Universiteit Antwerpen
Faculteit Wetenschappen

Modelling Of Software Intensive Systems

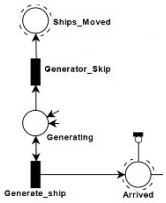
Assignment 3: Petri-Net

1st Master computer science
2024-2025

Liam Leirs
Robbe Teughels

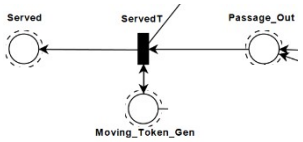
Petri-Net Construction:

Generator:



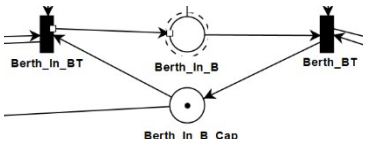
When The generator receives a token (generating), it can choose between Generator_ ship/skip to generate as many/few as non-deterministically chosen.

Sink:



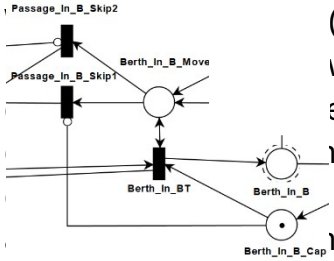
When moving out the passage, ships enter the sink, either a node to act as a counter or they can be deleted.

Capacity constraints:



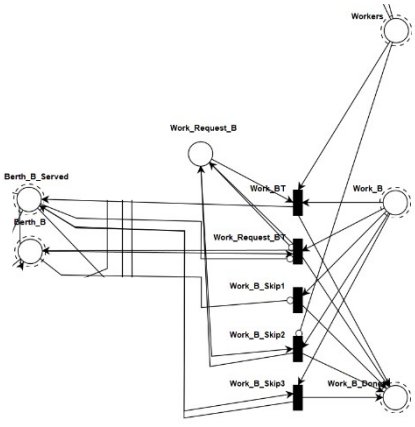
To make sure that the capacity of any state is respected, we keep track of the remaining capacity of the state. Entering a place consumes a capacity token and leaving places it back.
(Ship_Move.Berth_In_BT means moving to Inbound-passage-B, Ship_Move.Berth_BT means moving to Berth_B so leaving Berth_In_B)

Deterministically moving with Token:



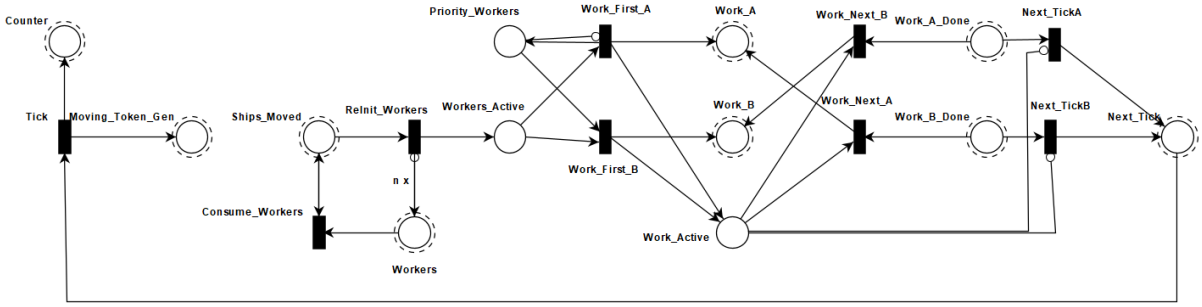
(token in Berth_In_B_Move), ships are allowed to move we are done moving, we must continue the token. To e (if possible), when moving is enabled, all skips must be ıg is disabled, at least one skip must be enabled. So skip 1 no remaining capacity left and skip 1 can fire when there ıove into the state. 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:



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:

