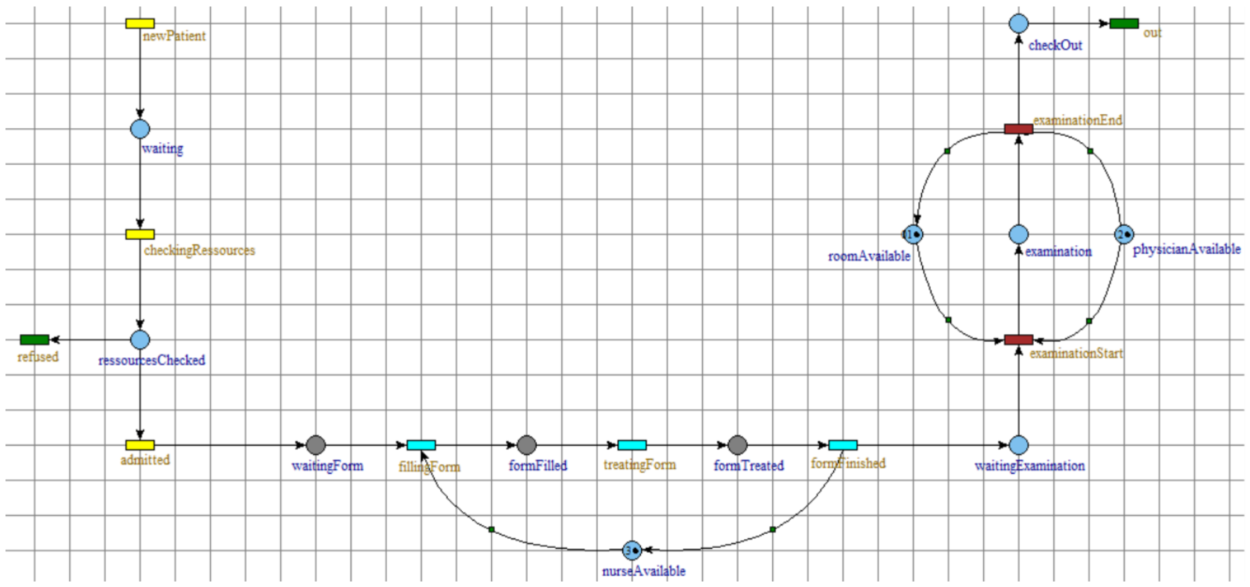
Hugues BEGEOT - 20160386
M2 Software Engineering 2020 - 2021

