# MoSIS Assignment 4

## Team


```
Name       :      Danielle Jongstra
Student ID :      20172260
e-mail     :      Danielle.jongstra@student.uantwerpen.be

Name       :      Joanna Kisaakye
Student ID :      20205490
e-mail     :      joanna.kisaakye@student.uantwerpen.be
```


Time spent on this assignment (in hours): 36

## The assignment

## Petri Net

The Petri net is modelled in `Assembly.xml`. The entire net is also shown in fig. 1. There is a simplified Petri net in fig. 2, which does not have the time requirements modeled. In this simplified form the general flow of tokens is easier to see, and therefore we'll describe this Petri net first.

### Petri Net Base

#### Shapes

In the Petri net, we start on the left side. We can see the transitions CubeArrives and CylinderArrives that produce a new shape of either Cube or Cylinder respectively and add them to the queue if the queue is not completely filled yet. The amount of places left in the queue is the amount of tokens in AssemblerQ.

#### Assembly

As soon as there are at least 2 Cubes and 1 Cylinder in the queue, the transition Assembly becomes enabled, under the extra condition that the assembler is not used already and that there are no remakes queued. The remakes are discussed a little further in the Petri net description. If the transition is taken, the token denoting that the assembler is available is taken away so that another object is not able to enter the assembler.

During assembly, there is a token in Assembling place denoting that the assembler is in use.

After the assembly is done, the transition ToInsp is used. This transition transfers a token back to AssemblyAvailable, as the assembler is no longer in use when this transition is taken. The object is then in a queue to enter the inspector. The amount of objects in this queue is kept the same way the queue of the assembler was kept.

#### Inspection

Transition Insp is taken to move the object from the queue to the inspector. Availability of the inspector is checked the same way it is in the assembler. After this, the inspector is vacated using InspLeave transition and the object will be in the flagging zone: Inspected. The object can be flagged as either thrashed, accept or remake. If it is flagged as thrashed or accept, there is a token added to ThrashedAmount and AcceptedAmount respectively. These states are counters used for keeping track of the performance.

#### Remake

If the object is flagged as a remake, there is a token added to the counter RemakeAmount which is also used for tracking the performance. Another thing that is done when an object is flagged as remake is that is takes a token from RemakeNotQueued. This place has 10 tokens if there are no remakes waiting for the assembler. For each remake waiting a token is taken away. There is also a token in RemakeQ, which is the object to remake being in the queue. The remake can enter the assembler by taking the transition AssemblyRemake. After this transition is taken, the object will follow the same route as described before. If there is a remake, the assembler can not make a new object from 2 Cubes and a Cylinder. The transition that would transfer a new object to the assembler requires 10 tokens in RemakeNotQueued. Because a token is taken away every time a new remake is flagged, the transitions Assembly and AssemblyRemake are mutually exclusive.

### Petri Net With Time

The complexity in fig. 1 comes from adding the time and the necessary checks that attributed to that. Most of the places related to the time are on the bottom half of the Petri net. The time works by having a token available in Time_NextAvailable place which starts the next time step. DoNextStep then distributes a token to each of the 3 main processes (shapes, assembler and inspector) to denote that they can still do something during that time slice. If a process is unable to do something, chooses to skip acting during the time slice or does an action, the token denoting that an action is available is transferred to a place denoting the process has been executed during the time slice.

Extra logic is added to check that a process can only skip processing if the process before it already attempted to process. This way, a process that was not able to become active without another process already having executed is still able to execute after this process did their step.

The assembler and inspector have some new places to check if they are able to execute during the time slice. They can only skip if these places denote that they are not able to execute. As soon as all the processes have either skipped or executed, the transition NextStepAvailable becomes available. This takes the tokens from the executed places and transfers one to the Time_NextAvailable place. This means the time logic can start from the beginning.

Figure 1: The complete Petri net

Figure 2: A simplified Petri net, not keeping time into account

## Simulation 1

Simulation 1 is the case of "The Assembler finished an assembly at the exact same timestep the Inspector marks an assembly for reassembly". The simulation is shown in fig. 3 till fig. 7.

1. In the first step, it can be seen that both the inspector and the assembler can finish.

   Figure 3: Step 1 in simulating case 1
2. Step 2 shows that within the same time step the object leaves the inspector, it can be flagged as a remake.

   Figure 4: Step 2 in simulating case 1
3. Step 3 shows that an object leaves the assembler. The other object is in the remake queue.

   Figure 5: Step 3 in simulating case 1
4. Step 4 shows that when the assembly object is moved to the inspector queue, the transition for the remake object to the assembly procedure becomes enabled.