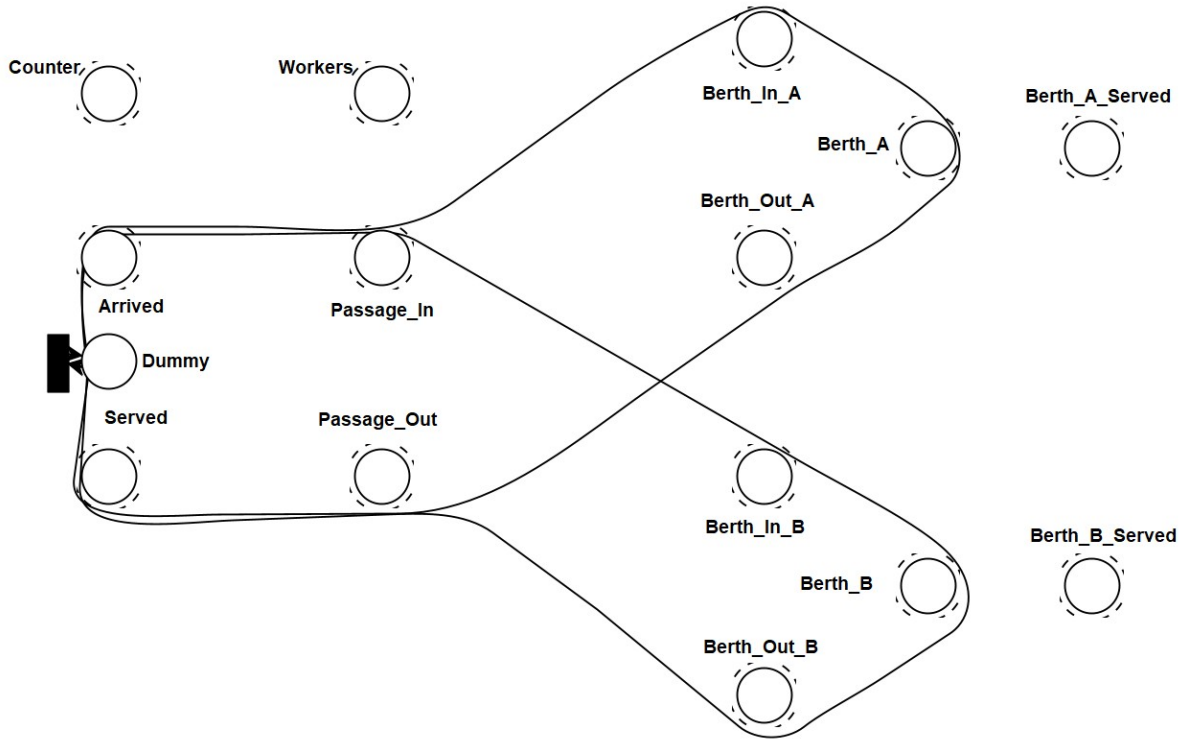
Scenario: m=3, 2 ships in Arrived, 1 or 4 worker

