Petri nets - Modeling, simulation and analysis

Ken Bauwens 20143225
and Baturay Ofluoglu 20174797

November 26, 2017

1 Modeling

1.1 1st Requirement

Requirement: "A junction has two entrances and one exit. For each entrance, there is exactly one train."

Solution:
In the previous exercises, we did not explicitly model the junction. This was because a junction is the connection between two tracks, and so it does not really feel like an entity. This reasoning is however not correct, because when a train is on the junction, he is essentially part of its inputtrack and the exittrack. Making this distinction can be important to prevent collisions. If a train is on the junction, it occupies both it's input track and the exit track, so a new train cannot arrive and there should be no train on the exittrack.
The distinction is also important to allow the modeling of e.g. a train breaking down while it is on both the tracks.
Our design for requirement 1 can be seen in figure 1.
A place indicates a track, or a junction. When a place has mark 1, there is a train on that track. This also means that there can never be more than one mark in a place, as this would indicate a collision.
It is obvious from figure 1 that the junction has two entrances and one exit. The initial state will have a single train on each of the input tracks.
To allow for new trains to arrive, we extended our junction to figure 2. When a train leaves the junction, a new train can arrive on one of the input tracks. To make sure that the new train does not arrive on an inputtrack that already has a train, we use inhibitor arcs from the input tracks to the trainXenters transitions.

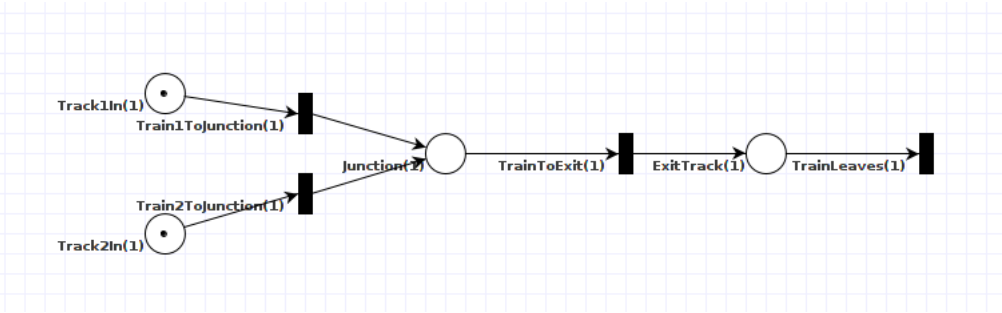


Figure 1: Junction

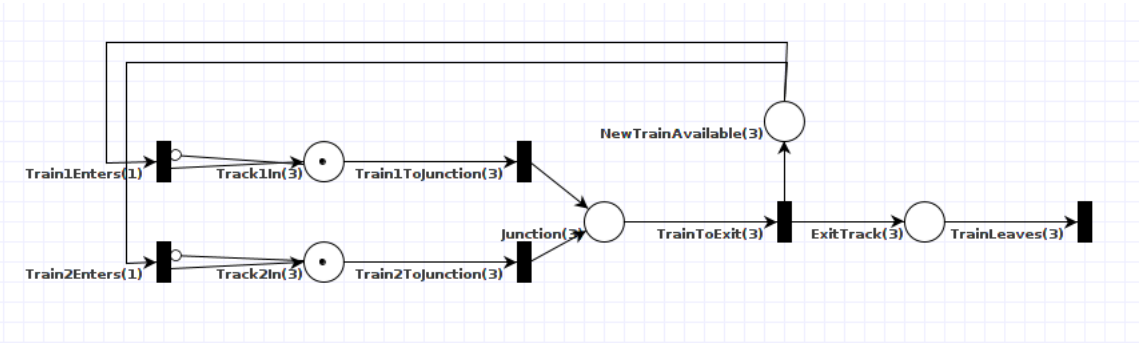


Figure 2: Junction with new train generator

1.2 2nd Requirement

Requirement: "Each entrance has a traffic light that is either red or green. Green means that a train on the associated entrance can go on the junction exit. Red means that a train on the associated entrance has to wait."

