Modelling of Software Intensive Systems

Assignment 4: Petri Nets

December 11, 2023

Thomas Gueutal
s0195095@ad.ua.ac.be

Felix Vernieuwe
s0196084@ad.ua.ac.be

Outline

0. Introduction **2**
0.1. Important Notes 2
0.2. Project Tree 2
0.3. Tools 2
0.4. Changes since last submission 3
1. Petri-Net Design (I) **3**
1.1. Problem Description 3
1.2. Roundabout Analysis 6
1.3. Unordered Non-Sequential Design 7
1.4. Clocked Design (D) 12
2. Scenario Simulation (II) **19**
2.1. Two cars cycle simulation (A) 19
2.2. Three cars sequentially entering simulation (B) 20
3. Graph Analysis (III) **21**
3.1. Reachability and Coverability (A) 21
3.2. Invariant Analysis (B) 23
4. Query Analysis (IV) **27**
4.1. Boundedness (A) 27
4.2. Deadlock (B) 30
4.3. Liveness (C) 33
4.4. Fairness (D) 35
4.5. Safety (E) 36
A. Appendix **37**
A.1. Old clocked roundabout version 37

0. Introduction

This report details our findings and solutions for assignment 4, concerning the inplementation of a Petri Net given a description, turning it into a discrete-step machine by introducing a clock controller, and then analyzing the resulting diagram on various aspects, such as its reachability/coverability, invariants, boundendness, etc.

The assignment is made for the course *Modeling of Software-Intensive Systems* at the University of Antwerp, with Rakshit Mittal as the TA for this specific assignment.

0.1. Important Notes

For this assignment, no additional packages need to be installed, aside from the ones used in the provided `RC.py` script.

All drawings and diagrams are generally also included in the report, in the caption for each image, you will be able to find the TAPAAL file where the simulatable diagram can be found (this is given as TAPAALFILE.TAPN/COMPONENT). In general, you will be able to find the TAPAAL files under `~/models/*`.

Further, each diagrams is also rendered as a SVG directly into the PDF, so they should be infinitely zoomable. In case this does not work properly, these are also separately available in the `~/diagrams/*` folder of the submission archive, following the same naming scheme.

When we make hypotheses and/or assumptions in a task, we will clearly mark this by prefixing **ASSUMPTION/HYPOTHESIS** to it.

For navigating the project and figuring out to which task each (sub)section belongs to, we have suffixed either roman numerals (I) and alphabetic characters (A) to the relevant sections. These correspond to the numbers (1.) and sub-bulletpoints (◦) found in the `Tasks` section of the assignment.

0.2. Project Tree

Our submission archive is structured as follows:

- `/analysis/` : Contains the generated reachability and coverability dot files
- `/assignment-files/` : Assignement description, and provided images and scripts
- `/diagrams/` : Cleaned up DrawIO diagrams of the TAPAAL models
  - `/building-blocks/` : Basic structures used to form the models
  - `/invariant-visualisation/` : Visualisation of the invariant paths
  - `/roundabout-basic/` : Roundabout without the clock system
  - `/v1/` : First attempt at clocked roundabout
  - `/v3/` : Final clocked roundabout solution
- `/graphs/` : Generated coverability and reachability graphs
- `/models/` : Contains all TAPAAL models generated for completing the tasks
- `/report/` : Folder that contains the report(s)
- `/traces/` : All simulation traces generated for completing the tasks

0.3. Tools

0.3.1. TAPAAL

We will use the [TAPAAL]() tool version 3.9.5 for its modelling, simulation and analysis features.

Note that TAPAAL has support for LTL and CTL query-based Petri Net analysis, in the form of its [verification functionality](), though the official verification documentation seems to be outdated or simply incomplete.

0.3.2. DrawIO

[DrawIO]() was generally used to create cleaner versions of the TAPAAL Petri Nets.

0.3.3. Typst

This report was entirely written in [Typst]().

0.4. Changes since last submission

Postscriptum note: The reason the report is this belated, is due to us underestimating how much we would need to write. **No changes were made to the code or to the models, only to the report.**
What was changed since the previous submission (old report included as reference under `~/report/` :

- Better describe the project structure
- Fixed spelling mistakes and sentences, reworded sentences and added better explanations
- ⚠ Add the missing sections (specifically: **1.4.5.**, **2.**, **3.1.** and **4.4.**)

1. Petri-Net Design (I)

In this section, we describe the design and development process of creating the Petri Net for the requirements given by the problem statement. First, we will delve into all the intricacies of the problem, and make an attempt at breaking it down into logical components. Further, we will convert this set of statements into a cohesive PetriNet, without any notation of ordering or duplicate actions. Finally, we will introduce a timer/clock component — which allows us to run the roundabout in discrete macro and micro steps.

1.1. Problem Description

Our task is to model a roundabout with *two* inroads and *two* outroads, henceforth called INPUTS and OUTPUTS respectively. These roads are connected to a roundabout – called the CORE of the roundabout – where cars will be able to cycle till they reach their desired exit.

The following subsections will break down the details as given in the textual requirement, into more digestable chunks, motivate any ASSUMPTIONS and HYPOTHESES made, and describe any deviations made from the convential names.

1.1.1. Inputs

The assignment states the following requirements for the inputs:

Regular description

- *"There are **two ways to enter** the roundabout (East and West)."*
  - ⟶ We add two inputs: East and West.
- *"[...] but the **number of vehicles at an input is unbounded**."*
  - ⟶ The inputs have infinite car capacity (or short: carpacity).
- *"NOTE: By input, we mean the **producer/generator pattern** (as described in the lectures)"*
  - ⟶ The producer/generator pattern will be described in a later section (see **Figure 2**).
- *"These inputs [...] model the environment of our System under Study."*

Clock description

- *"On **every input**, a **new vehicle may appear** or not (**non-deterministically**)."*
  - ⟶ There are two possible actions: generating or *not* generating.

1.1.2. Inroads

Regular description

- *"The [two] **road segments** that are a **buffer** between the **producers**[...] and the **core road segments** are called the [...] **East/West inroads**."*
  - ⟶ **Nomenclature**: we will generally call these the Inroads

Clock description

- *"A vehicle on a **West/East inroad** **should** **enter the core**, if the corresponding core road segment is empty."*
  - ⟶ Inverse statement: iff the next segment is full, the car *may* not move.
  - ⟶ **Assumption**: we understand the 'should' as a car on the input *deterministically* wanting to enter the core, if there is no car already on it
- *"However, **preference** should be **given to a vehicle that is on the corresponding North/South core road segment** if that vehicle wants to move to that East/West core segment."*
  - ⟶ Cars on the core have priority for moving into the segment, this should be implemented using clocks.

1.1.3. Core

Regular description

- *"The 'core' of the roundabout is made of **four road segments**, each road-segment can **hold upto one vehicle**."*
  - → Each segment (one for each cardinal direction) has a maximum capacity of a single car
- *"Vehicles are free to **exit the roundabout** in the South or North direction (**non-deterministic**). [...] a vehicle that enters from the West may exit the roundabout [...] or continue looping through the roundabout core."*
  - → There are two possible actions: exiting roundabout (if N/S) or moving to next segment.
  - → **Assumption**: in this assignment, we maintain right-hand driving rules, as this makes most sense to the both of us, and the diagram shows cars driving in a counter-clock wise manner around the roundabout.

Clock description

- *"The 4 road-segments that form a cycle in the centre of the roundabout are collectively called the 'core' of the roundabout."*
  - → **Nomenclature**: we will call the North and South segments the Output-adjacent Core Segments, and the East/West segments: Input-adjacent Core Segments for brevity's sake we will drop the "-Adjacent" part.
    Note that this should not be confused with the Inroad/Outroad Segments.
- *"If there is a vehicle on either of the North or South road segments of the core, it may either move to the corresponding outroad (if empty) or to the next East-West road segment of the core, if empty (non-deterministic)."*
  - → When a car is on a Core-Output Segment, it may choose between continuing to the next segment, or exiting the roundabout (if possible).
- *"If there is a **vehicle** on either of the **East or West road segments** of the core, it **has to move** (deterministically) to the North or South road-segment of the core respectively, of-course only if the North/South road segment does not already contain a vehicle."*
  - → When a car is on a Core-Input Segment, it **must** go to the next segment (if possible).

1.1.4. Outroads

Regular description

- *"Vehicles are free to exit the roundabout in the South or North direction (non-deterministic)."*
  - → Cars can non-deterministically exit the core and go on outroads.
- *"If there is a **vehicle on an output**, no other vehicle can leave on that output i.e that **output is blocked**."*
  - → **Assumption**: Outroad has a maximum capacity of one.

Clock description

- *"The [two] road segments that are a buffer between the [consumers] and the core road segments are called the North/South outroads [...]."*
- *If there is a vehicle on either of the North or South road segments of the core, it **may either move to the corresponding outroad** (if empty) or to the **next East-West road segment of the core**, if empty **(non-deterministic)**.*
  - → **Assumption**: In general, for non-deterministic actions, we will always add a skip action, even if the car is able to move to the next segment.

1.1.5. Outputs

Regular description

- *"There are **two ways to exit** the roundabout (North and South)."*
- *"There can be **at most one** vehicle at an output, [...]. by output we mean the consumer/sink pattern [...]."*
  - → The consumer/sink pattern will be described in a later section (see **Figure 3**).

Clock description

- *"On every output that contains a vehicle, the **vehicle present can disappear** or not (non-deterministically)."*
  - → There are two possible action: consuming or *not* consuming (if a car is present), otherwise skip

1.1.6. Summary

As the above sections contain quite a bit of information, and it is easy to miss or forget a particular property, we will briefly summarize all the properties for our roundabout.

Table 1 shows for each component whether it has *deterministic* or *non-deterministic* actions. In the table, we are temporarily ignoring the fact that each of the actions also has a set of requirements that need to be fulfilled — non-deterministic actions could obviously become deterministic if all other actions are unavailable. Case in point: car on roundabout not being able to go to output or next segment due to both already containing a car.

In general, we will always give non-determinstic actions the option to skip moving/generating/etc., as we were not entirely sure whether the non-determinism exists in having the option to *choose*, or the option *to do nothing*.

Deterministic	Non-Deterministic
• Inroad-segment: Merge into roundabout • Core-inputSegment: Drive to next segment	• Inputs: Produce car OR Skip • Outputs: Consume car OR Skip • Core-OutputSegment: Drive to next segment OR Move to outroad OR Skip

Table 1: (Non-)Deterministic Actions table

Next, we identify all the different actions and requirements for all parts of the roundabout. This entails listing all possible "events" (or actions), their prerequisite (that is, the conditions required for firing the event), and finally the attributes of the component in general (what property must hold at all times).

Input (e/w)

Events
Arrive Car arrives and heads to inroad
Skip No car arrives

Inroad Segment (e/w)

Events
Enter Car moves to core-input segment
Prerequisites: segment is empty **AND** cars on core-output segments have priority

Stall Car cannot enter roundabout
Prerequisites: core-input segment is full

Input-adjacent Core Segment (e/w)

Events
Move Car moves to next core-output segment
Prerequisites: segment is empty
Stall Car cannot move
Prerequisites: next segment is full

Attributes
- At most one car on segment

Output-adjacent Core Segment (n/s)

Events
Exit Car moves to corresponding outroad segment
Prerequisites: output segment is empty
Move Car moves to next core-input segment
Prerequisites: segment is empty
Skip Car does not move

Attributes
- At most one car on segment

Outroad Segment (n/s)

Attributes
- At most one car on segment

Output (n/s)

Events
Exit Car leaves output via outroad
Skip Car does not leave

Attributes
- At most one car on segment

1.2. Roundabout Analysis

This section exists as a brief bridge between the problem statement and our first design solutions. With this, we hope to have a better understanding of the dynamics of the roundabout, offer a clear reference for the components/agents of our system, and generally describe some possible problems that might occur when translating the requirements into the Petri Nets model.

Using **Figure 1**, we see that the cyclic nature of our roundabout will have interesting implications for liveliness, as cars can keep moving around the core, never exiting the rotary.

Furthermore, we also need to be mindful of the fact that the transitions within the simplified graph are not necessarily determenistic, as discussed in the previous section. Arcs between our places in the final graph will need to be constructed such that the deterministic property of the action is not violated, but not so strictly as to allow for potential deadlocks to occur. This should be done while minimizing the amount of redundant transitions and places.

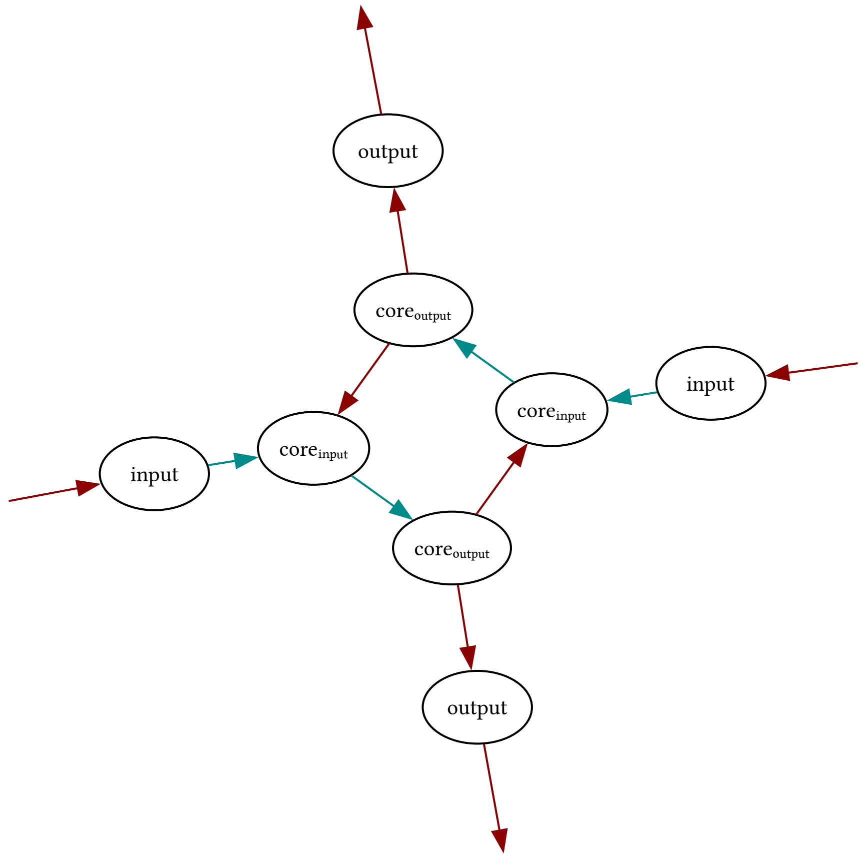

Figure 1: Flow of roundabout in simple notation

Still in **Figure 1**, we visualised the (non-)deterministic nature of each of the actions by applying a color to it. All **non-determinstic** arcs are colored **crimson**, whereas the **determinstic** ones are shown in **cyan**.

