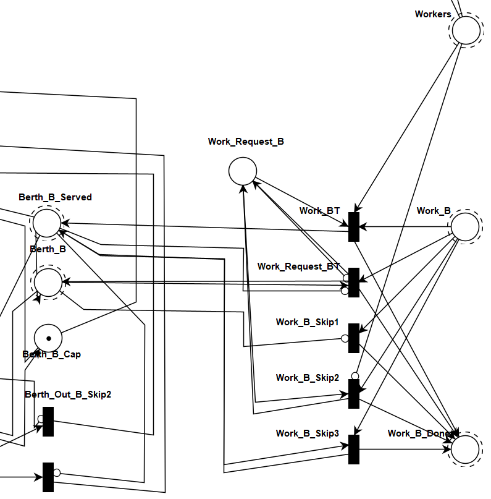
**(Ship\_Move.Berth\_In\_BT means moving to Inbound-passage-B, Ship\_Move.Berth\_BT means moving to Berth\_B so leaving Berth\_In\_B)**

**Deterministically moving with Token:**



**When token received (token in Berth\_In\_B\_Move), ships are allowed to move into Bert\_In\_B. When we are done moving, we must continue the token. To deterministically move (if possible), when moving is enabled, all skips must be disabled. When moving is disabled, at least one skip must be enabled. So skip 1 can fire when there is no remaining capacity left and skip 1 can fire when there are no ships able to move into the state. These combined give for all reachable combinations at least 1 possibility to continue the token and avoid deadlock.**

**Serving ships:**



**Ships can be served when a token is in Work\_B, When a ship enters, it must first request to be served. This request continues the token so the serving of the ship is delayed by 1 cycle. When there is an active request and a worker is available, a marking is applied in Berth\_B\_Served so the ship can continue. This consumes both the request and the worker, preparing for the next ship and making sure the workers aren’t used multiple times. To avoid deadlock, 3 skip conditions are present: No ship, no worker available, ship already served.**

**Clock:**

**A diagram of a network

Description automatically generated**

**The clock is a closed loop that controls what actions can be done. After some initialization (passage cap), the token starts in Next\_Tick. First we allow the movement of Ships. This is a token that continues in the opposite direction of the ship connections and allow them to fire. For multiple berths, the token is split up so berths can run individually and are required to both finish and synchronize the tokens back to 1 afterwards. After all connections had the chance to fire, the token reappears in Ship\_Moved. We reinitialize the workers by consuming all remaining and replacing them. We continue the token to either Work\_A or Work\_B to serve a ship. The priority is based on the value Priority\_Workers which prioritizes the one that was second, previous tick. When a ship is served (skip, …) the chance is given to the other one. With this, we introduces fairness for workers in the model. When both had the chance, the token appears back in Next\_Tick, ready for the next iteration.**

**Combining all the elements, we get created two parts: the clock and Ship\_Move for logic and Prot\_Overview for visual purposes. A diagram of a connection

Description automatically generated with medium confidence**

**A diagram of a network

Description automatically generated**

Scenarios:

m = 1, n = 1, 10 initial ships:

First iteration, one ship moves from Arrived into the passage. All following iterations, the ships move to the next place and a new ship enters the passage. (except for the berth because serving takes an extra tick). When we reach following state, the ships are block by each other because ships from Berth\_Out can’t move into the passage\_out, blocked by the incoming ship and the low capacity of the entire passage. This results in a live-lock where ships can’t move but the clock keeps running, allowing for (skip) moves to be made.

A diagram of a network

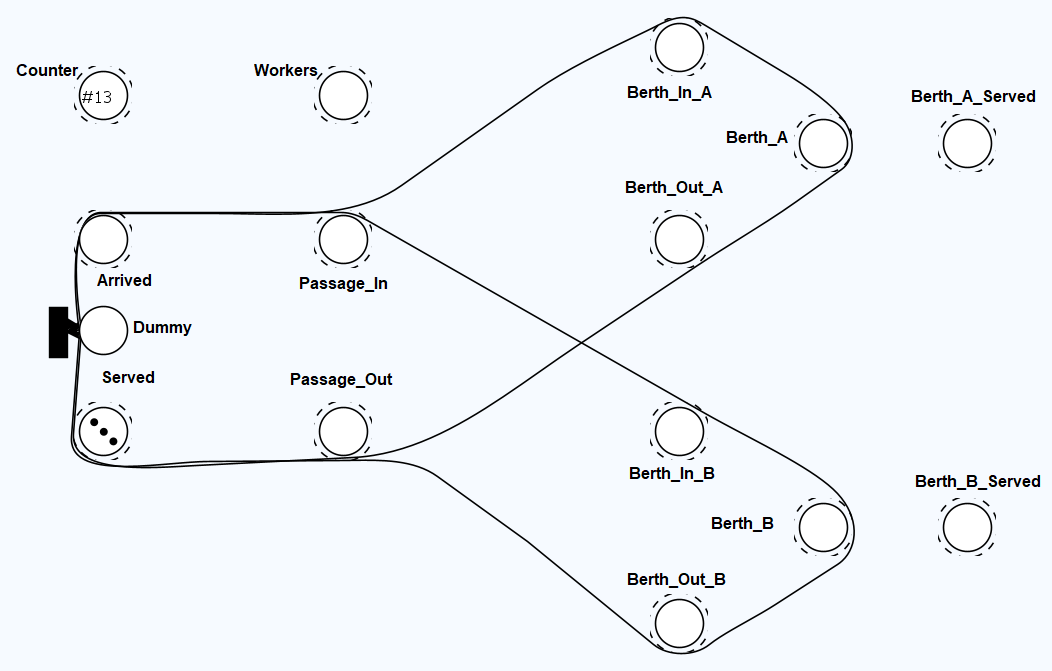
Description automatically generated

m = 3, n = 4, 3 ships over 3 ticks:

For the first 3 iterations, a ship arrives. Every iteration it continues to the next place, non-deterministically choosing the Berth. The first two ships enter berth A and the third enters berth B. As the first ship is served at the Berth, the second must wait a tick before being able to continue. As a result, the second ship is delayed by one tick and enters Served simultaneous with the third ship, “ending” the simulation in tick 10.

(Simulation trace continued for a couple ticks, but we assume it stopped)

As the simulation successfully terminated, there is no active looping going on so we are not in a live-lock. We successfully archived our goal as all ships are served. There are no request for moving a ship so no one is obstructed by terminating the program. (And no deadlock).



10

Reachability / Coverability analysis

For generating the reachability and coverability graphs we used the command ./RC.py [input] [output] [type] as described at the top of the file. For the P-invariants the same command was used but appended with the -p parameter.

Our solution clearly results in an infinite reachability graph, also visible in the coverability graph by the presence of ω in the states. The reason we get an infinite reachability graph is because an unbounded number of new ships can arrive at each tick alongside the fact that the waiting area is unbounded. This allows the system to produce infinitely many reachable states.

To make the reachability graph finite a limit could be placed on the number of allowed tokens in these states (waiting area / arrival). This will only allow a limited number of ships in the system at a time and will thus only generate a finite reachability graph.

**P-invariants analysis**

**M(T\_Generator) = 1:** generator token remains constant ensuring continuous generation of ships. This is to be expected because it keeps the system active.

**M(Berth\_B) + M(Berth\_In\_B) + M(Ship\_Move.Berth\_B\_Cap) + M(Ship\_Move.Berth\_In\_B\_Cap) = 2:** Tokens across the berth B, its input passage, and associated capacity places sum to 2. This models the bounded capacity of the berth and its input passage. This is again to be expected as each berth has a fixed capacity of 1 ship.

**M(Berth\_B) + M(Ship\_Move.Berth\_B\_Cap) = 1:** Ensures that at most 1 ship is either at berth B or its capacity placeholder.

**M(Berth\_A) + M(Ship\_Move.Berth\_A\_Cap) = 1**: Similar to previous one but for berth A.

**M(Passage\_In) + M(Passage\_Out) + 3 \* M(Ship\_Move.Init\_Model) + M(Ship\_Move.Passage\_Cap) = 3:** The passage's total capacity is distributed among incoming, outgoing, and in-transit ships. This models the bounded shared passage. To be expected since the common passage is bounded to 3.

**M(Workers\_Created) = 8**: The total number of workers in the system is fixed at 8.

**M(Berth\_In\_B) + M(Ship\_Move.Berth\_In\_B\_Cap) = 1:** Ensures only 1 ship can occupy berth B’s input passage or its capacity placeholder. This aligns with the uni-directional passage constraint.

**M(Pre\_Passage) = 12:** Represents the initial token count in the pre-passage place, modeling the initial setup.

**M(Berth\_Out\_A) + M(Ship\_Move.Berth\_Out\_A\_Cap) = 1:** At most one ship can occupy berth A’s exit passage or its placeholder. Consistent with uni-directional exit constraints.

**M(Berth\_Out\_B) + M(Ship\_Move.Berth\_Out\_B\_Cap) = 1:** Similar to previous one but for berth B.

**M(Berth\_A) + M(Berth\_B) + M(Berth\_In\_A) + M(Berth\_In\_B) + M(Berth\_Out\_A) + M(Berth\_Out\_B) + ... + 3 \* M(Ship\_Move.Init\_Model) + M(Ship\_Move.Passage\_Cap) = 9:**  Represents global conservation of tokens across all places in the system.

**M(Clock.Workers\_Active) + M(Moving\_Token\_Gen) + ... + M(Work\_B\_Done) = 1:** Represents single clock token that governs the system’s sequential evolution. It ensures proper clock-driven semantics.

**M(Berth\_A) + M(Berth\_In\_A) + M(Passage\_In) + M(Passage\_Out) + ... = 5:** Limits the total number of tokens within berth A’s subsystem and shared passage. Reflects bounded passage capacity.

**M(Post\_Passage) = 28:** Tokens in post-passage place

**M(Berth\_A) + M(Berth\_In\_A) + M(Ship\_Move.Berth\_A\_Cap) + ... = 2:** Total token conservation across berth A and its capacity. Similar to the one for berth B and to be expected.

**M(Berth\_In\_A) + M(Ship\_Move.Berth\_In\_A\_Cap) = 1:** Conservation of tokens in berth A’s input and capacity placeholder.