Solution:
For each entrance, we added a new place modeling a traffic light. A traffic light is green when it contains a mark, and it is red otherwise. Both traffic lights will be connected to the correct TrainXToJunction transition. This means that a train can only continue on the junction if it has a green light. This construction can be seen in figure 3.

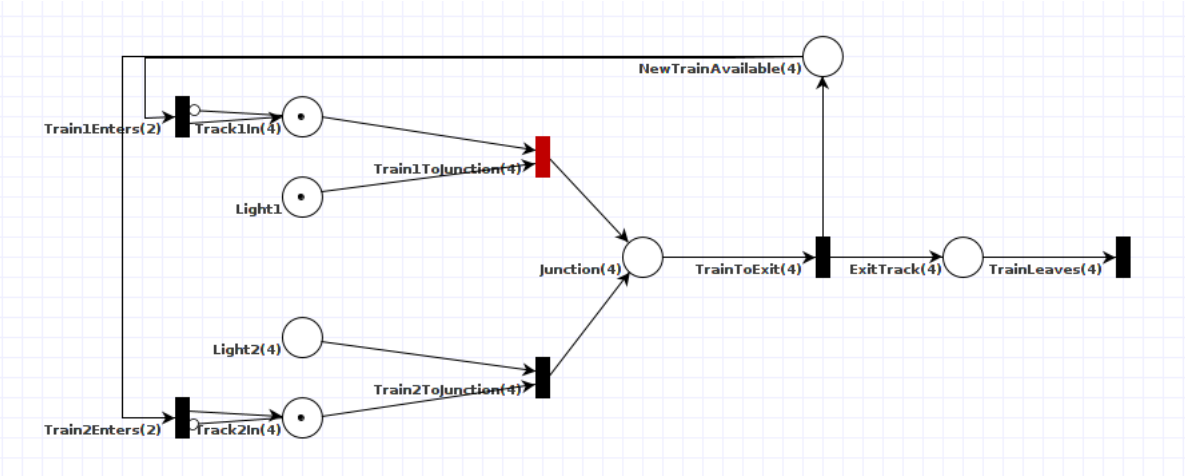


Figure 3: Lights added to construction

1.3 3rd Requirement

Requirement: "Whether a train can continue to the exit is only determined by the traffic lights, i.e., the train/junction is agnostic of whether there is a train on the exit. A traffic light controller however can change lights based on train location and should make sure that trains can pass but can never collide on the junction exit."

Solution: As can be seen from figure 3, a train can only continue if it has a green light, and it will also only care about the light to make its decision.
To control the traffic lights, we added a traffic light controller. This controller can be seen in figure 4. It consists of a place containing a single mark, the *lightgenerator*. The transitions *LightXActivate* will then allow the controller to turn a single light green. Notice that these transitions are only active if there is no train on the junction or exit track to prevent collisions. They are also only active when there is a mark in the *lightgenerator*, to ensure that only one light can be green at any time. A light can also only turn green when there is a train on the track. This is not mandatory, but we added it for completeness and because it does not make sense to turn on a green light when there is no train on the track.