Each state in the simplified graph also has a transition leading back to itself, either in the form of a SKIP action (not doing anything), or a STALL action (not *being able* to do anything). We will give a more formal description for either of these actions in **Section 1.3.**

1.3. Unordered Non-Sequential Design

1.3.1. Basic Design Patterns

Before describing and working out the Petri Nets components, we will first briefly construct and discuss their constituent building blocks. For this, we introduce a simple boolean-logic inspired notation for formally describing actions.

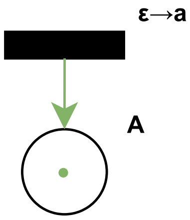

Figure 2: Generator pattern ($\varepsilon \to a$)

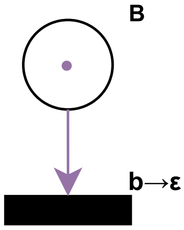

Figure 3: Consumer pattern ($b \to \varepsilon$)

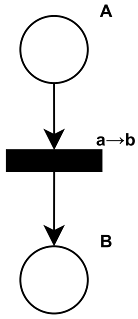

Figure 4: Use pattern ($a \to b$)

Figures 2, 3 and 4 generally show how tokens respectively get generated, consumed and used in a Petri Net, in other words: these three constructs define inputs, outputs and their interconnects.
Next, we need to be able to stitch together these flow patterns to implement our roundabout logic:

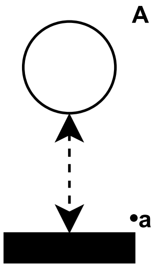

Figure 5: Boolean TRUE ($\bullet\ a$)

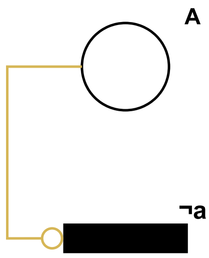

Figure 6: Boolean FALSE ($\neg a$)

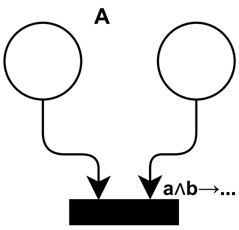

Figure 7: Boolean AND ($a \wedge b$)

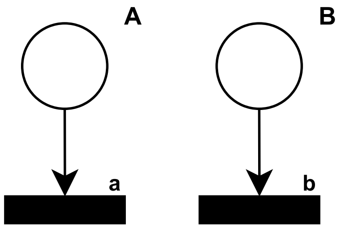

Figure 8: Boolean OR ($a \vee b$)

First, we need to be able to convert the place markers into a boolean statement. In Petri Net notation, an arc from $A \to$ action $\to$ …, states that an action may only fire when A has sufficient markers, this is the general Use pattern. If we only want to detect *whether* or not a given statement is TRUE, we obviously do not want the tokens to be consumed. To accomplish this, we simply add a double-sided transition ↔, as shown in Figure 5 — this transition can only fire if there is a token in A, and when it does fire, it will first consume and subsequently return this token.

In short: the difference between a and $\bullet\ a$ is that the former *uses* the a token, whereas the latter only *checks* whether the a token exits.

Boolean FALSE can simply be seen as an inhibitor arc from the place that should *not* have any markers, to the transition that should be blocked, if this previous statement is false (see Figure 6).

Finally, we can combine these statements using the AND and OR constructions given by Figure 7 and Figure 8.

A short example that combines all these constructions is given in Figure 9. In general, we will only be using the boolean logic to validly *construct* the diagrams, the boolean statements that we used will not be included within the diagram.

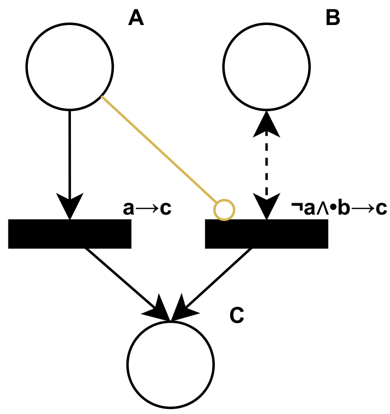

Figure 9: Petri net representing $(a) \lor (\neg a \land \bullet b) \rightarrow c$

One final note and disclaimer, all of the diagrams shown within this assignment were separately made in DrawIO instead of TAPAAL, as the authors of this assignment strongly object to the alignment and grid implementation found within the latter program. After hundreds of attempts to get a set of blocks aligned properly, while simultaneously pleading to – and cursing at – the pantheon of UIX gods, we decided to forego making the TAPAAL diagram pretty, and focus our efforts to DrawIO.

1.3.2. Inputs and Outputs (A)

Finally, we can move onto the actual implementation of the Petri Net, starting with the simplest components of them all: the inputs and outputs.

Luckily, this part is rather straight-forward: we simply need to make use the patterns described in Figure 2 and Figure 3, and then make them non-deterministic by adding a `skip_X` action for each direction — which is equivalent to a no-op. Note that we are already making use of SHARED PLACES formalisms in order to be able to cleanly link the different places with eachother.

Currently, this component is *not* conform to the requirements, as multiple cars can arrive on INPUT_W in the same tick, as the actions aren't hooked up to a clock. However, if we temporarily assume that this is done (i.e only one action for each discrete timestep), then our inputs —resp. outputs— are properly non-deterministic, and have the choice between generating/consuming tokens (if allowed) or skipping.

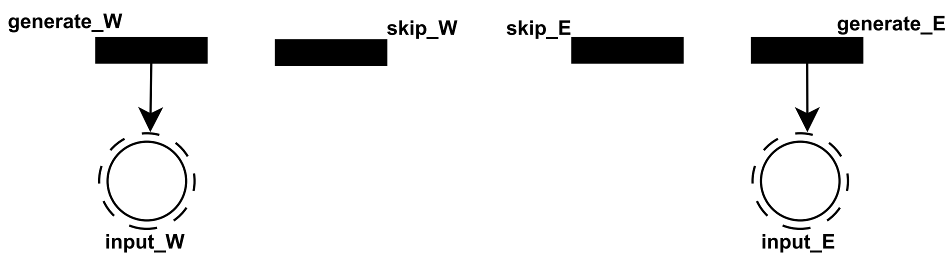

Figure 10: Clockless roundabout inputs (BASIC-ROUNDABOUT.TAPN/INPUTS)

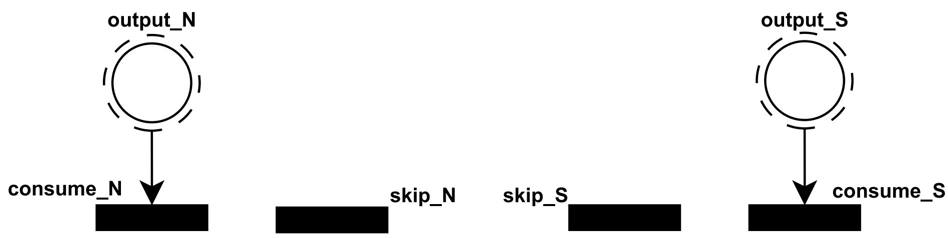

Figure 11: Clockless roundabout outputs (BASIC-ROUNDABOUT.TAPN/OUTPUTS)

Noteworthy for later on: in the output, we dodged the issue of potential deadlocks by virtue of always being able to SKIP consuming. If there isn't any car on the output, then obviously CONSUME_X cannot be run (as a token is required).

1.3.3. Inroads/Outroads (C)

Connecting the inputs with the core, are the in -and outroads of the system. Here, we have both a non-determinstic and determinstic actions. Let's first briefly discuss the non-determinstic one found in Figure 12.

As noted in Section 1.1.4., the car may non-deterministically choose to exit the roundabout, or continue driving. We additionally added the intepretation (assumption) that being *non-determinstic* also gives you the option of *not* moving and just staying in place. One interpretation of this, is a driver near an outroad waiting for the output to clear. If the non-determinism is instead solely understood as being able to *either* move or exit, then a separate stall action needs to be added (see Figure 13).

In the simple case, however, we state that you can *only* take the exit if the output is empty. Using our previously mentioned boolean logic, this is equivalent to $\text{core}_N \land \neg\text{output}_N \rightarrow \text{output}_N$ (idem for S). In layman's terms: you can EXIT iff there is a car (a token on core_N) AND there is not already a car on the exit (no token on destination output_N).

As with the inputs, you always have the option to *not* enter the core and stay in place. Themovement logic will be described in Section 1.3.4.

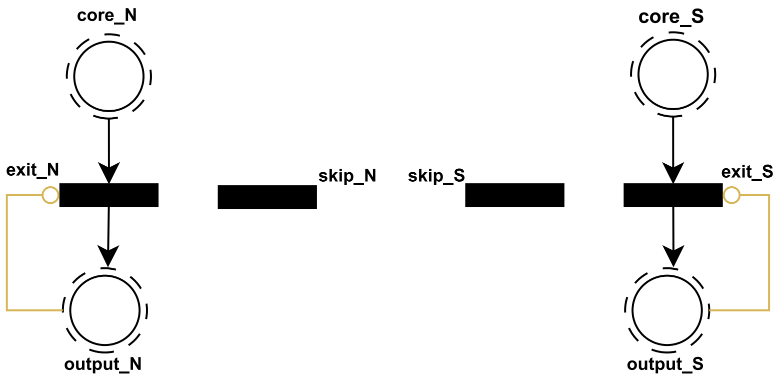

Figure 12: Clockless roundabout outroads (BASIC-ROUNDABOUT.TAPN/OUTROADS)

Next, Figure 13 shows the logic for *deterministically* merging a car into the roundabout. First, we need to define what we precisely understand under "deterministic movement":

*In **Deterministic movement**, for any combination of system states in a (sub)component, only **a single** transition is available at any time.*

In short: all transitions should be mutually exclusive with one another when we wish to move deterministically. To accomplish this, we need to first figure out and separate the different system states from one another, and figure out which transitions are allowed to fire.

input_X	core_X	Interpretation	Enabled Action
0	0	No car prepared to merge	SKIP
0	1	No car prepared to merge	SKIP
1	0	Car wants to merge, segment free	ENTER
1	1	Car wants to merge, segment blocked	STALL

Table 2: Actions table for Inroad

As seen in Table 2, we must SKIP if there is no car available to merge, STALL if a car can merge but the corresponding core-input segment is blocked, and MERGE otherwise (if there is a car ready on input, and the adjacent segment is not currently occupied).

In simplified boolean logic, where each action is mutually exclusive, this may be given as:

- SKIP: $\neg \text{input}_X$ *(skip only if there isn't a car on the input)*
- STALL: $\bullet\ \text{input}_X \land \bullet\ \text{core}_X$ *(stall if both segments are occupied, only* ***checking*** *if true)*
- ENTER: $\text{input}_X \land \neg \text{core}_X \to \text{core}_X$ *(enter if destination segment is free, actually* ***move*** *tokens)*

Converting these statements into the Petri Net formalisms (using the translation methods and structures specified in Section 1.3.1.), we finally end up with the figure found below:

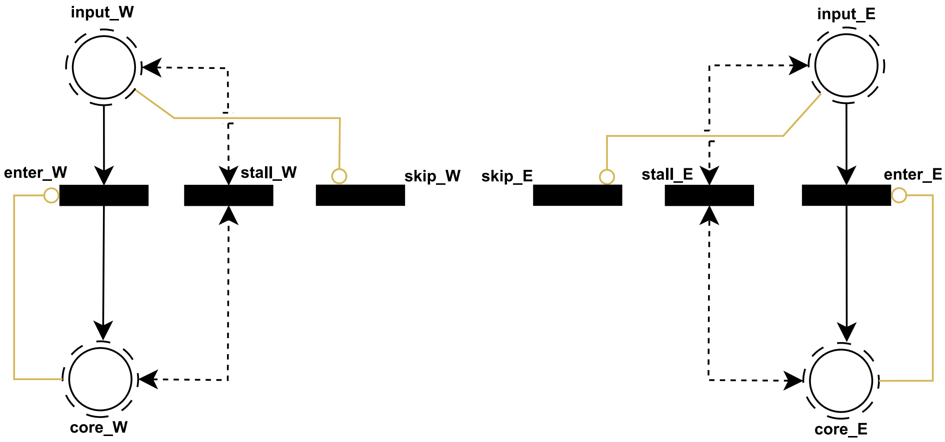

Figure 13: Clockless roundabout inroads (BASIC-ROUNDABOUT.TAPN/INROADS)

1.3.4. Roundabout (B)

To conclude this section, we showcase the design of the roundabout. It combines several aspects we already discussed, such as non-deterministic actions and determinstic movement.