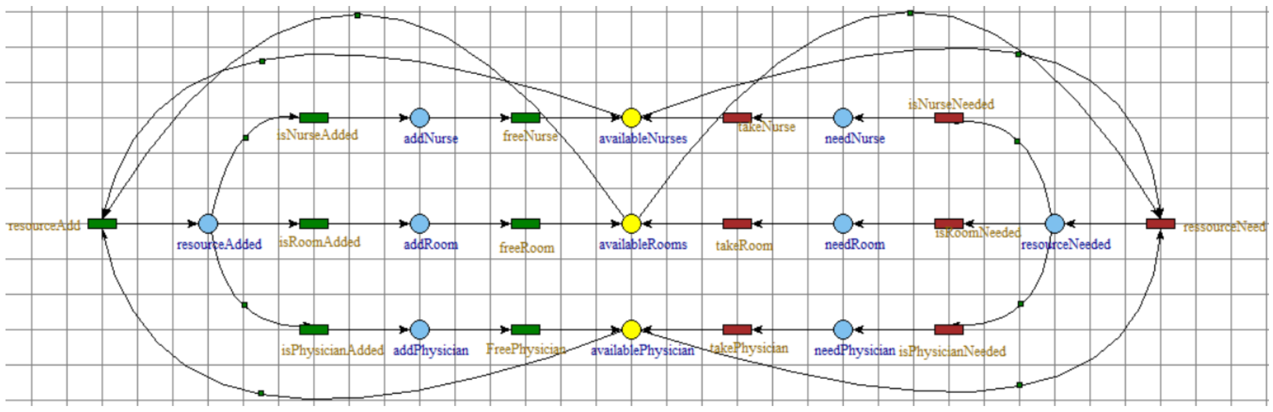
# ST2SCV : Formal Modelling

# Project M2 SE – Hospital modelization

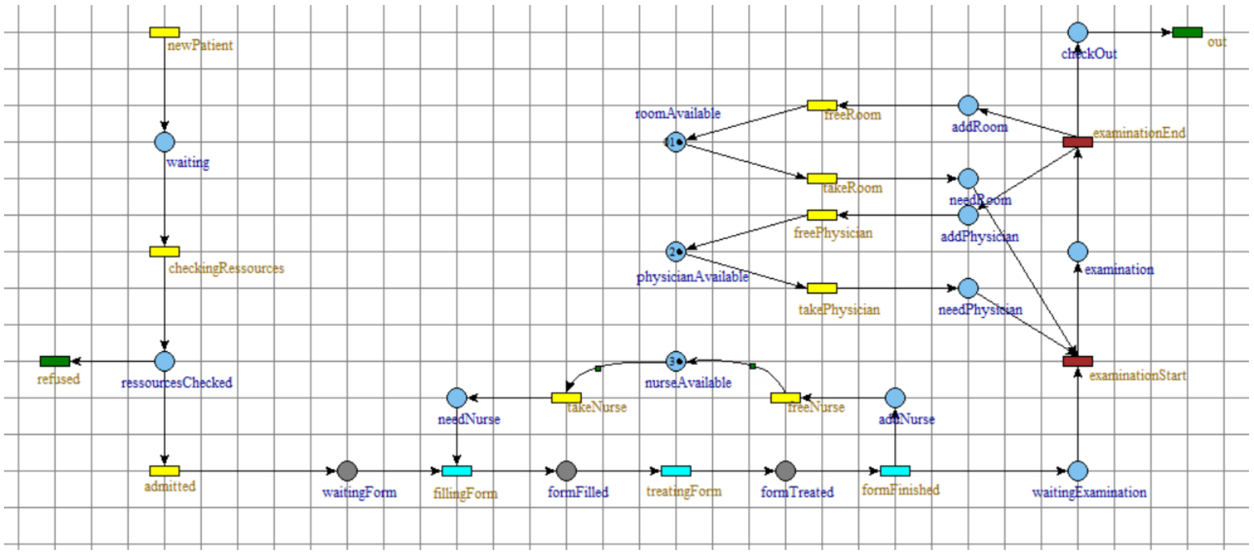
## a) Untimed Petri Nets : Emergency care model



## b) Untimed Petri Nets : Resource provider model



## c) Untimed Petri Nets : Combined model



## Implementation

*How to run it :*

*De-ZIP the project file, go to …/bin/Debug/netcoreapp3.1 and launch the .exe file*

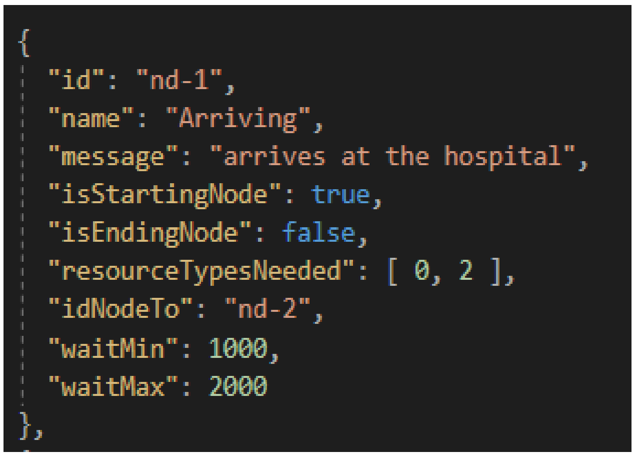
This project was implemented using C# and .NET, which has a great and efficient structure for asynchronous programming. I made sure that most of the project could be easily tweaked ; for that intent, most data can be found in JSON files and JAVA static values, ensuring that the simulation can be modified with ease.

For instance, the Petri Net is not "hardcoded" but represented in a JSON file, that the project interprets to build the simulation.

The project can be divided into :

- **Models** : representations of each element of the simulation
- **Threads** : Method simulating the behaviour of some Models (simulation of a Hospital ; simulation of a Patient)
- **Resources** : JSON files for the Petri Net and the Hospital representations, and a JAVA file that stores most variables for the simulation

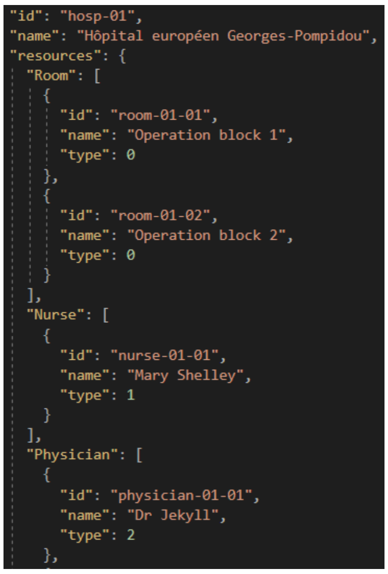
The petri-net is defined in the */database/nodes.json* file. Each state looks like the following (it does not need each value every single time, since it has default values):



```
{
  "id": "nd-1",
  "name": "Arriving",
  "message": "arrives at the hospital",
  "isStartingNode": true,
  "isEndingNode": false,
  "resourceTypesNeeded": [ 0, 2 ],
  "idNodeTo": "nd-2",
  "waitMin": 1000,
  "waitMax": 2000
},
```