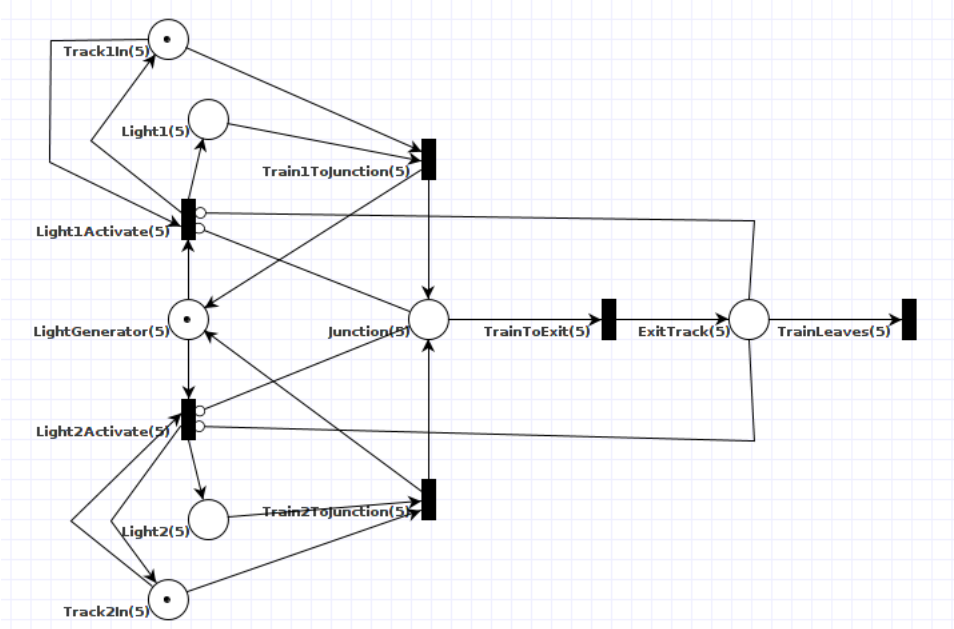


Figure 4: Light Controller

1.4 4th Requirement

Requirement: "Execution is done in a fair way using time cycles: the traffic light controller can change the traffic lights once per time cycle, and then every train has exactly one chance to move per time cycle (but it does not have to). So one time cycle consists of: the traffic light controller may change lights once, each and every train may move once."

Solution: To implement time cycles, we added a *Timecycle controller* which can be seen in figure 5. The controller will iterate between a *Lightmove* and *TrainMoves*. When a place contains a mark, the corresponding transition is allowed to fire. The *LightAllowed* place indicates that the lightcontroller can make a decision. If the lightcontroller finishes his move the *TrainsTurn* transition will fire. This transition gives each of the train transitions an opportunity to move. If each transition has had a chance to fire, the inhibitor arcs from the places to the *LightTurn* transition will allow the lightcontroller to make a new move, finishing the time cycle.
To prevent deadlocks, the transitions *DoesNotWantToX* are added. If for example there is no train on input track 1, but the track is allowed to move, the system can deadlock since the mark in the *Junction1Allowed* place won't be removed. In this case the *DoesNotWantToContinue1* transition will consume the mark, solving the deadlock.
These transitions also allow the modeling of choice. If a train does not want to continue even though it is allowed to, these transitions can facilitate that. This completes the *every train MAY move once* part of the assignment.
Similar checks are in place for the lightcontroller. One such check is to make sure that the controller won't assign a green light when there is already a green light active. All the checks can be seen in the complete petri net diagram in appendix C.
We define a single train move as a movement from any "legal" position to any "legal" position. This means that a train could start on the input track and end up leaving the output track in a single time slice. Adding extra logic to make sure a train only advances a single portion of the track would add unnecessary complexity. The movements of the trains are random and depend on the interleaving of the train movements (see requirement 5). The movements will however always be legal (No backwards movements for example).

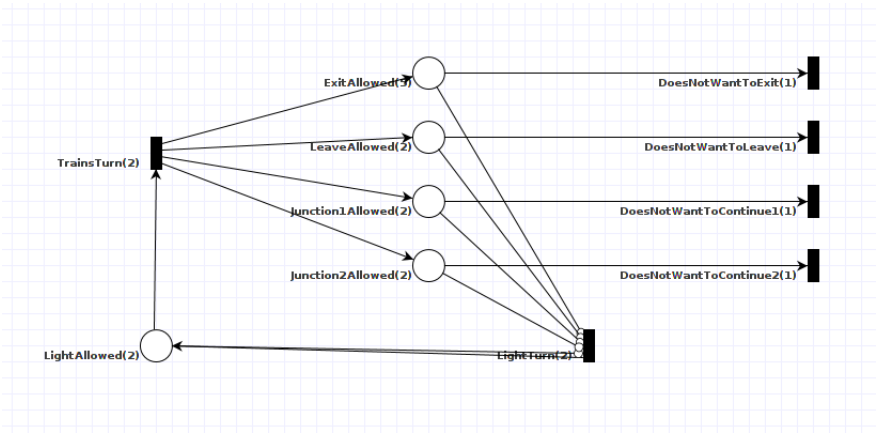


Figure 5: Light Controller

1.5 5th Requirement

Requirement: "Implement true fairness: the order of interleaving of events (i.e. of the train moves) within one time slice is nondeterministic."