Harkening back to Figure 1, the outputs of the outroad-adjacent core segments should be non-deterministic, or in other words: having the option between MOVE and SKIP/NO-OPS (STALL *could* also be modelled as a separate action, but since it has no semantic difference in our model, we fold it under the SKIP action). The transition structure of these segments will thus look a lot like the ones already shown in Figure 12, switching EXIT for MOVE and OUTPUT_N for the counter-clockwise adjacent segment.

We can make the same observation for the discrete movements: the set of available actions and their prerequisited is generally the same compared to the one that was constructed in the previous section (see Table 2), we will copy and properly adapt it for completeness' sake:

core_X	core_Y	Interpretation	Enabled Action
0	0	No car wants to move	SKIP
0	1	No car wants to move	SKIP
1	0	Car wants to move, segment free	MOVE
1	1	Car wants to move, segment blocked	STALL

Table 3: Actions table for Core_input

Combining both the North-South $\text{Core}_\text{output}$ segments, and the East-West Core_input segments, we end up with the diagram given as Figure 14.

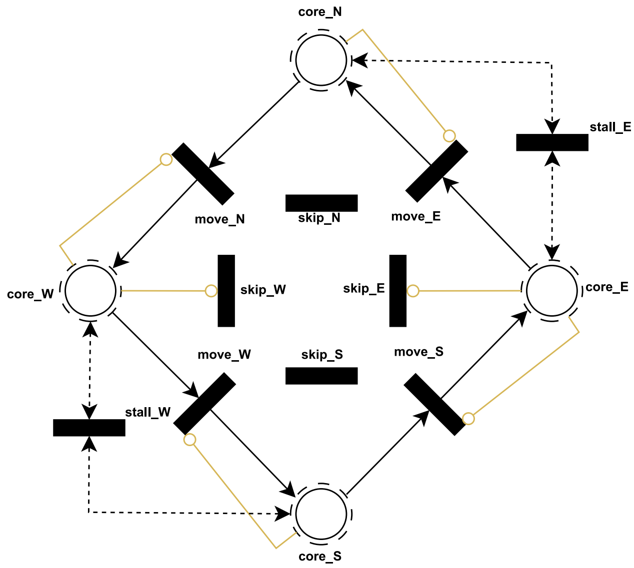

Figure 14: Clockless roundabout core (BASIC-ROUNDABOUT.TAPN/CORE)

1.3.5. Full roundabout diagram (C)

Finally, we can combine all the separate parts of our roundabout into a single model. This is only done for demonstrative purposes, and isn't necessarily correct — i.e.: it is not possible to have two identically named transitions in the same diagram. Furthermore, there are a few redundant transitions, such as the double SKIP$_{N}$ transition in CORE$_{N}$. The Inputs/Outputs are attached to the sides of the core.

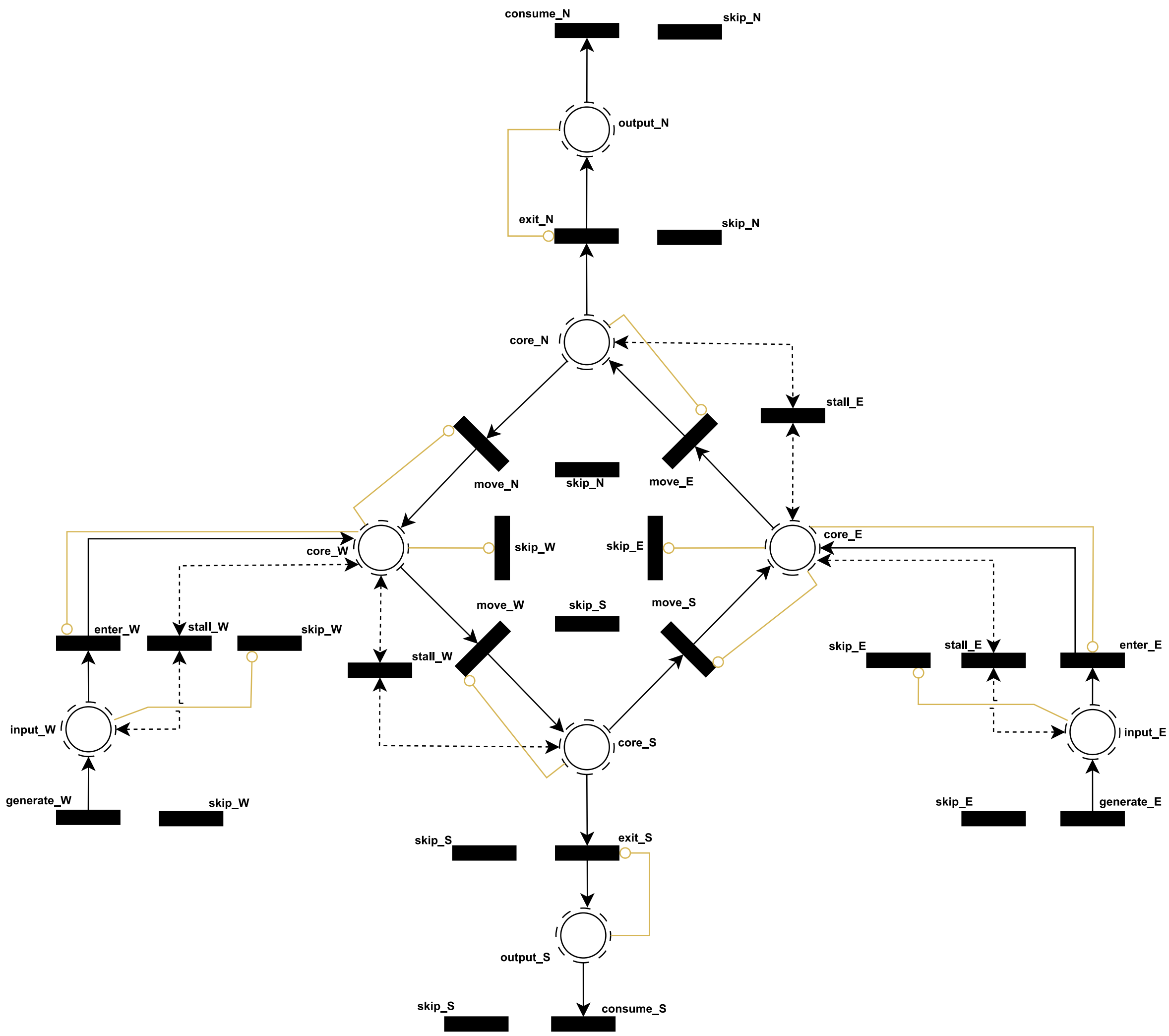

Figure 15: Clockless full roundabout

1.4. Clocked Design (D)

While the full roundabout given by Figure 15 satisfies (almost) all requirements posed in the assignment – and semantically half-correct – it does *not* have any notion of time or operation ordering. Without this property, in principle hundreds of cars could arrive at the roundabout one after another, till a hypothetical car on the core gets to move a single time.

In this section, we will thus describe how we tackled both the ordering of steps (which will be used to fulfil the last remaining requirement: merging priority), and how we restricted actions to occur exactly once every arbitrary timetick. As we will soon see, these two ideas are rather intertwined, and we will be able to solve the two with the help of a single component, and some additional basic constructs.

1.4.1. Basic Design

As stated in the assignment:

"NOTE: The above "micro-steps" are NOT given in a predefined order.
The clock is important because it restricts the movement of each vehicle to a single position in every major clock tick (unless the vehicle can/does not move)."

We need to design and implement a clock such that in each macrostep (a clocktick), a series of microsteps (instructions) are executed in a specific order, such that an arbitrary car in our system is only able to move *once* each tick of the tock of the clock.

Initially, we misunderstood this exercise, and assumed that the steps could be run in an arbitrary order — resulting in quite a bit confusion when we got to trying to implement the priority circuit. In Figure 32, we have an old, incomplete, version of the roundabout with a different (and much less clear) interpretation of the requirements.

We will again introduce a couple constructions that will outline how the clock was implemented:

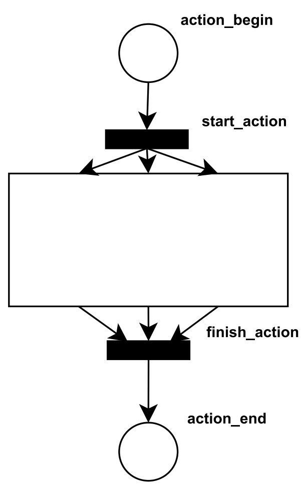

Figure 16: Parallel execution pattern

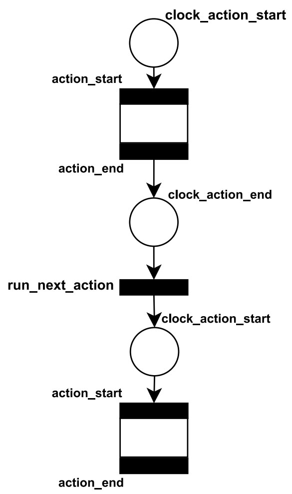

Figure 17: Sequential execution pattern

The core idea behind our clock, is that we will group actions together such that the the requirement of a given car only being able to move *once* within a single macrostep, is fulfilled. Thanks to good foresight (or rather: backsight organisation), we have already combined each of the actions into separate components.

On this set of components, we will need to apply the following two constructions for each component:

1. Apply the parallel execution pattern for components that exist twice in the roundabout (These should be able to be run in any order, EXAMPLE: E/W input generation)
2. Apply the OR pattern for actions that are mutually exclusive from one another (EXAMPLE: input generation may either generate a token, or skip, but only one of the two)

However, before we can do this, we must first find a suitable order for executing the components in. From the problem description, we already know that a car on the roundabout has *priority* over a car on the inroad. Put differently: the actions of the inroad component **depend** on the core movement. Besides, a car on the inroad should not move if a car in the core will be moving towards the same segment.

We can put this more broadly still: in general, we have a dependence relationship if by executing the actions of a component A before component B, there exists a situation where a car was able to move twice in the same macrostep.

For instance: say we run the actions of the INPUTS component — which spawns a car in $\text{INPUT}_{\text{East}}$, and afterwards execute the transitions in the INROAD, then there is the possibility that the car that just arrived, is also able to immediately merge into the roundabout. Or summarised: INPUTS depends on INROAD. Formally: INPUTS $\longrightarrow$ INROADS.

Repeating this logical reasoning for all other components, this results in following dependency chain:

$$\text{INPUTS} \longrightarrow \text{INROADS} \longrightarrow \text{CORE} \longrightarrow \text{OUTROADS} \longrightarrow \text{OUTPUTS}$$

This is nothing more than taking the logical path of the car throughout the roundabout. Note that this dependency chain should be used in a reverse order if we want to prevent cars moving multiple times.

To recap: we have five unique components, which we should parallelize such that they can start from any cardinal direction. Next, we need to order these components such that we can eliminate the possibility of a car moving twice in the same tick. For this, we determined a dependency graph, which will determine in which manner they will be visited.

Finally, we rename the components as follows:

- **PHASE 1:** Outputs — consuming cars on the output
- **PHASE 2:** Exit Core — exiting the core of the roundabout (formerly: OUTROADS)
- **PHASE 3:** Move Core — cycling through the roundabout
- **PHASE 4:** Enter Core — merging into the roundabout (formerly: INROADS)
- **PHASE 5:** Inputs — producing cars on the inputs

Now that everything is properly defined and layed-out, we will again run through each of the component to see how they were adapted.

1.4.2. Inputs (PHASE 5) and Outputs (PHASE 1)

As already discussed in the previous section, we first parallelize the each of the components by adding a CONSIDER$_X$ and FINISH$_X$ state for each cardinal direction in the component. This structure exists for allowing *a single* action to be picked in either direction. The CONSIDER$_X$ is connected to each of the actions and receives a single token from transition ACTION$_{START}$ at the beginning of the microstep. Executing any of the actions *requires* this token from the ACTION$_{START}$. Note that this enforces the rule that an action may only be executed once for every clock tick (as we will later construct a clock such that CLOCK_ACTION$_{START}$ only receives a single token).

Once either of the available actions have been executed for *each* of the directions (and both FINISH$_X$ places contain a token), only then can the microstep end by calling the ACTION$_{END}$ transition.

For the final result, see the figures below (nr. 18 and 19).

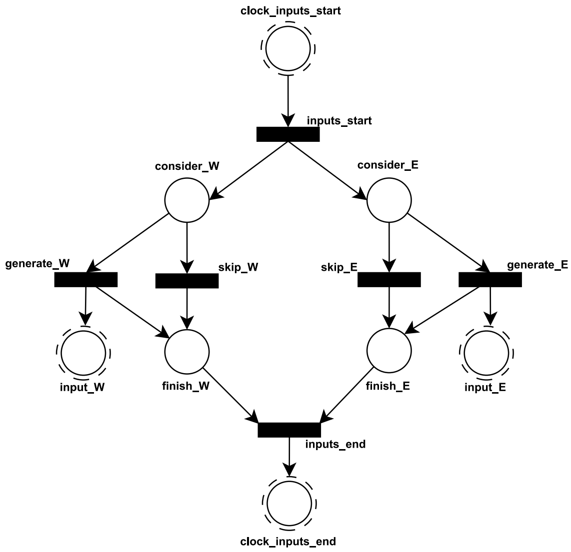

Figure 18: Final roundabout inputs (ROUNDABOUTMODEL-V3.TAPN/PHASE_5_INPUTS)

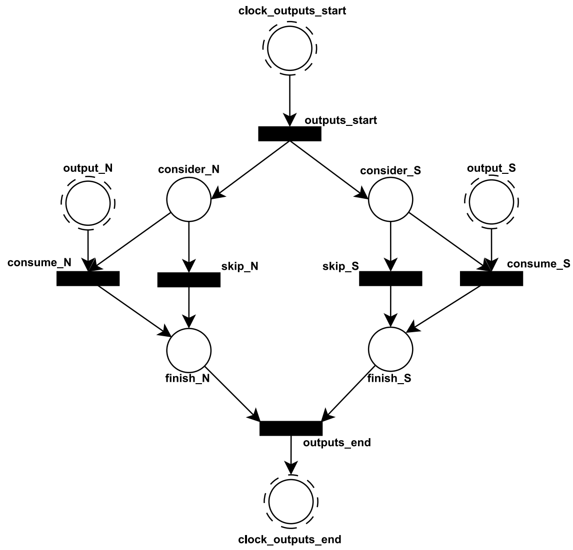

Figure 19: Final roundabout outputs (ROUNDABOUTMODEL-V3.TAPN/PHASE_1_OUTPUTS)

1.4.3. Inroad/Enter Core (PHASE 4) and Outroad/Exit Core (PHASE 2)

The inroad and outroad models are luckily quite similar in structure to the inputs/outputs components. The only major difference is found in the inroad, where we now have to consider three different actions, though this is functionally the same (besides the fact that it is ever so slightly more difficult to connect up tidily).

