A close-up of a logo

Description automatically generated

Modelling Of Software

Intensive Systems

Assignment 3: Petri-Net

1st Master computer science

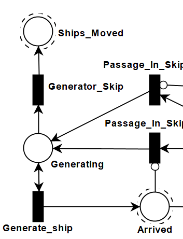
2024-2025

Liam Leirs

Robbe Teughels

Petri-Net Construction:

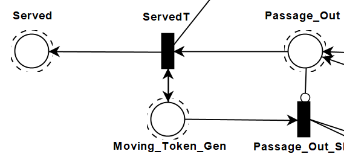
Generator:



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.



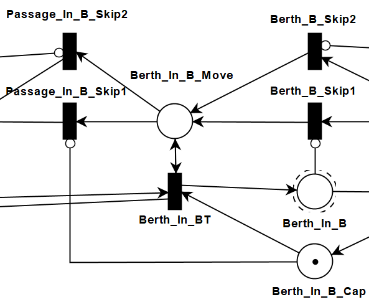
Capacity constraints:

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.

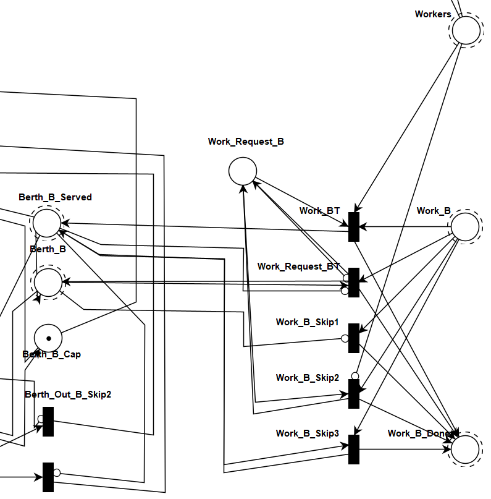
(Ship\_Move.Berth\_In\_BT means moving to Inbound-passage-B, Ship\_Move.Berth\_BT means moving to Berth\_B so leaving Berth\_In\_B)

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.

BONUS: difference between 1 and 4 workers:

Scenario: m=3, 2 ships in Arrived, 1 or 4 worker

Both ships move simultaneously (two sequential fires to move) into the passage. Both enter simultaneously in a (in\_) berth.

1 Worker:

When assigning workers, berthB gets priority. B gets served while A must wait a tick.

The ship moving through birthA arrives at Served a tick later than the other ship.

4 Workers:

Both ships gets served. Both ships move simultaneously until arrived in Served in the same tick.