   Figure 6: Step 4 in simulating case 1
5. Step 5 shows that after entering the assemble procedure, the object can move into the actual assembly. It is possible that the object is in the assembler for the first time step during the same time slice. This however depends on the leaving object. If the leaving object used the assembler during the time slice, the new object will only gain being in the assembler for 1 time unit in the next time slice. Otherwise it is possible for the new object to be in the assembler for one time unit during the current time slice.

   Figure 7: Step 5 in simulating case 1

## Simulation 2

Simulation 2 is the case of "There is an assembly waiting for reassembly (i.e., it is to be fixed) and at that exact time the third component of an assembly arrives". This can be seen in fig. 8 and fig. 9.

1. The first step its blue circles show that there is an assembly for remake in the remake queue and that it is not yet possible to assemble a new object. The pink circle shows that in this time step, the last needed shape can arrive.

   Figure 8: Step 1 in simulating case 2
2. The pink circle in the second step shows that the last shape has arrived, but that the transition to the assembler for the new object is not enabled. The blue circle shows that the transition for the remake is enabled.

   Figure 9: Step 2 in simulating case 2

## Reachability

The reachability graph for arriving cubes and cylinders is in fig. 10. This graph does not cover the cubes and cylinders leaving the queue. It can be seen in the graph that a shape arrives till the queue is full, which are the red states. Every state can be reached by multiple paths, except for the states that denote the queue being filled by either just cubes or just cylinders.

Figure 10: The reachability graph of the arriving of cylinders an cubes

The full Petri net would produce an infinite reachability graph, as it is possible that an infinite amount of objects are accepted/thrashed/remade in an infinite amount of time steps. All those are kept track of in the Petri net. Even if the Petri net has a limited stock, the reachability graph would still be infinite. This is because the time will go on even if there is nothing that can be done, which would be the case if the stock was empty. Not taking the time into account, the reachability graph would still be infinite. The inspector could keep flagging the same object as a remake over and over again. This means that the remake counter could get infinitely big.

## Invariants

- **M(AssemAble) + M(AssemblyActive) + M(AssemblyAvailable) = 1**: This invariant was not expected, but does make sense. AssemblyAvailable denotes that no object has tried to enter the assemble procedure. When an object does try, the single token is taken from AssemblyAvailable and moved to AssemAble. This means that the object is trying to get in the assembler. If this happens, the token is taken from AssemAble and moved to AssemblyActive. When the assembler is done, the token is taken from AssemblyActive and moved to AssemblyAvailable. This means that there is never a token in all three places at the same time.
- **M(AssemAble) + M(AssemblyActive) + M(AssemUnable) = 1**: This was definitely an unexpected invariant. The expected invariant consisted of AssemAble and AssemUnable being 1, as the assemble procedure is either available or not. That AssemblyActive is included in this, is because the token does not move immediately from AssemUnable to AssemAble. It first goes to AssemblyActive to be transferred back to AssemAble when the assembler frees up.
- **M(AssemblerQ) + M(CubesInQ) + M(CylindersInQ) = 10**: This invariant was expected. That the available assembly queue and the amount of the cubes in the queue and the amount of cylinders queued are a total of 10.
- **M(InspAble) + M(InspectinAvailable) + M(Inspecting) = 1**: the same reasoning goes for this as for the the equal assembly invariant (first on the list).
- **M(InspQ) + M(InspQAvailable) = 10**: This invariant was expected, as the token moves from the available queue to the actual queue every time an object arrives.
- **M(InspAble) + M(Inspecting) + M(InspUnable) = 1**: For this invariant, the same goes as for the related assembler invariant (bullet-point 2)
- **M(RemakeNotQueued) + M(RemakeQ) = 10**: same goes as for the inspector queue.
- **M(Time_Assembly) + M(Time_AssemblyExecuted) + M(Time_NextAvailable) = 1**: This invariant was expected, because the Time_Assembly denotes that the assembler can still do a step in that time slice, while Time_AssemblyExecuted has a token when the assembler has executed in that time slice. Time_NextAvailable contains a token if all processes had a token which can be taken away, so Time_AssemblyExecuted does not have the token anymore as soon as Time_NextAvailable has one. It would be bad if there were multiple tokens in these three places combined.
- **M(Time_InspectingExecuted) + M(Time_Inspection) + M(Time_NextAvailable) = 1**: same goes as for the Time_Assembly equation above
- **M(Time_NextAvailable) + M(Time_Shape) + M(Time_ShapeExecuted) = 1**: same goes as for the Time_Assembly equation above
- **M(AssemblerQ) + M(TotalInQAssem) = 10**: same goes as for the inspector queue.