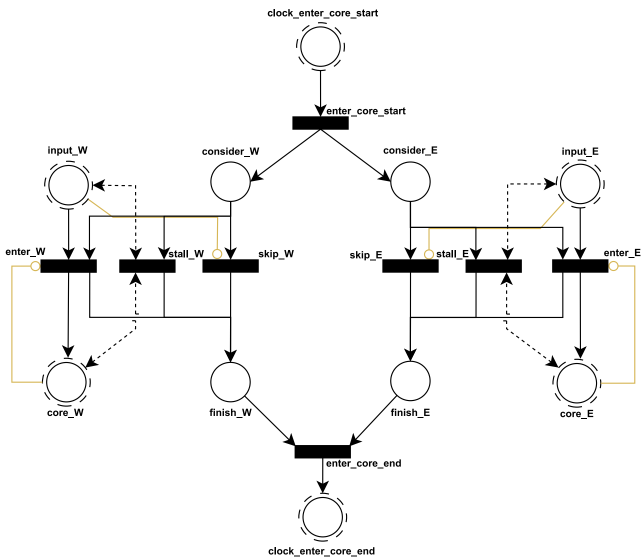

Figure 20: Final roundabout inroad (ROUNDABOUTMODEL-V3.TAPN/PHASE_4_ENTER_CORE)

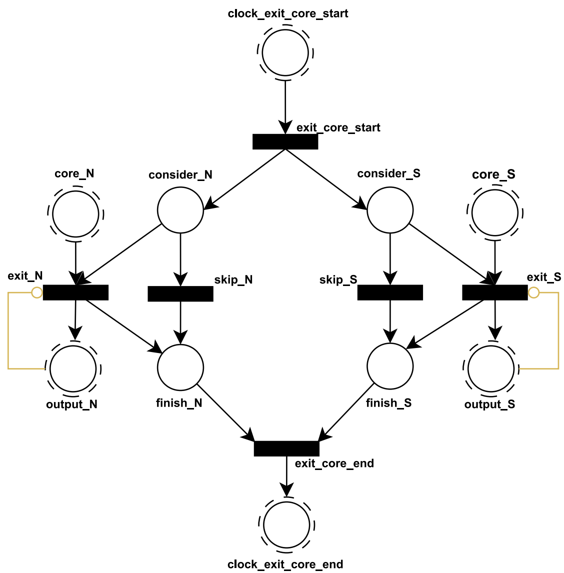

Figure 21: Final roundabout outroad (ROUNDABOUTMODEL-V3.TAPN/PHASE_2_EXIT_CORE)

1.4.4. Roundabout

The design of the core roundabout is much more involved than all those seen in the previous sections, as we need to find solutions to two entirely different problems:

1. How do we move all the cars in the core *once*?
2. How can we do this while still having a clean and easily digestable diagram?

Our solutions started with trying to work out how (1.) could be implemented: multiple ideas were tested out (parallel core movements, folded states, etc.), until we found a logical simplification for our problem.

Essentially, we made the observation that there are three possible cases for our core:

1. The core is empty ⟶ Trivial, we don't need to do anything — Action: SKIP
2. The core is entirely full ⟶ None of the cars will be able to move — Action: STALL
   ASSUMPTION: if the entire core is filled with cars, either all cars are stalled, or all cars move once. However, since the Petri Net is not able to distinguish individual tokens, the end result is the same. As such, we just consider this as a STALL.
3. The core is partially filled ⟶ There is *atleast* a single free spot within the core (at most three, as four empty spots would result in a SKIP action). We start our core movement cycle at this empty spot — Action: CYCLE

The intrinsics of the core movement will be discussed later on, but for now, let's briefly configure a truth table for each of the possible input configurations.

core_N	**core_E**	**core_S**	**core_W**	**Interpretation**	**Enabled Action**
0	0	0	0	No car in the roundabout	SKIP
0	0	0	1	Empty spot NORTH, EAST and SOUTH	CYCLE
0	0	1	0	Empty spot NORTH, EAST and WEST	CYCLE/SKIP
…	…	…	…	…	…
1	1	1	0	Empty spot WEST	CYCLE
1	1	1	1	All segments are blocked	STALL

Table 4: Actions table for Core Cycle Manager

While CYCLE is by far the most important operation, it might be interesting to first briefly discuss SKIP and STALL.

- **SKIP:** $\neg\text{core}_E \land \neg\text{core}_W$ — we may only SKIP if there isn't a car currently in the onroad (as this car should always determinstically move to the next segment).
  NOTE: CORE_N and CORE_S are not included in this boolean statement, as cars on this segment are allowed to non-deterministically skip moving. This is generally a shorthand for not having to go through the entire cycle code. The only case where the skip is strictly necessary, is when there are *no* cars on the roundabout.
- **STALL:** $\bullet\,\text{core}_N \land \bullet\,\text{core}_E \land \bullet\,\text{core}_S \land \bullet\,\text{core}_W$ — we may only STALL when all places of the core contain a token.

If, instead, the enabled action is CYCLE, then we have a non-deterministic choice in which $\text{ACTION\_START}_\text{x}$ transition we select. Any empty segment of our core is a candidate starting point for our core cycle.

The chosen $\text{ACTION\_START}_\text{x}$ transition will consume the token in our clock, and then pass it to the corresponding $\text{ALLOW\_ACTION}_\text{x}$ place.

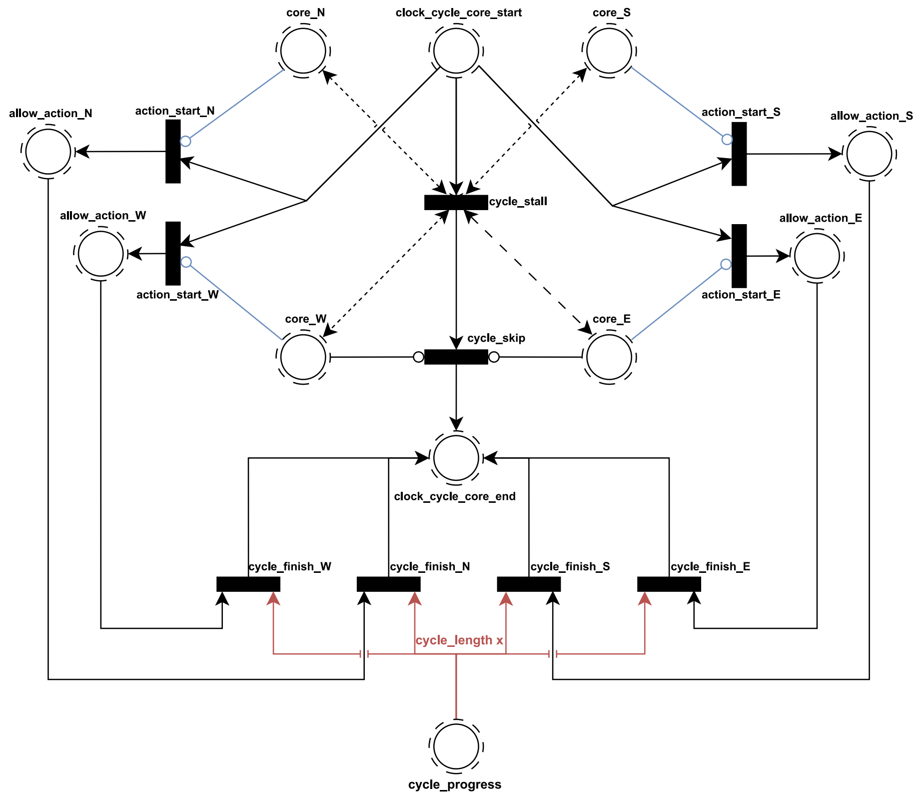

Figure 22: Final roundabout core manager
(ROUNDABOUTMODEL-V3.TAPN/PHASE_3_MANAGE_CORE)

Now that the main ideas behind to cycle manager are explained, we can discuss what exactly the ALLOW_ACTION$_x$ place is used for. Figure 23 contains all the movement logic we will discuss in the paragraphs below.

On a high-level view, the component consists of an outer and inner ring, where the former represents the basic roundabout movement logic we've discussed in Figure 14, and the latter implements the clock signal (and limiter) for our roundabout.

The entire core is connected with eachother using only three types of transitions:

- **MOVE** (N/E/S/W)**:** Car moves to the next segment
  PREREQUISITE: car on segment with next segment being empty AND action is available
  BOOLEAN FORMULA: $\bullet\ \text{allow\_action}_X \land \text{core}_X \land \neg\text{core}_Y$
- **STALL** (E/W)**:** Car stays in place
  PREREQUISITE: both segment are occupied AND action is available
  BOOLEAN FORMULA: $\text{allow\_action}_X \land \bullet\ \text{core}_X \land \bullet\ \text{core}_Y$
- **CYCLE** (N/S, E/W)**:** Car stays in place
  PREREQUISITE: no car on segment AND action is available
  BOOLEAN FORMULA: $\text{allow\_action}_X \land \neg\text{core}_X$ (E/W directions), otherwise (N/S): allow_action_X

In the inner ring, we cycle a token (the clock token) around our core, which both serves as the tracker of the currently moving car, as well as an ordering for the car movement. Since there is only a single token available in the ALLOW_ACTION cycle at any time – and each action given above requires the presence of this token, we can by extension state that only a single car can execute transitions at any moment.

Of note is that both the STALL or CYCLE actions *move* this token to the **clock-wise** next ALLOW_ACTION$_x$ state, whereas MOVE only checks whether the token exists. This drastically simplifies our graph by not having to add additional control and output signals for our MOVE action. Instead, after taking this MOVE transition, the CYCLE action will run by default to handle moving the ALLOW_ACTION to the next place.

Finally, we'll need to briefly elaborate on why we chose a **clock-wise** cycling process of our ALLOW_ACTION token, as opposed to moving it around in the same direction as our core movement. This second part of the sentence already partially showcases the problem: if we cycle in the same direction as the movement, then the ALLOW_ACTION token is essentially following the car around — and thus allowing for the vehicle to keep acting upon that same token.

So now that we have a control token going around the ALLOW_ACTION cycle, moving it after each car has finished a transition (either MOVING, STALLING or SKIPPING), we now need to add some extra logic ontop of this system, in order to prevent this process from going on indefinitely.

Luckily, this part is the relatively simple: if we want to go through the cycle *exactly* once, we just need to count how many times the clock token was passed along through a ALLOW_ACTION place (or: how many times CYCLE/STALL were called). This simply entails adding an additional transition for every *token cycle action* to a place called CYCLE_PROGRESS.

Then, once we have executed *four* transitions (or our CYCLE_PROGRESS place contains four tokens), we block every action (i.e. MOVE, CYCLE and STALL) with the help of an inhibitor arc. In the diagram found in Figure 23, these blocking arcs are visualised in **red**.

At this moment, the movement logic is considered to be finished, and the manager component should kick in again and finish the phase. Note that our system state currently contains *exactly* 4 tokens in CYCLE_PROGRESS, and *precisely* 1 token in any of the four ALLOW_ACTION places. So to completely clean-up the movement and manage logic, we add four transitions as:

$$\text{allow\_action}_X \land 4 \times \text{cycle\_progress} \rightarrow \text{clock\_cycle\_core\_end}$$

This will clear the CYCLE_PROGRESS for the next macrostep, and convert the ALLOW_ACTION token back into the appropriate clock token.