Solution: Our construction of the timecycle controller as seen in figure 5 automatically facilitates nondeterministic interleaving of the trainmoves, because of the inherent nondeterministic nature of transition selection in petrinets. All trains will get a token allowing them to move. Which train consumes it's token first is then randomly chosen.

2 Simulation

In this section, we simulate our model and illustrate all of the possibilities starting from initial marking state until train exit state.

At the initial marking state, by default there is a train at both input track1 and track2. Therefore, the *Track1In* and *Track2In* have one token. To enable the light, by default *LightGenerator* place has one token as well.

2.1 Steps of Petri Net for 1 train

2.1.1 Input Tracks

After setting initial marking state, the first fired transition is *LightTurn*. This transition enables *Light1Active* and *Light2Active* transitions as in A. Thus, if *Light1Activate* fires randomly first then only first train will have a chance to go to junction and *Light2Active* will be disabled. Suppose *Light1Activate* is fired first. Then, *TrainsTurn* transition is enabled. Firing of *TrainsTurn* enables 5

transitions which are *Train1ToJuncktion*, *DoesNotWantToExit*, *DoesNotWantToLeave*, *DoesNotWantToContinue1*, *DoesNotWantToContinue2* as shown at B. If *Train1ToJunction* transition is fired first then *TrainToExit* transition is enabled. However, by chance if *DoesNotWantToContinue1* is fired first then *Train1ToJunction* is disabled. In this case, we need to wait *DoesNotWantToExit*, *DoesNotWantToLeave*, *DoesNotWantToContinue1*, *DoesNotWantToContinue2* transitions to fired. After all that transitions are fired then *LightTurn* transition enables again. Firing *LightTurn* enables *LightStillGreen* transition because train1 have already got green light. When *LightStillGreen* is fired then light turn ends and train turn begins again. Therefore, *TrainsTurn* transition is enabled and fired. Firing *TrainsTurn* enables *TrainToJunction*, *DoesNotWantToExit*, *DoesNotWantToLeave*, *DoesNotWantToContinue1*, *DoesNotWantToContinue2* again. This cycle continues until *TrainToJuction* is fired before *DoesNotWantToContinue1* transition. Suppose *TrainToJunction* is fired before than *DoesNotWantToContinue1*. In this case, *DoesNotWantToContinue1* is disabled and *TrainToExit* transition is enabled. It means that train come to the junction.

2.1.2 Junction

In this state, if *DoesNotWantToLeave* is fired before *TrainToExit* then *TrainToExit* is disabled. However, to exit from out track, *TrainToExit* transition should be fired. To enable *TrainToExit* again, same steps need to be applied as in the example of *DoesNotWantToContinue1* is fired before *Train1toJuntion* transition. If *TrainToExit* transition fired first then *TrainLeaves* and *Train1Enters* is enabled. It means that train can proceed to out track and because junction is empty, a new train can come to input track 1 to continue the cycle.

2.1.3 Exit from Out Track

In this case, if *TrainLeaves* is fired before *DoesNotWantToLeave* then train leaves the out track safely. In case of firing *DoesNotWantToLeave* before *TrainLeaves*, then same cycle will be repeated as in the example of *DoesNotWantToContinue1* is fired before *Train1toJuntion* transition.

2.1.4 Going Back To Initial State

After *DoesNotWantToExit*, *DoesNotWantToLeave*, *DoesNotWantToContinue1*, *DoesNotWantToContinue2* transitions are fired, then the system go back to the initial state.

3 Analysis

3.1 Boundedness

This model is bounded. Because, all the places can have either 0 or 1 marks. It means that none of the place can have infinite marks. Since none of the place can have infinite mark, the model is considered to be bounded.

3.2 Deadlock

The Petri Net is deadlock free. Deadlock occurs when two items wants to share same resources and each item wait the others. Thus, both cannot seize that resource and they wait forever. It is called deadlock.
In our case, we handle all possible deadlocks. It means none of the tokens had to wait for infinity. For instance, if we would not handle deadlock problem, then the trains at Track1In and Track2In would wait each other forever to go to the Junction and none of them would pass to the Junction. Therefore, it would lead to deadlock.