-an Id
-A name
-A message to display when reached
-Whether the node is a starting one
-Whether the node is an ending one
-List of resources required to move to the next node
-Id of the next node
-Minimum waiting time in the node
-Maximum waiting time in the node

Since it is a json file, it can easily be modified to enhance the project, or to better suit a new model. In the same category, many variables of the project can be modified in the */Program-Properties.cs* file, such as the minimum / maximum time before a new Patient is created, or the default number of shared resources.

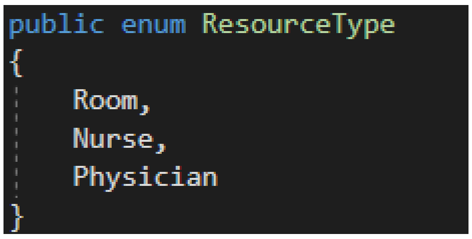


```
"id": "hosp-01",
"name": "Hôpital européen Georges-Pompidou",
"resources": {
  "Room": [
    {
      "id": "room-01-01",
      "name": "Operation block 1",
      "type": 0
    },
    {
      "id": "room-01-02",
      "name": "Operation block 2",
      "type": 0
    }
  ],
  "Nurse": [
    {
      "id": "nurse-01-01",
      "name": "Mary Shelley",
      "type": 1
    }
  ],
  "Physician": [
    {
      "id": "physician-01-01",
      "name": "Dr Jekyll",
      "type": 2
    },
```

The same goes for the hospital ; each representation contains :

- An Id
- A name
- A list of type of resource, and for each type, a list of resources

Speaking of resources : I made the whole project the most modulable possible, so the resources are defined in an enumeration :



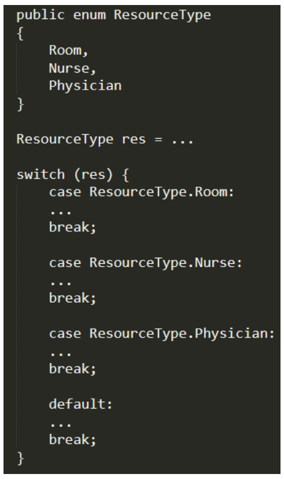
```
public enum ResourceType
{
    Room,
    Nurse,
    Physician
}
```

This allow the project to be immensely flexible, since it was built while keeping the enumeration in mind : every time a resource is needed, rather than specifically calling a Room or a Nurse, we refer globally to a Resource.

How is it more useful ? Let's take the example of a Hospital.

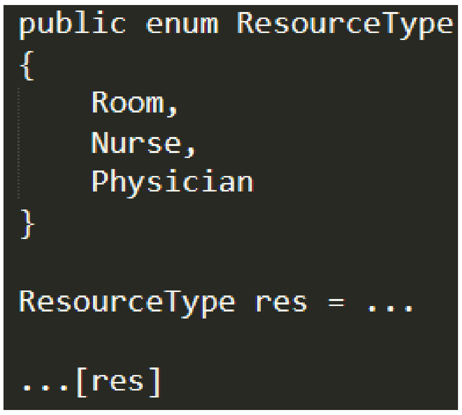
With a regular model, a Hospital would need a semaphore for each resource : one for the rooms, one for the nurses, one for the physicians... Each one is hardcoded, and every time we want to add / withdraw a resource to the project, it has to be remade.

Here, I use a dictionary : I give a resource in entry and I dynamically get the corresponding semaphore. It allows me to be extremely flexible throughout the project and to optimise the overall flow : Instead of having big to massive IF / SWITCH cases, with one statement per Resource, now it is straightforward.