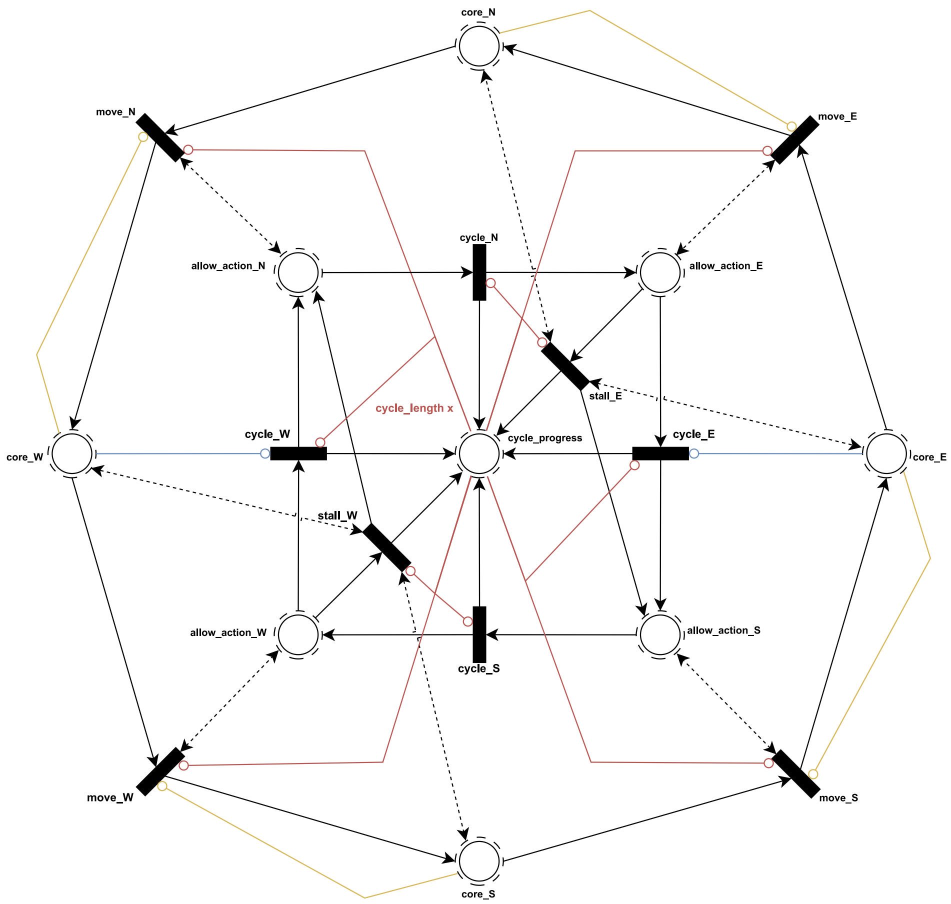

Figure 23: Final roundabout core movement (ROUNDABOUTMODEL-V3.TAPN/PHASE_3_MOVE_CORE)

1.4.5. Clock and Visualisation

In this last section, we briefly detail how the system clock is actually implemented. Recall that we needed to cycle through five different phases in a particular order, in order to validly simulate a single macrostep of our system (OUTPUTS ⟶ OFFROADS ⟶ … ⟶ INPUTS).

As our components internally already implement transition of CLOCK_ACTION$_{\text{START}}$ ⟶ CLOCK_ACTION$_{\text{END}}$ (by virtue of running through their available actions), our clock simply needs to implement the transitions between different phases, or: CLOCK_ACTION_X$_{\text{END}}$ ⟶ CLOCK_ACTION_Y$_{\text{START}}$, following the ordering given above. These phase transitions can be found as RUN_ACTION in **Figure 24**.

Once all phases are completed (the clock token is in CLOCK_INPUTS_END), we need to take a macrostep and repeat the entire process. CLOCK_TICK will move the token back to phase 1, and simultaneously increment the amount of CLOCK_TICKS by one.

Depending on whether our clock token is initially placed in CLOCK_INPUTS_END or CLOCK_OUTPUTS_START, the timer starts at 1 or 0 tokens. In our implementation, we opted for the latter approach (incrementing the tick count *after* all phases were considered).

For running traces (see **Section 2.**), we made heavy of the VISUALISATION component, which combines all the important shared places in a single view. This is given in **Figure 25**.

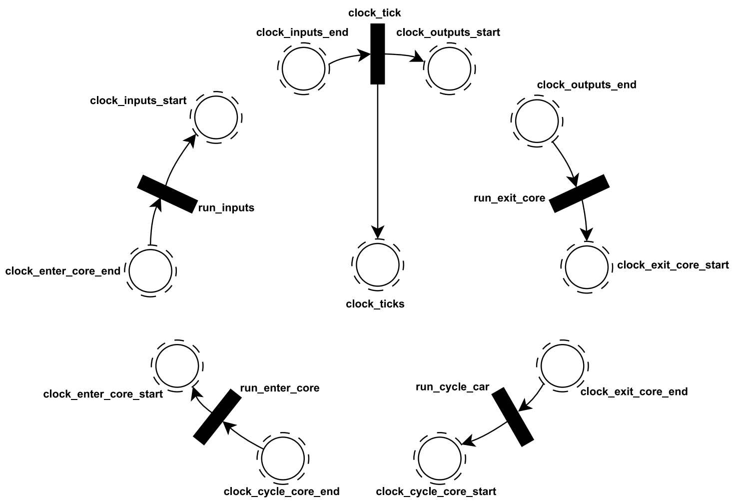

Figure 24: Final roundabout — clock (ROUNDABOUTMODEL-V3.TAPN/CLOCK)

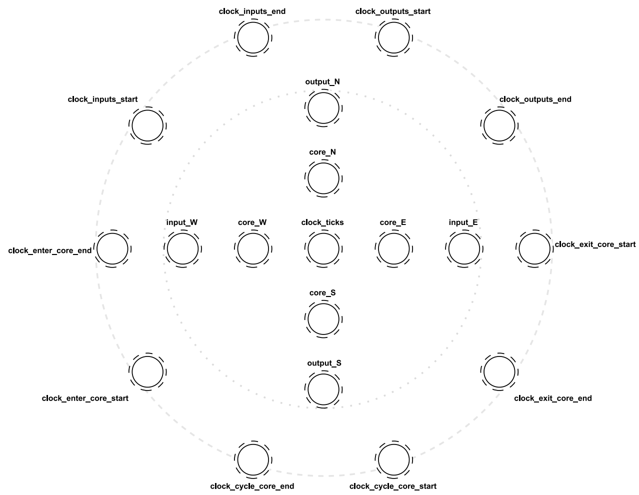

Figure 25: Overview of most important shared places (ROUNDABOUTMODEL-V3.TAPN/VISUALISATION)

2. Scenario Simulation (II)

For this entire section, we will exclusively make use of `roundaboutModel-v3.tapn` to simulate our model and generate the traces. In the submission archive, these traces can be found in `~/traces/*`.

Following general **ASSUMPTIONS** were made when generating the traces:

- (given) Vehicles always try to exit when they arrive at an outroad-adjacent segment.
- Our clock starts at the CLOCK_OUTPUTS_START state (CLOCK_TICKS starts at 0)
- The roundabout starts entirely empty, when the task describes that "*X vehicles arrive at the roundabout (…)*", we take this as the cars *not having arrived yet* (thus, we should generate them in our simulation). This makes the most sense to us, as this would actually showcase that our inputs design was implemented correctly.

2.1. Two cars cycle simulation (A)

The tasks asks us to:

- *"Two vehicle (one East and West)* ***arrive*** *at the roundabout in the* ***same step****.""*
  - → In timestep 1, we should generate a car at each of the inputs.
- *"The* ***simulation ends*** *as soon as both vehicle have* ***fully exited the roundabout****."*
  - → **ASSUMPTION:** we understand "fully exited" as the car token getting removed from the model when doing the EXIT action.

The trace for this simulation can be found in the file `traces/two-cars-in-to-out-truncated.trc`. Instead of discussing the entire process step-by-step, we make a few simplifications:

- We will go over the trace tick by tick, phase by phase.
- The clock actions may be hidden, as these will always be the same for every tick
- Phases where no actions (i.e. only SKIPS or equivalent) were executed, will be completely hidden, as they provide no additional information.
- If the order of an operation is interchangeable, the action will be placed on the same line.

Phase 1: OUTPUTS
- SKIP N/S

Phase 2: EXIT_CORE
- SKIP N/S

Phase 3: MOVE_CORE
- CYCLE_SKIP

Phase 4: ENTER_CORE
- SKIP E/W

Phase 5: INPUTS
- GENERATE E/W

Phase 1: OUTPUTS
- SKIP N/S

Phase 2: EXIT_CORE
- SKIP N/S

…

Table 5: Part of complete trace (showing only actions)

Phase 5: INPUTS
- GENERATE E/W

Phase 4: ENTER_CORE
- ENTER E/W

Phase 3: MOVE_CORE
- ACTION_START N/S
- CYCLE N/S
- MOVE E/W
- CYCLE S/N
- CYCLE W/E

Phase 2: EXIT_CORE
- EXIT N/S

Phase 1: OUTPUTS
- CONSUME N/S

Table 6: Simplified trace for two cars moving

Phase 1: OUTPUTS
- OUTPUTS_START
- *Set of actions*
- OUTPUTS_END

RUN_EXIT_CORE

Phase 2: EXIT_CORE
- EXIT_CORE_START
- *Set of actions*
- EXIT_CORE_END

Phase 3: MOVE_CORE (…)

Phase 4: ENTER_CORE (…)

Phase 5: INPUTS (…)

CLOCK_TICK

Phase 1: INPUTS (…)

…

Table 7: Single clock tick structure (showing only clock events)

Per our clock construction detailed in Section 1.4.1., we can actually intuitively see in Table 6 that our phases are being run through in reverse order. Further, every macrostep/clocktick, only a single movement action gets executed for each car, which gives shows (albeit informally) that our basic design does conform to the requirements.

Finally, we can also inspect (part) of a full clock tick in our model in **Table 7**. This table has mainly been added to showcase when our clock signal gets fired, and when the actions get run. Be aware that the clock management logic for MANAGE_CORE is ever so slightly different, refer to **Section 1.4.4.** for more information.

Our entire simulation finishes in either four or five ticks, depending on whether you consider the simulation to have ended as soon as the EXIT transitions have been executed, or when the CLOCK_TICK runs. We interpreted the requirement literally, such that EXIT immediately leads to the end of the simualtion — but we have also added the other case as a trace for completeness' sake (see: `traces/two-cars-in-to-out-full.trc`).

2.2. Three cars sequentially entering simulation (B)

The second simulation task is described as:

- *"**Three vehicle arrive** on the East input, **enter** the **roundabout** and try to **exit** through the **North output**."*
  - ⟶ The car arrives East, enters CORE$_E$, moves to CORE$_N$, and attempts to exit to OUTPUT$_N$.
- *"However, there is a **vehicle** which **blocks the North output**."*
  - ⟶ OUTPUT$_N$ contains a car that never EXITS.
- *"The **execution ends after the third vehicle has arrived**."*
  - ⟶ **ASSUMPTION:** this indicates to us that each clock tick, a new car *should* arrive.

The resulting trace of this simulation is generated as `traces/three-cars-in-to-out-blocked.trc`. As before, we will list the main steps of this simulation:

Phase 1: OUTPUTS
- SKIP N

Phase 5: INPUTS
- GENERATE E

Phase 1: OUTPUTS
- SKIP N

Phase 4: ENTER_CORE
- ENTER E

Phase 5: INPUTS
- GENERATE E

Phase 1: OUTPUTS
- SKIP N

Phase 3: MOVE_CORE
- ACTION_START N
- CYCLE N
- MOVE E
- CYCLE S, W

Phase 4: ENTER_CORE
- ENTER E

Phase 5: INPUTS
- GENERATE E

Table 8: Simplified trace for three cars entering

The cars arrive one after another, enter the core as soon as possible, and then move to the next segment.

Again, depending on whether the simulation ends the moment the third car ENTERS, or if it still completes the current tick, the simulation finishes in either 2 or 3 ticks.

3. Graph Analysis (III)

3.1. Reachability and Coverability (A)

Reachability is a measure of which states are reachable from an initial marking in the petri net. Coverability adds the notion of one state dominating another to reachability, which allows it to reduce parts of some states to an omega (infinity) symbol. This reduction removes the problem of infinitely increasing markings and makes more states identical, resulting in a usually finite graph.

The following sections make use of `roundaboutModel-v3.tapn` : **Section 3.1.1.**, **Section 3.1.3.**

The following sections make use of `roundaboutModel-v3-reachability-remove-infty.tapn` : **Section 3.1.2.**

3.1.1. Reachability Graph

We claim that an infinite reachability graph is produced. To see this, we need to look no further than the `clock_ticks` place in the **clock** component. After every full macro step, the number of tokens in this place increases by one. This means that the **marking of the `clock_ticks` place increases infinitely**, for as long as the clock keeps ticking. Reachability analysis simply checks which system states are reachable from the initial marking. Due to the inifinitely increasing `clock_ticks` marking, there also exist an infinite amount of such system states. Consequently, an infinite reachability graph is produced.

In addition to the `clock_ticks` place, the `input_E` and `input_W` places also have the potential to accumulate tokens infinitely, as they are attached to the generators.

Generating the infinite reachability graph (see **Figure 26**) results in an image that consists of a single DFS-esque (Depth First Search) branch. This is because the clock tick count differentiates all states that would otherwise be identical. The monotonically increasing clock tick count makes it impossible for two states that were reached during two separate macro steps to be identical.

Note that the infinite reachability graph is barely visible due it consisting of a single, extremely long branch. The corresponding dot file is provided in the submission archive.

Figure 26: Reachability graphs (infinite and finite)
(GRAPHS/REACHABILITY-INFINITE and GRAPHS/REACHABILITY-FINITE)

3.1.2. Altering Reachability

We can make two minor changes to the original model, `roundaboutModel-v3.tapn` , which results in the changed model `roundaboutModel-v3-reachability-remove-infty.tapn` . These changes will make the reachability graph finite.

See **Figure 26** for the reachability and coverability graps of the adjusted petri net model.

The **first change** is in the **clock** component. We disconnect the directed arc from the `clock_tick` transition to the `clock_ticks` place. This way the clock counter value is always constrained to 0.

The **second change** is in the **phase_5_inputs** component. We add inhibitor arcs from place `input_W` to transition `generate_W` and from place `input_E` to transition `generate_E`. This way the input places are constrained to one car/token at most.

The two changes together **remove all sources of unbounded token generation**. This constrains the amount of possible states to a finite amount, such that the corresponding reachability graph becomes finite as well.

3.1.3. Coverability Graph

We consider the coverability graph for the original model, `roundaboutModel-v3.tapn`.

We identify a split-and-join pattern in Figure 28, which is the coverability graph generated for the original model. Figure 29 Shows the same image, but roughly shows the different branching points using coloured lines.

Near the top right of the image, the nodes "initially" split up into two branches due to non-deterministic choices during phase 5, the input generation. In phase 5, there is always the option to either skip generation or to do generation – for both the E and W generators.