3.3 Reachability

In fully complete Petri Net, there is 13 places and 16 transactions. Therefore, reachability graph cannot be plotted by PIPE2 Platform Independent Petri Net Editor 2.5.

To generate reachability graph, we eliminate time cycle from the petri net aswell as the generation of new trains. This means that in this modified petri net only the two first trains will be processed. This is however enough to show that it is impossible to have a collision. We used the reachability module in PIPE2 to generate the reachability graph for the petri net showed in figure 4. Refer to figure 6 for reachability graph.

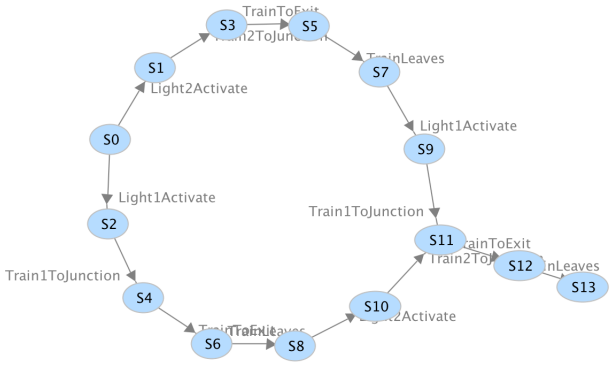


Figure 6: Light Controller

In this graph, the reachable states are:
S0: 1,0,0,1,1,0,0
S1: 1,0,1,1,0,0,0
S2: 1,1,0,1,0,0,0
S3: 1,0,0,0,1,1,0
S4: 0,0,0,1,1,1,0
S5: 1,0,0,0,1,0,1
S6: 0,0,0,1,1,0,1
S7: 1,0,0,0,1,0,0
S8: 0,0,0,1,1,0,0
S9: 1,1,0,0,0,0,0
S10: 0,0,1,1,0,0,0
S11: 0,0,0,0,1,1,0
S12: 0,0,0,0,1,0,1
S13: 0,0,0,0,1,0,0

After analyzing the states, we can interpret that all the places can have at most 1 mark and for example for the S4 state which checks the junction, this state 0,0,0,2,1,1,0 is not reachable. It is desirable because it means that there are two trains at the junction.

3.4 Liveness

All transitions can fire during an execution. If this was not the case, we would have dead transitions and we could remove these. In our petri net however each transition serves a fairly specific purpose, so removing one would completely destroy the net. A short simulation can already prove that each state is at least fired once.
To figure out what level of liveness the transitions have, we can look at the invariants. By definition transition invariants are sequences of transitions which will start from a state and end in the exact same state. Obviously these cycles can be ran an infinite amount of time without ever stopping. From this we can conclude that all transitions in these transition invariant cycles are l3-live. In subsection 3.5.1 we can see that all transitions are part of some invariant cycle, so all our transitions are at least l3-live.
We can also check if any transitions are l4-live. Our petri net is "continuous", because for example a new train will arrive when a train has left the junction. This means that it is always possible to return to a previous state. We also know that all transitions are l3-live, meaning that they will always be part of a set of transitions starting from a specific state. This means that all transitions can become active from whatever state reachable from the initial marking, making all the transitions l4-live.

3.5 Invariants

3.5.1 Transition Invariant

Transition invariants lead to a cycle at the reachability graph. Firing the transitions occurred in transition invariants in any order leads to a cycle and it returns back to initial marking again.