**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

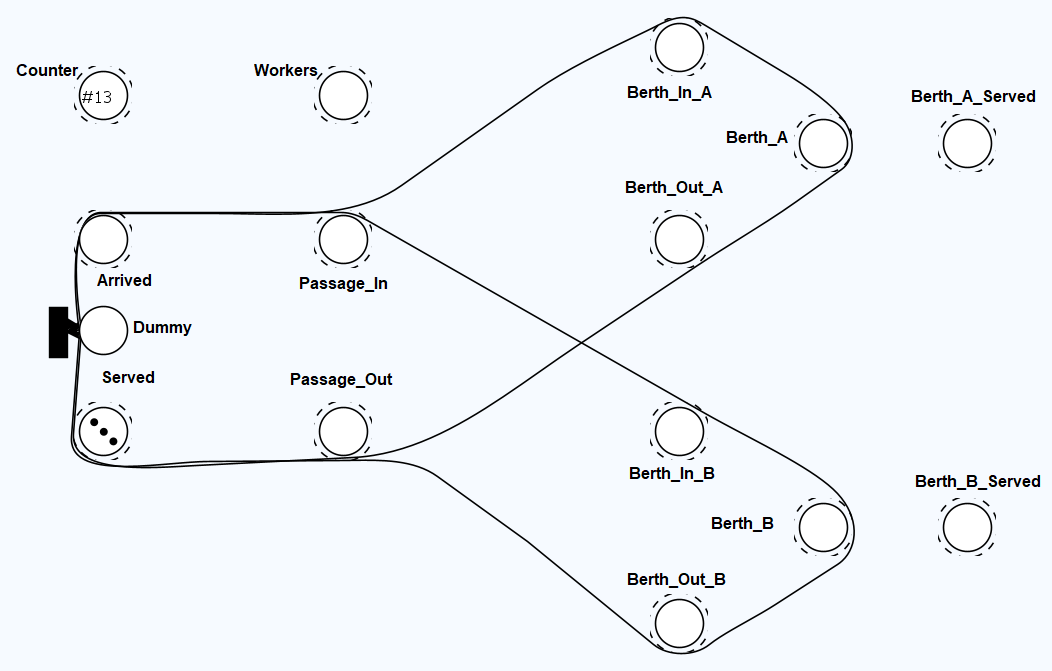
Description automatically generated

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.

aspects of the port system:

Boundedness:

To describe boundedness of the system, we can split it into multiple cases. The counter increases every tick by one, leading to infinity. Therefore we assume that this will not be part of the system we are evaluating. Our system has an invariant that assumes that the sum of all ships and remaining capacity of the states (excluding Arrived and Served) is 9. Therefore these are bounded. The clock logic running through Bert\_A also has an invariant that sums up to 1 (same for berth B, this includes worker logic).

Our system is bounded as what parts are described as the system. The Arrived state and Served state are parts that can be excluded and given to the environment to manage. Then the system becomes bounded. Including one or both leads to an unbounded system.

Deadlock:

Deadlocks can occur in the model. This happens when the generator generates to much ship that enter the passage\_in. When the entire passage capacity is used for inbound traffic, ships will not be able to leave the port. The entire berth will fill with ships until no one can move any more. This is a live-lock for the system and a deadlock for the ships.

When excluding the Arrived state, the deadlock is a cycle where all transitions are skips. A token is moved around in a circle while the values of ships is consistent for all states in the cycle (no one moves).

There are two options for preventing a deadlock. Reserving capacity for outbound ships makes sure ships can always leave the system, preventing the lock. But for m = 1, it creates a new deadlock that new ships can not enter resulting in another deadlock. Limiting the ships in the system. When analyzing the system, we can find a minimum number of ships in the system that results in a deadlock. By limiting the input ships to a value lower than this, the deadlock can be prevented.

Liveness:

Picking any arbitrary transition in the model and connecting it to the init\_model ensures that when it doesn’t fire as firs, it will never fire. Connecting For most transitions, this will result in deadlock of the entire system. There are a few transitions (Generator\_Skip, Init\_Passage, …) Connecting them backward ensures that once they fire, they can always fire.

Fairness:

We implemented fairness for all events by design of the clock. This holds only when some assumptions are hold: there is exactly one token circulating for serial event. Tokens can split up for multiple parallel tasks (berth\_A / berth\_B movement) but must synchronize into a single token (always eventually 1 token). There clock must be a cycle with: for every possible path, every event must had access to a token.

On the level of the clock, we allow every events to happen exactly once before starting the next iteration. (Events in order: Move\_Ships, Reinit\_Workers, Work A/B, or B/A)

When each of these events is finite, fairness is guaranteed. Reinit\_Workers is trivial and Work should be clear. Movement takes the token and moves it backwards through the move transitions and allows for movement of ships. By doing it backwards, a moved ship is in a state where the transitions to move forwards has been passed already and must wait for the next cycle. Berts are parallel executed and synchronized afterwards. The movement ends in generating new ships. When we assuming Fairness, we assume that the choice between generating ships and skipping eventually skips. Without this assumption we can get into an infinite cycle of generating ships and Fairness wouldn’t hold.

Safety:

It is impossible to crash ships in the current model. By construction of the capacity constraint (as long as no other transitions alter the content of the state/cap), there is an invariant for every state with a capacity, the sum of the state and its remaining cap is equal to the capacity itself.

To make a trace where ships crash, we add a arc (the transition must fire at least once) to the remaining cap. This allows for more ships than allowed in a state. (Crash happened in Berth\_A in the trace)