These branches then remain fairly sequential as they execute the the remainder of phase 5, then phase 1 and phase 2. When either branch reaches phase 3, it starts splitting due to non-determinism again. **However**, note the point in the bottom middel of part of Figure 29 where the green and red branches join again. All arcs that join into that single node from both branches, are from **phase_3_manage_-core** component `cycle_finish_X` transitions. This join makes sense, because it happens due to the core cycle process in the same macro tick finishing.

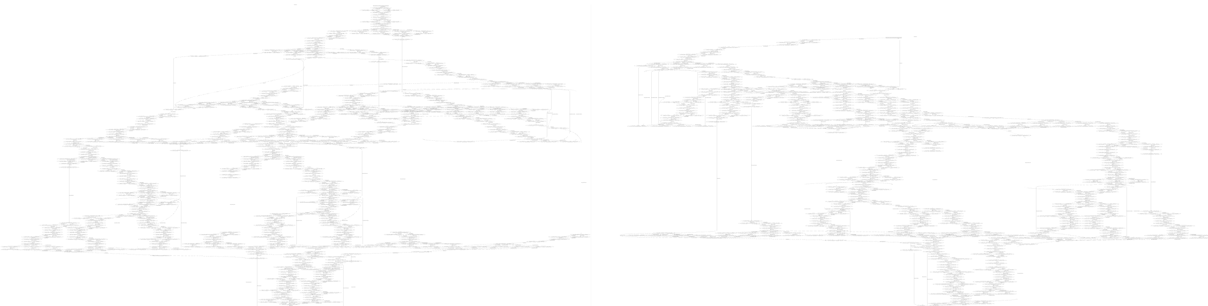

Figure 27: Coverability graphs (infinite and finite)
(graphs/coverability-infinite and graphs/coverability-finite)

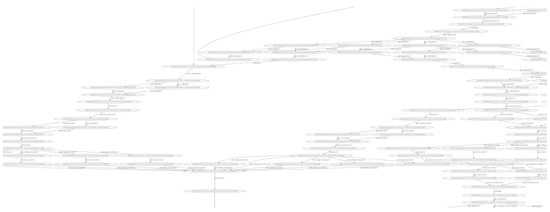

Figure 28: Coverability graph pattern inspection
(graphs/coverability-split-and-join)

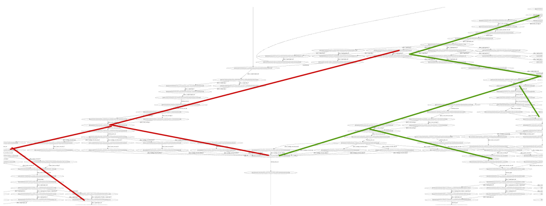

Figure 29: Coverability graph with patterns marked
(graphs/coverability-split-and-join-marked)

3.2. Invariant Analysis (B)

The following sections make use of `roundaboutModel-v3.tapn` : Section 3.2.1.

The following sections make use of `roundaboutModel-v3-invariant-change.tapn` : Section 3.2.2.

3.2.1. Roundabout P-Invariants

The following command was used to generate the P-invariants for the valid roundabout model (`roundaboutModel-v3.tapn`):

```
python3 assignment-files/RC.py roundaboutModel-v3.tapn roundaboutModel-v3-
garbage.dot coverability -p
```

The resulting P-invariants are displayed in Codeblock 1. Each of the four invariants consists of the same three distinct subexpressions.

- *clock* places subexpression' markings
- *phase_3_manage_core* compon subexpression' markings
- (*consider_X* + *finish_X*) place pair markings for *phase_1_outputs*, *phase_2_exit_core*, *phase_4_enter_core* and *phase subexpression_5_inputs*

Before we explain these subexpression, note that all places that are mentioned in the invariants, are strictly part of the "clock logic" places. That is, none of the in-road, out-road or core segment places are part of the invariants. This implies that we will only be able to make claims about the "clock logic" places.

Firstly, the *clock* component places subexpression is the following:

```
M(clock_cycle_core_end) + M(clock_cycle_core_start) +
M(clock_enter_core_end) + M(clock_enter_core_start) +
M(clock_exit_core_end) + M(clock_exit_core_start) +
M(clock_inputs_end) + M(clock_inputs_start) +
M(clock_outputs_end) + M(clock_outputs_start)
```

Note that this exact subexpression is part of all four invariants. It contains the markings for the start and end places for all five phase c In the context of the entire expression, it implies that at most a single token is present in all places of the *clock* component when excluding the `clock_ticks` place.omponents. This is expected, because the clock was designed to sequentially execute all f by passing around a single tokeni

Secondly, the *phase_3_manage_core* component places subexpression is the following:

```
M(allow_action_E) + M(allow_action_N) + M(allow_action_S) + M(allow_action_W)
```

This exact subexpression is also part of all four invariants. It contains the markings for the phase_3_move_core component's `allow_action_X` places. These four places were designed to facilitate sequentially iterating over all four phase_3_move_core component's `move_X` transitions. Thus, this subexpression is unsurprising, as they were designed to pass around a single token between the four of them.

Additionally, it is useful to know the fact that these four places only every have one token between the four of them. It ensures that when that one token is flushed to the phase 3 output place, no tokens remain in other places within the phase 3 component, ignoring the `core_X` shared places. This implies that only one token ever passes through the "clock logic" of phase 3 at a time.

Thirdly, the last subexpression is actually a set of subexpressions. It pertains to the `consider_X` and `finish_X` places of the components for phases 1, 2, 4 and 5. Very important to notice, is that the subnets of the components for phases 1, 2, 4 and 5 are extremely similar. All four of them essentially consist of:

1. a split structure from the phase's start place to the two `consider_X` places,
2. followed by a transition from each `consider_X` place to the corresponding `finish_X` place,
3. followed by a join structure from the two `finish_X` places to the phase's end place

Each of the invariants essentially contains a subexpression that includes one `consider_X` and `finish_X` pair for each of the phases 1, 2, 4 and 5:

```
M(phase_1_outputs.consider_A) + M(phase_1_outputs.finish_A) +
M(phase_2_exit_core.consider_B) + M(phase_2_exit_core.finish_B) +
M(phase_4_enter_core.consider_C) + M(phase_4_enter_core.finish_C) +
M(phase_5_inputs.consider_D) + M(phase_5_inputs.finish_D)
```

That such subexpressions are part of the invariants is not very surprising, given the following analysis of the components for phases 1, 2, 4 and 5. None of the `consider_X` places nor any of the `finish_X` places in any of the phase components are shared. The only source from where the `finish_X` places receive any token is directly from their corresponding `consider_X` place. So, by logical reasoning, **we may assume** that if a token arrives at the start place of any of the phase 1, 2, 4 or 5 components – the start of the split structure–, then that token surely flows through to that component's end place – the end of the join structure–, not adding or subtracting any tokens from the `consider_X` and `finish_X` places in doing so. In other words, the "clock token" is not consumed by these phase components, nor does it leave behind any after-effects inside the non-shared places of these components. Then, it is not surprising that each such branch of the split-then-join structure simply passes along a token from the split input to the join output, and can result in an invariant subexpression. We do make one assumption:

If a token arrives in the **phase_3_manage_core** start place
and the token subsequently enters **phase_3_move_core**,
then eventually the token will arrive at the **phase_3_manage_core** end place.
In other words, a token that enters the phase 3 components also exists.

What *is* surprising, is that not all variations of these `consider_X` plus `finish_X` place-pairs for each of the four involved phases resulted in an invariant. Because each of the phases has two place-pairs and there are four such phase components, we would expect $2c \cdot 4 = 8$ different invariants to be generated. Though it is possible that these four invariants are enough to derive the four missing ones.

Finally, we bring all three subexpressions together plus the assumption for phase 3. We notice that each complete invariant expression denotes a single, sequential path of places through the "clock logic". By "clock logic", we mean the places that are solely dedicated to implementing a functioning clock. Stronger still, each invariant describes a sequantial path through the clock, that when a token travels through it, a single macro time step is completed. Not only that, but each of those four paths always contains a single token. In other words, they describe macro-steps/runs of the clock without unwanted side-effects within the clock places.

3.2.2. Altering Roundabout P-Invariants

We can make a simple alteration to the petri net that both changes the invariants for the worse as well as violates the requirements.

To generate the P-invariants for the altered net (`roundaboutModel-v3-invariant-change.tapn`), we used the following command:

```
python3 assignment-files/RC.py roundaboutModel-v3-invariant-change.tapn
roundaboutModel-v3-garbage.dot coverability -p
```

We simply add a single directed arc with default weight 1 in the **clock** component. The arc goes from the `clock_tick` transition to the `clock_inputs_start` place. Firing the `clock_tick` transition will now add one additional token to the clock circuit. Over time, more and more tokens will circulate in the clock circuit at the same time. The resulting petri net has **no invariants at all** left.

The altered petri net violates the requirements, because the different phases can be executed in any order or even all at the same time. This violates the fact that cars on the in-road must give priority to the cars in the core that want to enter the in-road adjacent core segment, because this priority was a result of the clock's ordered execution of the phases. Cars may now perform multiple steps in the same macro step as well. Though the notion of a macro step went out the window when more than one token started circulating in the clock.

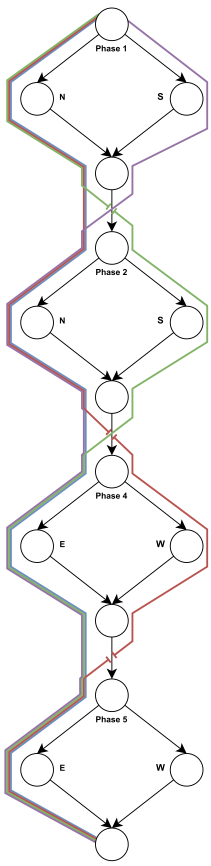

Figure 30: The subexpressions for the (consider_X, finish_X) pairs for phases 1, 2, 4 and 5 represented as paths through those phases (DIAGRAMS/INVARIANT-VISUALISATION/INVARIANT)

```
P-Invariants:
      M(allow_action_E)    +    M(allow_action_N)    +    M(allow_action_S)    +
M(allow_action_W)  +  M(clock_cycle_core_end)  +  M(clock_cycle_core_start)  +
M(clock_enter_core_end) + M(clock_enter_core_start) + M(clock_exit_core_end)
+  M(clock_exit_core_start)  +  M(clock_inputs_end)  +  M(clock_inputs_start)  +
M(clock_outputs_end) + M(clock_outputs_start) + M(phase_1_outputs.consider_S)
+      M(phase_1_outputs.finish_S)      +      M(phase_2_exit_core.consider_N)
+     M(phase_2_exit_core.finish_N)     +     M(phase_4_enter_core.consider_E)
+    M(phase_4_enter_core.finish_E)    +    M(phase_5_inputs.consider_E)    +
M(phase_5_inputs.finish_E) = 1

    M(allow_action_E)     +     M(allow_action_N)     +     M(allow_action_S)     +
M(allow_action_W)  +  M(clock_cycle_core_end)  +  M(clock_cycle_core_start)  +
M(clock_enter_core_end) + M(clock_enter_core_start) + M(clock_exit_core_end)
+  M(clock_exit_core_start)  +  M(clock_inputs_end)  +  M(clock_inputs_start)  +
M(clock_outputs_end) + M(clock_outputs_start) + M(phase_1_outputs.consider_N)
+      M(phase_1_outputs.finish_N)      +      M(phase_2_exit_core.consider_S)
+     M(phase_2_exit_core.finish_S)     +     M(phase_4_enter_core.consider_E)
+    M(phase_4_enter_core.finish_E)    +    M(phase_5_inputs.consider_E)    +
M(phase_5_inputs.finish_E) = 1

    M(allow_action_E)     +     M(allow_action_N)     +     M(allow_action_S)     +
M(allow_action_W)  +  M(clock_cycle_core_end)  +  M(clock_cycle_core_start)  +
M(clock_enter_core_end) + M(clock_enter_core_start) + M(clock_exit_core_end)
+  M(clock_exit_core_start)  +  M(clock_inputs_end)  +  M(clock_inputs_start)  +
M(clock_outputs_end) + M(clock_outputs_start) + M(phase_1_outputs.consider_N)
+      M(phase_1_outputs.finish_N)      +      M(phase_2_exit_core.consider_N)
+     M(phase_2_exit_core.finish_N)     +     M(phase_4_enter_core.consider_W)
+    M(phase_4_enter_core.finish_W)    +    M(phase_5_inputs.consider_E)    +
M(phase_5_inputs.finish_E) = 1

    M(allow_action_E)     +     M(allow_action_N)     +     M(allow_action_S)     +
M(allow_action_W)  +  M(clock_cycle_core_end)  +  M(clock_cycle_core_start)  +
M(clock_enter_core_end) + M(clock_enter_core_start) + M(clock_exit_core_end)
+  M(clock_exit_core_start)  +  M(clock_inputs_end)  +  M(clock_inputs_start)  +
M(clock_outputs_end) + M(clock_outputs_start) + M(phase_1_outputs.consider_N)
+      M(phase_1_outputs.finish_N)      +      M(phase_2_exit_core.consider_N)
+     M(phase_2_exit_core.finish_N)     +     M(phase_4_enter_core.consider_E)
+    M(phase_4_enter_core.finish_E)    +    M(phase_5_inputs.consider_E)    +
M(phase_5_inputs.finish_E) = 1
```

Codeblock 1: P-Invariants, four in total

4. Query Analysis (IV)

4.1. Boundedness (A)

The following sections make use of `roundaboutModel-v3.tapn` : Section 4.1.2., Section 4.1.3.

The following sections make use of `roundaboutModel-v3-reachability-remove-infty.tapn` : Section 4.1.4.