In our model we have 5 different transition invariants:

1. DoesNotWantToExit + DoesNotWantToLeave + DoesNotWantToContinue1 + DoesNotWantToContinue2 + JunctionOccupiedWhileLightMove + LightTurn + TrainsTurn
2. DoesNotWantToExit + DoesNotWantToLeave + DoesNotWantToContinue1 + DoesNotWantToContinue2 + ExitOccupiedWhileLightMove + LightTurn + TrainsTurn
3. DoesNotWantToExit + DoesNotWantToLeave + DoesNotWantToContinue1 + DoesNotWantToContinue2 + LightStillGreen + LightTurn + TrainsTurn
4. Light1Activate + DoesNotWantToContinue2 + Train1ToJunction + TrainToExit + TrainLeaves + Train1Enters + LightTurn + TrainsTurn
5. Light2Activate + DoesNotWantToContinue1 + Train2ToJunction + TrainToExit + TrainLeaves + Train2Enters + LightTurn + TrainsTurn

All the transition invariants will end in LightTurn + TrainsTurn. This indicates a single timeslice.
The first 3 equations symbolize a lightmove that does nothing. This can happen when either a train is on the junction (1), a train is on the exit (2), or a train on an input track already has a green light (3). Another requirement for these cycles is that all trains decide not to move, either because there is no train on a track or because they nondeterministically decided to not move (see question 1 requirement 4).
The 4th and 5th equations symbolize the process of train arrivals to input track until it exits from the junction. These processes are same for both 1st and 2nd incoming trains. For example, for the 4th case Train1Enters transition is fired and train come to input track. Then, LightTurn and Light1Activate transition will let train1 to move. Following to LightTurn, TrainsTurn is fired to move the trains. However, to avoid train crashes DoesNotWantToContinue2 is fired to stop train 2 because train 1 gets green light. After this transition, train 1 come to junction by using Train1ToJunction transition. At the end, the train will follow TrainToExit and TrainLeaves transition to complete the cycle.

3.5.2 Place Invariant

Place invariant means that weighted token sum is always constant. Suppose weight is z, p is place and m is mark. Then $z1.m(p1) + z2.m(p2) + z3.m(p3) +$

$zkm(pk) = z0$.

In our model we have two place invariants which are:

1. $M(Track1In)+M(Track2In)+M(Junction)+M(NewTrainAvailable) = 2$
2. $M(Light1) + M(Light2) + M(LightGenerator) = 1$

The first invariant means that if there is a train at input track 1 and 2, then Junction and NewTrainAvailable should be 0 to avoid train crashes or if there is a train on input track 1 and and on junction then M(Track2In) and M(NewTrainAvailable) should be zero as well for the same security reason.

Second invariant controls that two lights can not get green light at the same time. Either light1 or light2 may be green.

3.6 Can two trains crash?

To check this requirement, we need to analyze 'Track1In', 'Track2In', 'junction' and 'TrackExit' places. If these places have more than one mark, then it means that there are two trains at that place which symbolize a train crash. Thus, we check these places from the reachability graph and the state values at these places are not more than 1. Therefore, we can conclude that the trains will not crash.

4 The Advantages and Comparison of Petri Nets

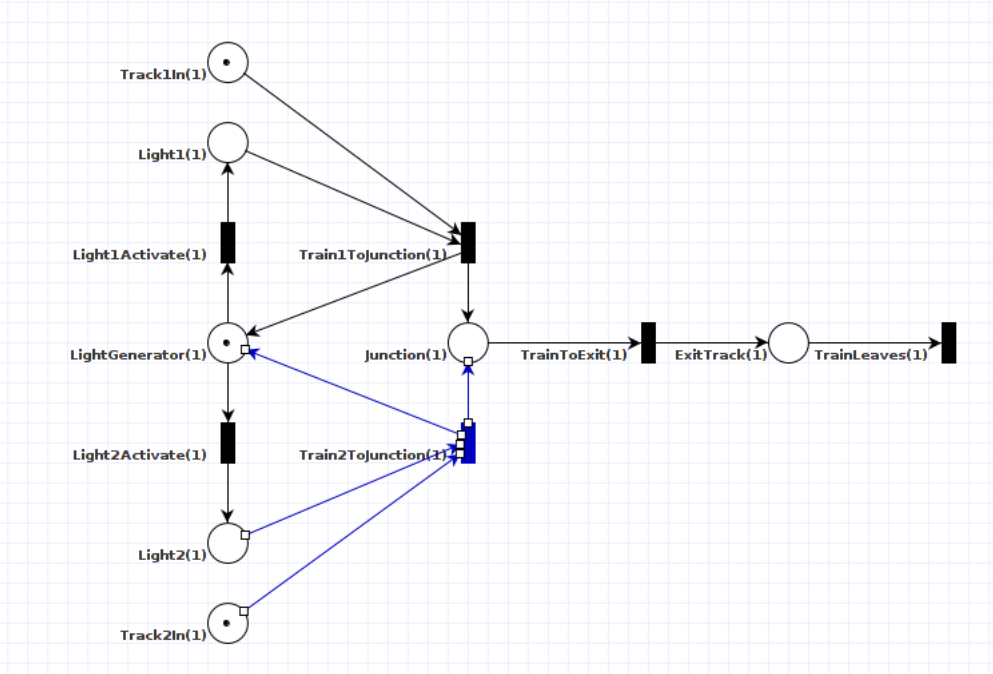
Petri nets can be very useful for modeling systems that require parallelism and nondeterminism. They can be useful to analyse desired properties of a system before actually building that system. In this assignment for example, we were able to model a system which prevents collisions between trains.
One disadvantage of petri nets is that large systems can often become unclear because of the high amount of transitions and arcs. In such cases it might be more advantageous to model using state automata. Note that state automata are not always more concise. This is because petri nets can model multiple states with a single net. A queue can for example be modeled way more concise with a petri net compared to a state machine.