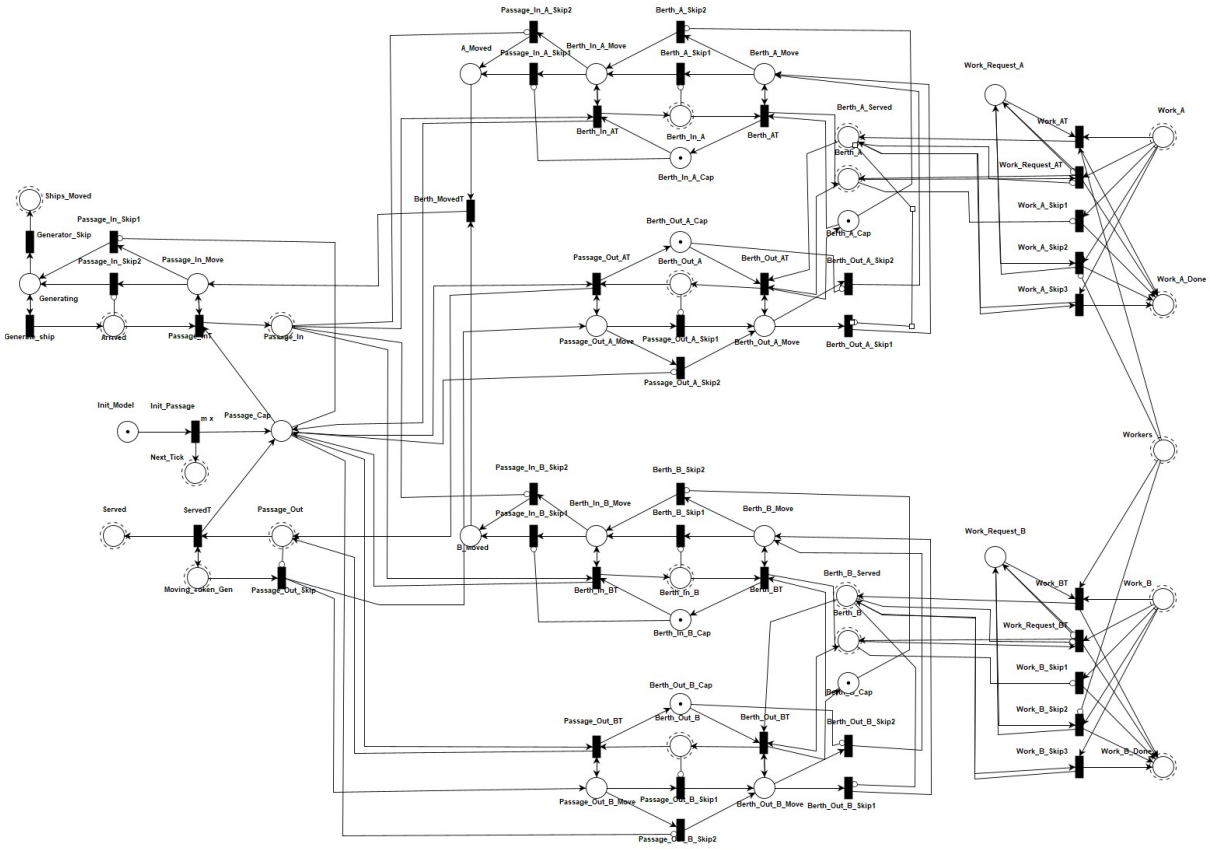
Both ships move simultaneously (two sequential fires to move) into the passage. Both enter simultaneously in a (in_) berth.

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.

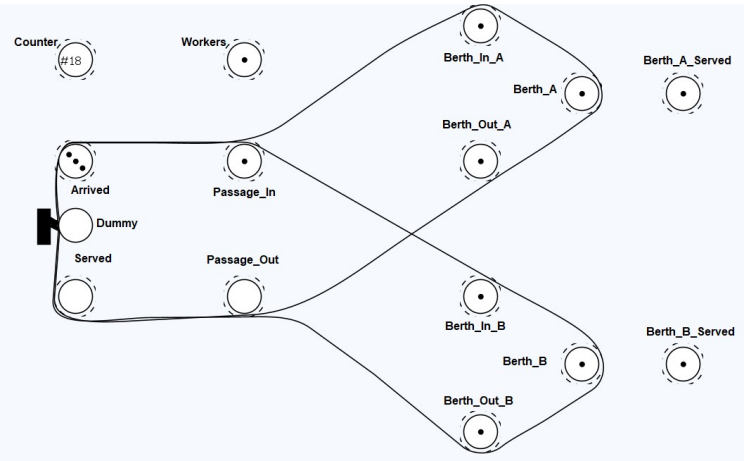

Counter
Workers
Berth_In_A
Berth_A_Served
Berth_A
Berth_Out_A
Arrived
Passage_In
Dummy
Served
Passage_Out
Berth_In_B
Berth_B_Served
Berth_B
Berth_Out_B