4.1.1. Definitions

We first introduce the notion of an **extra** token. The amount of extra tokens a net contains in a given state, is the amount of additional tokens in the net compared to the initial amount of tokens.
As an example, suppose we have a petri net that consists of only a split structure. By this we mean a net consisting of three places P_0, P_1 and P_2, one transition T_0 and the directed arcs (P_0, T_0), (T_0, P_1) and (T_0, P_2). The initial state of the petri net is $x_0 = [P_0 = 1, P_1 = 0, P_2 = 0]$, which represents a vector of marking values for the three places. If we were to then fire transition T_0, state $x_1 = [P_0 = 0, P_1 = 1, P_2 = 1]$ is reached. In initial state x_0 there was 1 token. In the state x_1 there are 2 tokens. Thus, x_1 has one extra token compared to x_0.

Next, we specify the used definition of boundedness. We call an entire petri net **k-bounded** iff. the maximal amount of tokens contained in the system in any of its states, is less than or equal to k. Said differently, a net is k-bounded if the the initial amount of tokens plus the amount of extra tokens in circulation is less than or equal to k, for all reachable states.
Consequently, we call a petri net **unbounded** in case the net is not k-bounded for any $k \neq \infty$.

4.1.2. Boundedness Check

We will verify the boundedness of the full roundabout petri net model, using TAPAAL's *Check Boundedness* feature. For this, we allow **20 extra tokens**.

As expected, our roundabout petri net is **unbounded**. The generators can accumulate an infinite amount of tokens in the in-road places as long as the out-road places non-deterministically choose to never let any car exit the roundabout completely. This is because if no cars every leave the out-roads, then they block other cars from having the possibility to exit (queueing), all the while new cars enter from the in-roads. The same is true for the clock tick place – it will accumulate tokens ad infinum.

4.1.3. Boundedness Query

From the discussion of P-invariants (see Section 3.2.), we can derive a set of places that is assured to be 1-bounded. Notice that each invariant expression equals 1. And **invariants, by definition**, hold for every reachable system state. So, **we may assume** that every place mentioned in any of the invariants is 1-bounded. Because, if any of those states were to ever contain more than one token, then the related invariants would be invalidated.

The places of which the **boundedness remains unknown**, are as follows, ordered by the component that the places logically belong to:

- **clock**: `clock_ticks`
- **phase_1_outputs**: `output_N` , `output_S`
- **phase_3_manage_core**: `cycle_progress`
- **phase_3_move_core**: `core_N` , `core_E` , `core_S` , `core_W`
- **phase_5_inputs**: `input_W` , `input_E`

We now determine the boundedness of each of these places, so that an overall boundedness conclusion can be reached. For every query, we will **allow 20 extra tokens** for the verification, unless stated otherwise.

For **phase_1_outputs**, we utilize a positive match: for all reachable markings (AG), `M(output_X) ≤ 1` must hold for $X \in \{N, S\}$. This **CTL** query is called 'output-boundedness-check':

```
AG (output_N ≤ 1 and output_S ≤ 1)
```

This query is satisfied in our net. Due to the use of `AG` each of the `output_X` places is at the very least 1-bounded.

For **phase_3_manage_core**, we utilize a positive match: for all reachable markings (AG), `M(cycle_progress) ≤ 4` must hold. This **CTL** query is called 'cycle-progress-boundedness-check':

```
AG (cycle_progress ≤ 4)
```

This query is satisfied in our net. Due to the use of `AG` the `cycle_progress` place is at the very least 4-bounded.

For **phase_3_move_core**, we utilize a positive match: for all reachable markings (AG), `M(core_X) ≤ 1` must hold for $X \in \{N, E, S, W\}$. This **CTL** query is called 'core-boundedness-check':

```
AG (core_N ≤ 1 and core_E ≤ 1 and core_S ≤ 1 and core_W ≤ 1)
```

This query is satisfied in our net. Due to the use of `AG` each of the `core_X` places is at the very least 1-bounded.

Checking the boundedness of the remaining places **proved tricky**. We encountered a major problem: How do we express that the number of markings of a place monotonically increases? If a query could express this, then we could guarantee unboundedness. We tried to make use of the LTL *next* operator to no avail. Using it, we tried comparing the current marking of a place to its marking in the next state, but we could not get this to work as we wanted it to. The problem was that we could not obtain the marking value of a place both in the current state and the next state within the same equation. One such failed attempt was created for a toy example net, see **Figure 31** and **Codeblock 3**.

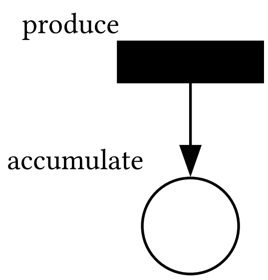

Figure 31: Toy example for unbounded net

```
E (G (F (X accumulate > accumulate)))
```

Codeblock 3: There exists a state such that
always eventually the accumulate marking strictly increases

So we decided to take a different approach. Instead of making claims about the markings of places, we make claims about the liveness of transitions instead.

For **clock**, we know that the only way to change the tokens count inside the `clock_ticks` place, is by firing the `clock_tick` transition. It is visually obvious that the token count in the `clock_ticks` place can only increase. So, for this place to be unbounded, the `clock_tick` transition must be fired infinitely often. We utilize a positive match: for all computations (A), always (G) eventually (F) the transition `clock_tick` should be enabled. In other words, due to the sequential clock behavior discussed in **Section 3.2.**, that transition must fire infinitely often. This **LTL** query is called 'clock-unboundedness-check':

```
A (G (F clock.clock_tick))
```

This query is satisfied in our net. Regardless of the use of `A` the `clock_ticks` place is unbounded because at least one trace satisfying the query was found.

For **phase_5_inputs**, we know that the there are two ways to change the tokens count inside the `input_E` and `input_W` places. The first is incrementing the count by firing the phase 5 component's `generate_E` and `generate_W` transitions respectively. The second is decrementing the count by firing the phase 4 component's `enter_E` and `enter_W` respectively. For unboundedness of either of the inputs to occur, the `enter_E` or `enter_W` transitions should not fire, while the `generate_E` or `generate_W` transitions fire infinitely often. Said differently, the roundabout should be congested and not let cars in the out-roads leave, while new cars continually arrive on the in-roads. As long as one trace exists where this is possible, the inputs are unbounded.
We utilize a positive match: there exists a computation (E), so that eventually (F) all `output_X` places and `core_Y` places occupied and that both `generate_W` and `generate_E` always (G) eventually (F) fire. In other words, due to the sequential clock behavior discussed in **Section 3.2.**, the entire core and out-roads are congested and consequently the two `generate_Z` transitions can separately fire infinitely often. This **LTL** query is called 'inputs-unboundedness-check':

```
E (
  F (
    output_N = 1 and output_S = 1 and
    core_N = 1 and core_E = 1 and core_S = 1 and core_W = 1 and

    (G (F phase_5_inputs.generate_W)) and
    (G (F phase_5_inputs.generate_E))
  )
)
```

This query is satisfied in our net. Due to the use of `E` the `input_E` and `input_W` places are unbounded because at least one trace satisfying the query was found.

We close with a **general conclusion** on the boundedness of the net. It was expected that only the in-roads and the clock counter would be unbounded places. All other places were either clock logic — designed to sequentially "pass around" a single token – or the roundabout's core and output logic – designed to have a car capacity of 1. Thus, it was expected that the net in general would be unbounded, as at the very least the clock counter had to be unbounded by definition and design.

4.1.4. k-Boundedness Check

Note that a k-boundedness check was not required by the assignment. This section is to be used as a reference by other sections, to construe arguments based on k-boundedness.

Our boundedness is *infinite* in the regular roundabout model, as we can infinitely keep generating clock ticks and inputs. Note that by using the diagram of the previous reachability solution, we can ensure that the model will not generate more than 11 **extra tokens** in the entire system at maximum. This means that the model modified for reachability is 12-bounded, because the initial state contains only a singular token — the one within the clock.

The situation where 11 extra tokens can exist at the same time, is given as:

- +2 tokens – one on each of the inputs (the input places have been constrained to a max. 1 token capacity each)
- +2 tokens – one on each of the outputs (which are already limited to one to a max. 1 token capacity each)
- +3 tokens – spread across all core segments arbitrarily, with every core segment consequently having at most one token
    - +4 tokens due to all core segments being filled is impossible here, see the `cycle_stall` explanation below
- +4 tokens – which keep track of the cycle progress (note that the `allow_action_X` state receives the token from `clock_cycle_core_start`, so there isn't an additional token that will be generated)

Additionally, we need to observe that:

1. the clock ticks are not kept (as per the previous, reachability solution)
2. if every core segment contains a token, then we cannot execute the core cycle process, and we instead have a `cycle_stall` action, which bypasses the more finegrained cycle movement process — in this case, we would have 8 extra tokens at maximum (4 from the I/O and 4 from the core segments and *none* from the cycle progress)

4.2. Deadlock (B)

Deadlock in the context of petri net simulation, occurs when a state is reached where **none of the outgoing transitions are enabled**. This means that no other state can be reached from any given deadlock state.

The following sections make use of `roundaboutModel-v3.tapn`: sections 4.2.1., 4.2.2., 4.2.3.

4.2.1. Deadlock Check

TAPAAL provides us with an easy way of testing deadlock: the deadlock keyword. We make use of a negative match: is there a reachable marking (EF) so that there is deadlock. This **CTL** query is called 'deadlock-check':

```
EF (deadlock)
```

This query is **not satisfied** in our net. This implies that no deadlock was found. For this, we allowed **20 extra tokens**. Since Section 4.1.4. shows that the more constrained version of our net is 12-bounded, 20 extra tokens should suffice for a proper deadlock analysis.

4.2.2. Deadlock in Graphs

In both the reachability graph as well as the coverability gaph, deadlock takes the form of nodes that **do not have any** outgoing arcs. They only have ingoing arcs. This way, the simulation may bring the system into the deadlock state, but it is not able to leave that deadlock state again.

4.2.3. Absence of Deadlocks

We refer to the invariant analysis in Section 3.2.. More specifically, to the fact that the clock logic consists of several sequential paths of places, such that each path always contains exactly one token. Note that the initial system state consists of a single token at the start of a macro step in the clock component. From these observations, we infer that the clock logic does not leave any after effects in any of the places that strictly conduct the clock's token.

In simpler terms, the clock only passes along its single initial token, which is split in two and then merged again into one token in phases 1, 2, 4 and 5, during every single macro step. The invariants guarantee that the clock does duplicate its initial token.

The **clock component** itself simply passes tokens from phase's end place to the start place of the next. No deadlock may occur here.

The **phase_1_outputs**, **phase_2_exit_core** and **phase_5_inputs** components are all similar, in that they always have the option to simply fire the `skip_X` transition, so no deadlock can occur in any of the three components.

The **phase_4_enter_core** component is slightly more complex. We note that all four possible state combinations for the places `input_X` and `core_X` are accounted for in Table 9. From this, we conclude that there is always a path through this component, so no deadlock can occur in this component.

	M(input_X) = 0	**M(input_X) ≥ 1**
M(core_X) = 0	skip	enter
M(core_X) ≥ 1	skip	stall

The **phase_3_manage_core** component provides two general ways for a clock token to pass through:

1. The simplest is the combination of `cycle_stall` and `cycle_skip`.
    - If all four core segments are full, then we assume that traffic in the core is congested and cannot cycle around the roundabout, so `cycle_stall` offers a way out. **This measure avoids deadlock in case of congestion**.
    - Firing `cycle_skip` just offers a shorter way to skip the core cycling process of this macro step. This guard against cases of deadlock.
2. If any `allow_action_X` transition is fired, then the core cycle process begins.
    - The **only way for deadlock to occur** here, is for the `cycle_progress` place to never accumulate 4 tokens. So if it always does accumulate 4 tokens, then no deadlock can occur. To disprove the possibility of deadlock, we must consider the **phase_3_manage_core** component.

The **phase_3_move_core** component functions as follows:

1. A token arrives in **exactly one** of the `allow_action_X` places. That only one of the four places contains a token is guaranteed by the invariants discussed in Section 3.2.
2. This token iterates through the `allow_actions_X` places in a sequential, circular order. We prove that no deadlock is possible for any of these four places using two examples.
   - Consider Table 10. Note that only the places `core_E` and `core_N` influence the outgoing transitions of `allow_action_E`, apart from `cycle_progress`. Unless `cycleprogress = 4`, there cannot occur deadlock for `allow_action_E`. An analogous example can be constructed for `allow_action_W`.
   - Consider Table 11. Note that only the place `cycle_progress` influences the outgoing transitions of `allow_action_S`. Unless `cycleprogress = 4`, there cannot occur deadlock for `allow_action_S`. An analogous example can be constructed for `allow_action_N`.
3. Note that every `cycle_X` and each `stall_X` increments `cycle_progress` by one.
4. Because `cycle_progress = 4`, every single transition within the **phase_3_move_core** component is blocked.

The previous description of the workings of **phase_3_move_core** should convince you that deadlock does **not** occur in this component. On top of that, `cycle_progress = 4` always occurs, because the token in the `allow_action_X` loop is able to keep looping until `cycle_progress` reaches 4 and blocks it from looping any more. Though it might be tempting to consider the blocking of all transitions in **phase_3_move_core** as deadlock, we now argue that the system has not actually reached deadlock.

Remember that **phase_3_manage_core** could only reach deadlock if `cycle_progress = 4` never occurs after a transition `action_start_X` passes the clock token to the corresponding shared place `allow_action_X`. However, we just proved that if a token arrives in any `allow_action_X` place, then `cycle_progress = 4` is guaranteed to occur. Thus, it is impossible for the only option for deadlock in **phase_3_manage_core** to occur. So there is **no deadlock** in this component.