5 Optional Questions

5.1 Classification

We can use the PIPE2 tool with the *classification module* to check if our petri net belongs to a subclass. The module returns false for all classes, so we know that our petri net is part of the broadest class.
We can reduce our petri net to the net shown in 5.1. When we use the *classification module* on this net, we see that the narrowest class this petri net belongs to is a *Free Choice Net*.

"In a free choice net (FC), every arc from a place to a transition is either the only arc from that place or the only arc to that transition. I.e. there can be both concurrency and conflict, but not at the same time." [1]



In terms of the functional properties, we notice that the collision prevention is gone. There is no lightcontroller which checks if the junction or exittrack are empty, so two trains can be on the juction at any time.
We can also see that it is possible that a train will never get a green light. Assume that train 1 gets a green light and completely exits the system. LightGenerator will have a mark, but it might "give" this mark to Light1. Because there won't be a train on Track1In anymore, Light1 will stay green forever and Light2 will never change.

References

[1] *Petri Nets*. Wikipedia, `https://en.wikipedia.org/wiki/Petri_net#Restrictions`

Appendices

A LightTurn Fired

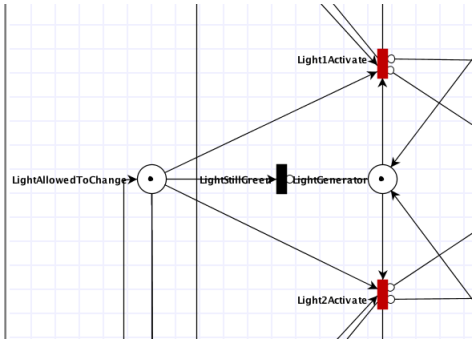
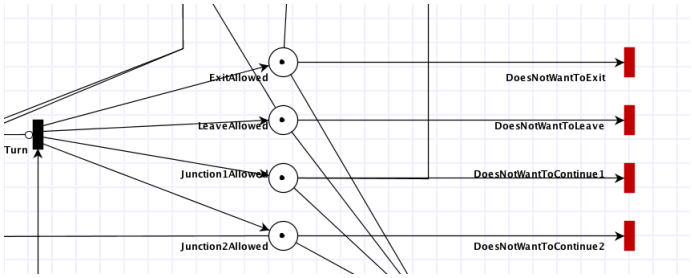
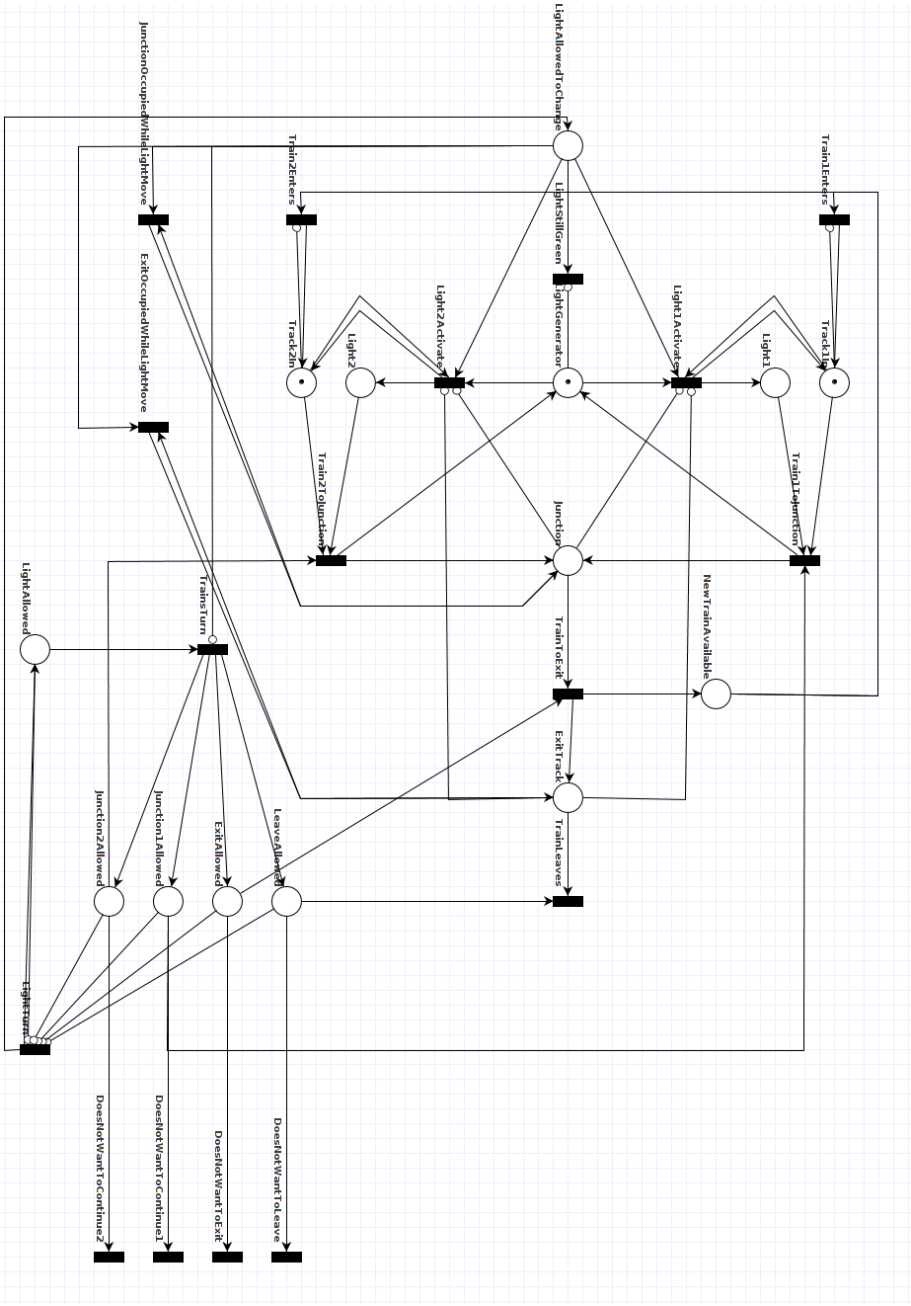


Figure 7:

B TrainsTurn Fired



C Petri Net


LightAllowedToChange
JunctionOccupiedWhileLightMove
Train2Enters
Train1Enters
LightStillGreen
ExitOccupiedWhileLightMove
Track2In
Light2
Light2Activate
LightGenerator
Light1Activate
Light1
Track1In
Train2ToJunction
Train1ToJunction
Junction
LightAllowed
TrainsTurn
NewTrainAvailable
TrainToExit
ExitTrack
TrainLeaves
Junction2Allowed
Junction1Allowed
ExitAllowed
LeaveAllowed
LightTurn
DoesNotWantToContinue2
DoesNotWantToContinue1
DoesNotWantToExit
DoesNotWantToLeave