The analysis of these invariants is in `invariants.html`. The invariants do not change by adding a stock value. The invariants can be changed by adding a single place. In the example that generated `invariantChange.html` a place InvarChange was added. It has 200 tokens at the start of a simulation. Every time a cylinder arrives, a token is taken, and every time a cylinder an two cubes are moved to the assembler, a token is added. This means that the sum of tokens in the cylinder queue and the InvarChange places is always 200. This adds the last invariant on the list, **M(CylindersInQ) + M(InvarChanger) = 200**.

## Boundedness

A Petri net is (k-)bounded if the amount of tokens in each place never exceeds a certain value of k. This would also mean that the reachability graph would have to be finite, which is not the case for our graph. Therefore, we can conclude that our Petri net is not bounded. In LoLA, the option to use for a boundedness check is '--formula="AG p < oo" with p being the place of interest. To get the boundedness for the whole net, you have iterate over every place. If the place was bounded, the k-boundedness could be discovered by using the option --formula="AG p <= k". The boundedness check can be found in script `LoLARunner_Boundedness_AssemblySimStock.py`. This uses the stock version of the net, because if that version is not bounded, then the non-stock version is certainly not bounded. Other used options are --search=cover and --markinglimit=1000, this is to prevent the analysis running forever in the case that the net is not bounded. When using these options, we get a segmentation error. Else, the analysis runs for a long time and the result of this can be seen in `output_boundedness_asimstock.json`. As LoLA seems to run out of space, we should try to reduce the state space to analyze. This can be done by using the --cycle option, which stores only a subset of the markings. However, that does not do anything for our petri net.

## Deadlock

For discovering a deadlock using LoLA, the option --formula="EF DEADLOCK" can be used. Just as with the boundedness, there is a segmentation fault. However, a deadlock means that no transition can fire. Because of how the time has been designed to work in our Petri net, the next time step can be achieved by every process either running or skipping, a deadlock should not be possible. The script for the deadlock is in `LoLARunner_Deadlock_AssemblySimStock.py` and the output after a long run is in `output_deadlock_asimstock.json`. Using the --search='cover' flag, the analysis returns a segmentation fault. Without this flag, the analysis runs for a long time.

## Queue overflow

The queue overflow can be checked by seeing if the states that queue objects for the assembler/inspector have a boundedness of 10. This can be checked by using the option --formula="AG TotalInQAssem + RemakeQ < 11" in LoLA, with TotalInQAssem the amount of cubes and cylinders, and RemakeQ the amount of objects queued for remake for the assembler. LoLA returns that these states are not 10-bounded, so a queue overflow is possible for the assembler. For this analysis the --search='cover' flag was used. This can be seen in the execution log, `AssemblySimStock_Queue_Overflow.log` generated by this script, `LoLARunner_Queue_Overflow_AssemblySimStock.py` which returns the output in `output_queue_overflow_asimstock.json`.

For the inspector, the queue limit was set in the Petri net by having an amount of available places of the queue in a place. Therefore, it should not exceed the queue limit. This can be tested by running the option --formula="AG InspQ < 11" instead of the previous formula. This runs longer than the boundedness check of the assembler. The computer started getting a hard time after 20 minutes, so the analysis didn't come to an end. However, it is safe to assume the queue of the inspector does not overflow, as the invariants also include M(InspQ) + M(InspQAvailable) = 10.

The same result was observed when the query was run in a different machine which produced this execution log, `AssemblySimStock_Queue_Overflow_2.log`, by running this script, `LoLARunner_Queue_Overflow_AssemblySimStock_2.py`, and we obtained the output in this file, `output_queue_overflow_asimstock_2.json`. The process was killed by the machine before it reached it's time limit.

## Liveness

The liveness of entering the assembler procedure was checked by using --formula="AGEF FIREABLE(Assembly)". This makes the computer crash within a minute. Using the --cycle option does not change a thing about this. It is assumed that the chosen transition is live, as otherwise the assembler can not be entered.

In addition, performing the same analysis with the *markinglimit=602207402* on another machine, using the script `LoLARunner_Liveness_AssemblySimStock.py` produced the output in `output_liveness_asimstock.json`. As can be seen in the execution log, `AssemblySimStock_Liveness.log`, LoLA had yet to find a state in which this transition was not live by the time the marking limit was hit.

## Fairness

Fairness is assumed if all the procedures can run at the start of a time step. This is checked by using --formula="AG Time_Shape + Time_Assembly + Time_Inspection < 3". This returns that the 3 places enabling the 3 procedures are not 2-bounded, so at least all procedures can run during a time step.

This can be seen in the execution log, `AssemblySimStock_Fairness.log` generated by this script, `LoLARunner_Fairness_AssemblySimStock.py` which returns the output in `output_fairness_asimstock.json`.