	M(core_N) = 0	**M(core_N) = 1**
M(core_E) = 0	cycle_E	cycle_E
M(core_E) = 1	move_E **THEN** cycle_E	stall_E

Table 10: Transition sequence for E-to-N core cycle state combinations

	M(core_E) = 0	**M(core_E) = 1**
M(core_S) = 0	cycle_S	cycle_S
M(core_S) = 1	cycle_S **OR** (move_S **THEN** cycle_S)	cycle_S

Table 11: Transition sequence for S-to-E core cycle state combinations

In conclusion, the **system is free of deadlock** because the clock and all phase components are free of deadlock and because a single macro step consists of a sequential run of all phase components via the clock component.

4.3. Liveness (C)

Liveness refers to the number of times transitions are fired.

The following sections make use of `roundaboutModel-v3.tapn` : Section 4.3.2.

The following sections make use of `roundaboutModel-v3-liveness-change.tapn` : Section 4.3.3.

You may assume all queries in this section make use of **20 extra tokens**, unless specified otherwise.

4.3.1. Definitions

Liveliness Definition

Given initial state x_0, a transition in a Petri net is:

- **L_0-live** (dead): if the transition can never fire.
- **L_1-live:** if there is some firing sequence from x_0 such that the transition can fire at least once.
- **L_2-live:** if the transition can fire at least k times for some given positive integer k.
- **L_3-live:** if there exists some infinite firing sequence in which the transition appears infinitely often.
- **L_4-live:** if the transition is L_1-live for every possible state reached from x_0.

4.3.2. Liveness Query

We refer to the partial liveness analysis in Section 4.1.3.. There, we used liveness to demonstrate unboundedness. In that section, we already constructed two queries that determined liveness.

The **first query** showed that the `clock_tick` transition is **L_4-live**. It stated that for every computation the `clock_tick` transition would always eventually be enabled. This is synonymous with it firing infinitely often in every trace.

The **second query** showed that both the `generate_E` and the `generate_W` transitions are **L_3-live**. It stated that there exists some computation, where the core and outputs are congested, such that both the `generate_E` and the `generate_W` transitions would always eventually be enabled. This is synonymous with some trace existing where they fire infinitely often.

We consider **one last transition**: `cycle_stall` in the **phase_3_manage_core** component. We check a selection of liveness levels, from lower to higher.
The **CTL** query called 'cycle-stall-liveness-L1' checks **L_1-liveness**: there exists a state/computation so that eventually the transition is enabled

```
EF (phase_3_manage_core.cycle_stall)
```

The **LTL** query called 'cycle-stall-liveness-L_3' checks **L_3-liveness**: there exists a state/computation so that always eventually the transition is enabled

```
E (G (F (phase_3_manage_core.cycle_stall)))
```

The **LTL** query called 'cycle-stall-liveness-L_4' checks **L_4-liveness**: for every state/computation always eventually the transition is enabled

```
A (G (F (phase_3_manage_core.cycle_stall)))
```

We can conclude that `cycle_stall` is **L_4-live**.

4.3.3. Altering Liveness

We can make a simple change to impact the liveness of most if not all transitions in the entire petri net: we **limit the amount of macro steps** that may be performed. To do this, in the clock component we simply add an inhibitor arc from the `clock_ticks` place to the `clock tick` transition.

In the **first macro step**, every phase except for the inputs generation phase is essentially a noop, because the input generation step is performed sequentially last. After this, the `clock_tick` transition is fired for the first and last time – the `clock_ticks` place marking becomes 1.

The **second macro step** will be able to actually perform the inputs phase and the enter core phase. The other phases are essentially noops, because there are not yet any cars in the core segment places, nor in the out-road places.

We will now consider the changes in liveness for the three previously described cases. We remind you that for the following discussion, all queries are run in `roundaboutModel-v3-liveness-change.tapn`.

First is the `clock_tick` transition. It is entirely expected that it is now **L_1-live** instead of **L_4-live**. The alteration to the net essentially blocks the `clock_ticks` transition after it fires once. Running the original query, called 'clock-unboundedness-check', in the new net, reveals that it is not satisfied anymore. We can prove that `clock_tick` is **L_1-live**, but not **L_2-live** using a single query. The following **LTL** query is called 'clock-liveness-not-L_2'. It states that there exists some state/computation such that eventually the `clock_tick` transition is enabled (L1), but that after that `clock_tick` does not become enabled ever again (not L_2 for $k \geq 2$).

```
E (F (clock.clock_tick and (X !(F clock.clock_tick))))
```

This query is true in the modified net, and implies **L_1-liveness**.

Second, the `generate_E` and `generate_W` transitions are both **L_2-live**. Running the original query, called 'inputs-unboundedness-check', in the new net, reveals that it is not satisfied anymore. This means that neither `generate_E` nor `generate_W` is **L_3-live** anymore. The new **LTL** queries are 'inputs-E-liveness-L_2' and 'inputs-W-liveness-L_2' respectively. They each require that some trace exists, so that the corresponding `generate_X` transition eventually becomes enabled twice.

```
E (F (phase_5_inputs.generate_E and (X (F phase_5_inputs.generate_E))))
```

This query is true in the modified net, and implies **L_2-liveness** for $k \geq 2$ for transition `generate_E`.

```
E (F (phase_5_inputs.generate_W and (X (F phase_5_inputs.generate_W))))
```

This query is true in the modified net, and implies **L_2-liveness** for $k \geq 2$ for transition `generate_W`.

Third, the `cycle_stall` transition becomes **L_0-live**. Intuitively, a minimum amount of macro steps are needed for the roundabout core segments tp become occupied. The `cycle_stall` transition only becomes enabled once all four core segments are occupied. Only two full macro steps – the second macro step deadlocks just before incrementing the counter at te end – are not enough for the `cycle_stall` transition to ever become enabled. We can check this easily with the **CTL** query named 'cycle-stall-liveness-L0'. This query states that for every reachable state/marking, it is globally true that `cycle_stall` is not enabled:

```
AG !(phase_3_manage_core.cycle_stall)
```

This query is true in the modified net, and implies **L_0-liveness** for transition `cycle_stall`.

4.4. Fairness (D)

We must prove that our net implements “true” fairness, meaning that every clock event can only happen once for every (macro) clock tick.

In Section 3.2.1. we argued that given an initial state where only the **clock** component is initialized to its start state, then that singular initial token is guaranteed to run through the entire clock. It does this in the order of the phases, from phase 1 to phase 5.

In Section 4.2.3. we argued that given an initial state where only the **clock** component is initialized to its start state, then:

> *“In conclusion, the **system is free of deadlock** because the clock and all phase components are free of deadlock and because a single macro step consists of a sequential run of all phase components via the clock component.”*

It does this in the order of the phases, from phase 1 to phase 5.

So, given our initial state, we can claim that the initial clock token will infinitely loop within the clock. Every loop of the token inside the clock will consist of sequentially passing through the phase components in the following order:

1. **phase_1_outputs**
2. **phase_2_exit_core**
3. **phase_3_manage_core**
4. **phase_3_move_core**
5. **phase_4_enter_core**
6. **phase_5_inputs**

The reasoning for this order is as follows. A car is able to go from input to output in a single macro tick iff. the clock first generates a car, then lets that car enter the roundabout core, then lets it cycle around the core, then lets it exit the core to an out-road and finally lets the car leave entirely. To prevent a single car from making more than one move per macro tick, we must satisfy two requirements:

- the order the phases are executed by the clock **must** be as ordered above
- the **phase_3_move_core** phase must take care to only move any car at most one step per macro step. This can be achieved by firing all three sequential transitions of the four `move_X` transitions in the **phase_3_move_core** components

4.5. Safety (E)

The definiton of safety we use is the following: The roundabout is safe if it is impossible for two vehicles to collide in a core segment.

The following sections make use of `roundaboutModel-v3.tapn` : Section 4.5.1.

The following sections make use of `roundaboutModel-v3-safety-change.tapn` : Section 4.5.2.

You may assume all queries in this section make use of **20 extra tokens**, unless specified otherwise.

4.5.1. Safety Analysis

In our roundabout implementation, it is impossible for two vehicles to crash. We earlier proved that that all core segment places are 1-bounded, in Section 4.1.3.. The relevant query is found in Codeblock 2. Because **every `core_X` place is 1-bounded, it is impossible for a crash to occur**.

A more practical explanation looks at which transitions lead into the `core_X` shared places.

1. In the **phase_3_move_core** component, every `core_X` place has one `move_Y` transition leading into it.
   → **But** `core_X` inhibits `move_Y`
2. In the **phase_4_enter_core** component, `core_W` has an incoming `enter_W` transition and `core_E` has an incoming `enter_E` transition.
   → **But** `core_W` and `core_E` inhibit `enter_W` and `enter_E` respectively

All other arcs connected to any `core_X` place are either double sided/headed – meaning there are *both* an incoming arc as well as an outgoing arc – OR they are outgoing. Thus, there cannot be any collisions on the core.

The out-roads similarly inhibit all of their respective, incoming transitions, which prevents collisions in `output_N` and `output_S` as well.

4.5.2. Altering Safety

It is **impossible to find an acceptable initial marking** that causes a crash to occur, unless the petri net's arcs are first modified (if some are deleted). Except if two tokens were to be put on the same core segment initially, but that is a rather pointless example. To understand why no such initial marking can be found, we refer to the practical explanation in the previous section: Section 4.5.1. Essentially, every `core_X` and `output_Y` place inhibits als of its incoming transitions. This unconditionally prevents collisions from occurring.

We can again make a **small adjustment to the original petri net**, `roundaboutModel-v3.tapn` , which results in `roundaboutModel-v3-safety-change.tapn` . In the **phase_4_enter_core** component we can simply remove either of the inhibitor arcs going from one `core_X` place to the corresponding `enter_X` transition. We shall remove the inhibitor arc from the place `core_E` to transition `enter_E` . This will allow a car on the east in-road to enter the eastern core even though there is no room to enter.

The corresponding trace is called `roundaboutModel-v3-safety-change.trc` . Note that at the very last step of the trace, the `core_E` place in the **phase_3_move_core** component **contains 2 tokens**.

A. Appendix

A.1. Old clocked roundabout version

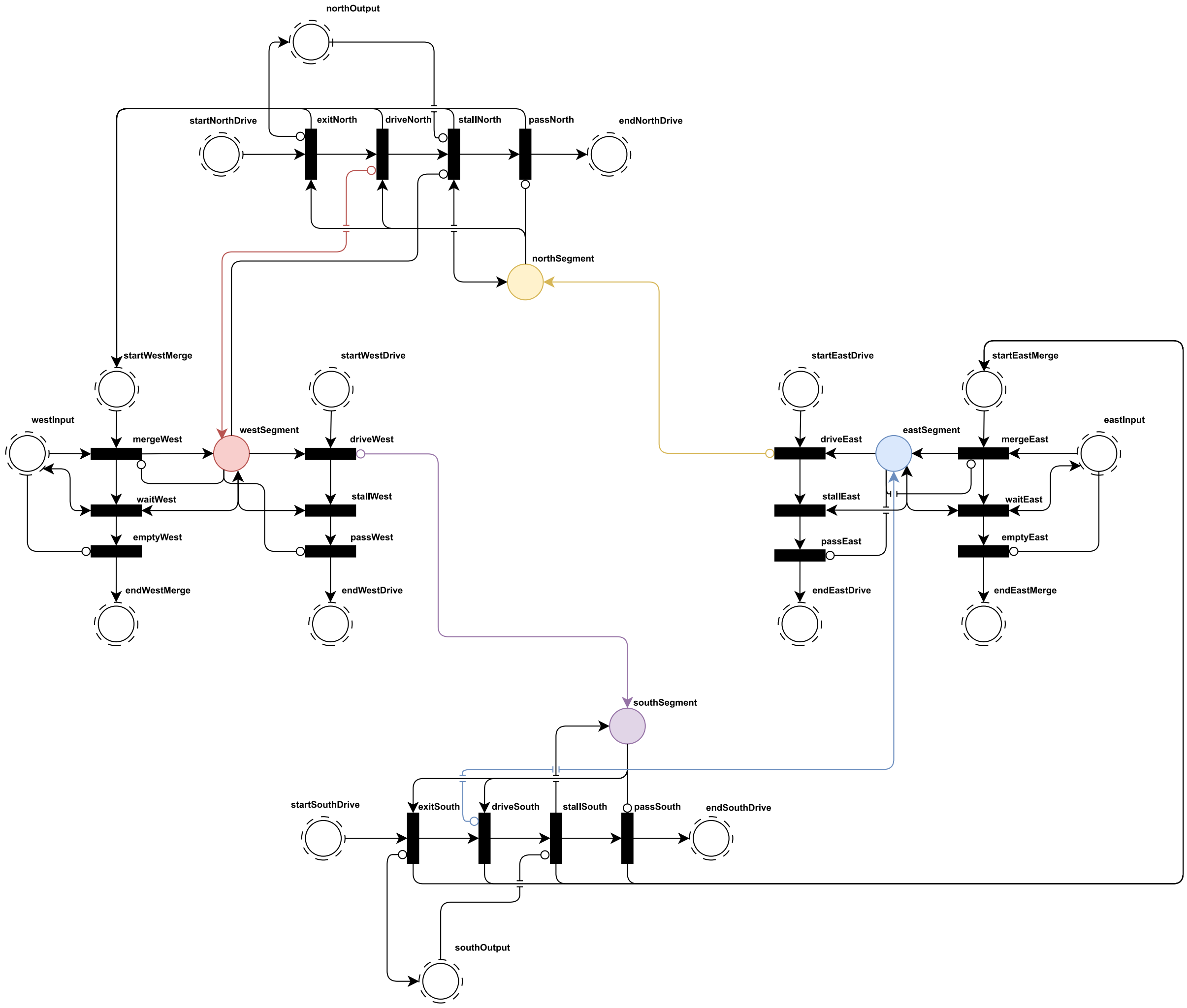

Figure 32: Roundabout with unordered actions