Passage_In_A_Skip2
Berth_A_Skip2
A_Moved
Passage_In_A_Skip1
Berth_In_A_Move
Berth_A_Skip1
Berth_A_Move
Work_Request_A
Berth_A_Served
Work_AT
Work_A
Berth_In_AT
Berth_In_A
Berth_AT
Berth_A
Work_Request_AT
Berth_In_A_Cap
Ships_Moved
Berth_MovedT
Passage_In_Skip1
Berth_Out_A_Cap
Work_A_Skip1
Generator_Skip
Berth_A_Cap
Work_A_Skip2
Passage_In_Skip2
Passage_In_Move
Passage_Out_AT
Berth_Out_A
Berth_Out_AT
Berth_Out_A_Skip2
Work_A_Skip3
Work_A_Done
Generating
Generate_ship
Arrived
Passage_In
Passage_Out_A_Move
Passage_Out_A_Skip1
Berth_Out_A_Move
Berth_Out_A_Skip1
Passage_Out_A_Skip2
Init_Model
Init_Passage
m x
Passage_Cap
Workers
Next_Tick
Passage_In_B_Skip2
Berth_B_Skip2
Served
ServedT
Passage_Out
Passage_In_B_Move
Berth_B_Skip1
Berth_B_Move
Work_Request_B
B_Moved
Berth_B_Served
Work_BT
Work_B
Berth_In_BT
Berth_In_B
Berth_BT
Berth_B
Moving_token_Gen
Passage_Out_Skip
Berth_In_B_Cap
Work_Request_BT
Berth_Out_B_Cap
Work_B_Skip1
Berth_Out_B_Cap
Work_B_Skip2
Passage_Out_BT
Berth_Out_B
Berth_Out_BT
Berth_Out_B_Skip2
Work_B_Skip3
Work_B_Done
Passage_Out_B_Move
Passage_Out_B_Skip1
Berth_Out_B_Move
Berth_Out_B_Skip1
Passage_Out_B_Skip2

Scenarios:

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

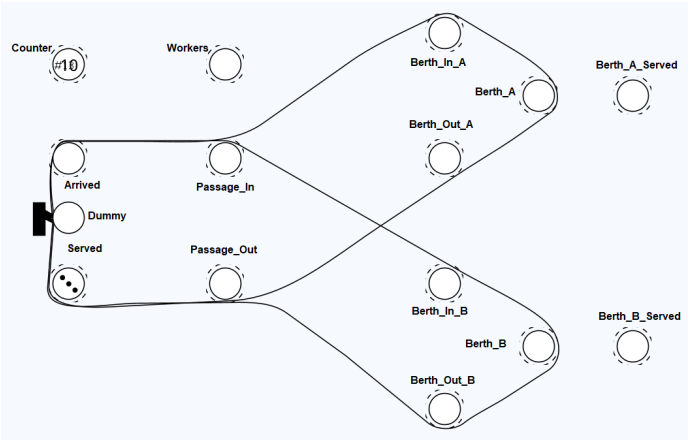
First iteration, one ship moves from Arrived into the passage. All following iterations, the ships move to the next place and a new ship enters the passage. (except for the berth because serving takes an extra tick). When we reach following state, the ships are block by each other because ships from 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.



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).



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.

Another way of limiting the size of the reachability graph is by removing the counter, served and arrived places from the petri net as these are the only unbounded states present in the net.

The coverability graph starts of with a clear linear progression, this is for the intitialization of the net (for example capacity placeholders). Alongside this ships move from the waiting area into the port. After this, branching occurs in the coverability graph indicating nondeterministic choices, this is because ships in the common passage may choose which berth to enter and workers may service ships in either berth. All together the branches also align with the fairness constraints.

P-invariants analysis

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(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(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(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.

We can change the boundedness of the system by adding a new place that acts as an overflow buffer, holding excess ships when the waiting area / common passage becomes full. If this place is unbounded, the system will in turn also become unbounded. This helps to ensure that the waiting area and common passage remain operational even under heavy loads.

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)