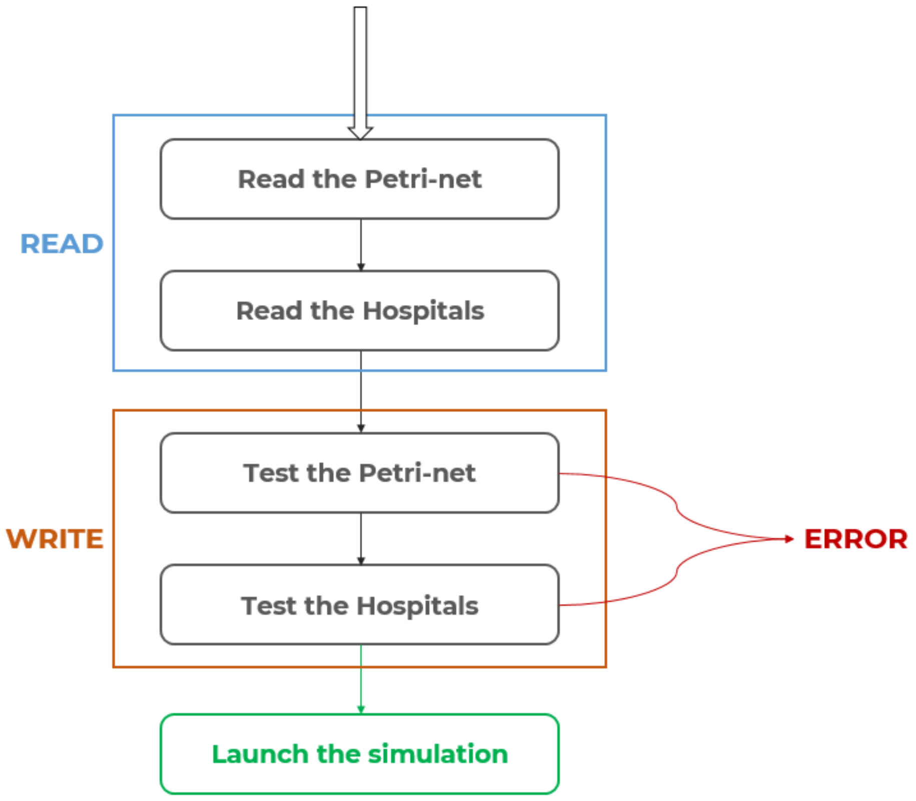
```
public enum ResourceType
{
    Room,
    Nurse,
    Physician
}

ResourceType res = ...

switch (res) {
    case ResourceType.Room:
    ...
    break;

    case ResourceType.Nurse:
    ...
    break;

    case ResourceType.Physician:
    ...
    break;

    default:
    ...
    break;
}
```

*is simplified by :*



```
public enum ResourceType
{
    Room,
    Nurse,
    Physician
}

ResourceType res = ...

...[res]
```

The program can be summarized as the following :



- We scan the resource files (JSON for the petri-net and hospitals)

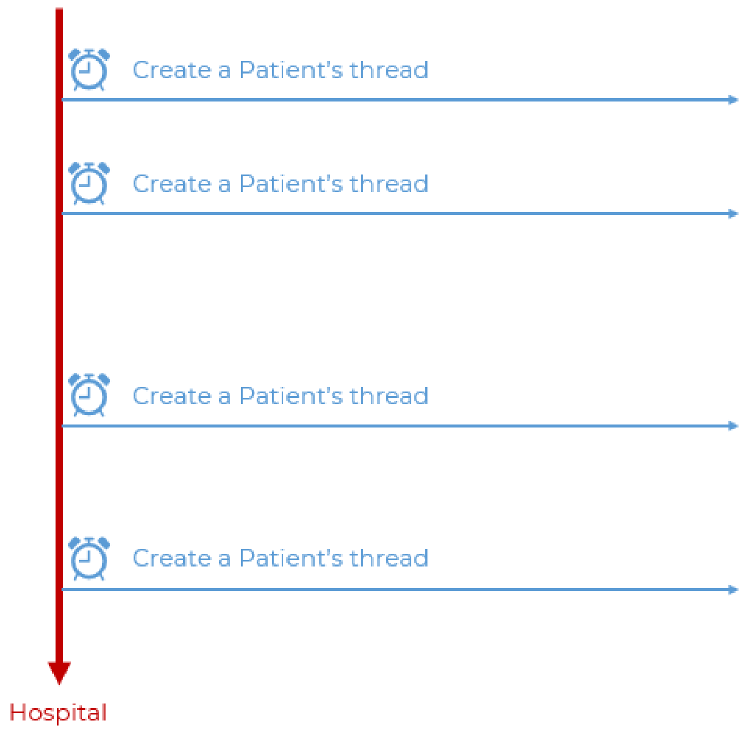
- We test the representations :
  - Petri-net :
    - No empty petri-net
    - Each state must have a proper id / name
    - At least one entrance, at least one exit
    - Each entrance must lead to an exit
    - If a state has a successor, it must be correctly defined (no pointer to non-existing state)

  - Hospital :
    - At least 1 hospital
    - Each hospital must have a proper id / name

If any of these rules is disrespected, we end the simulation ;

otherwise, we launch the simulation !

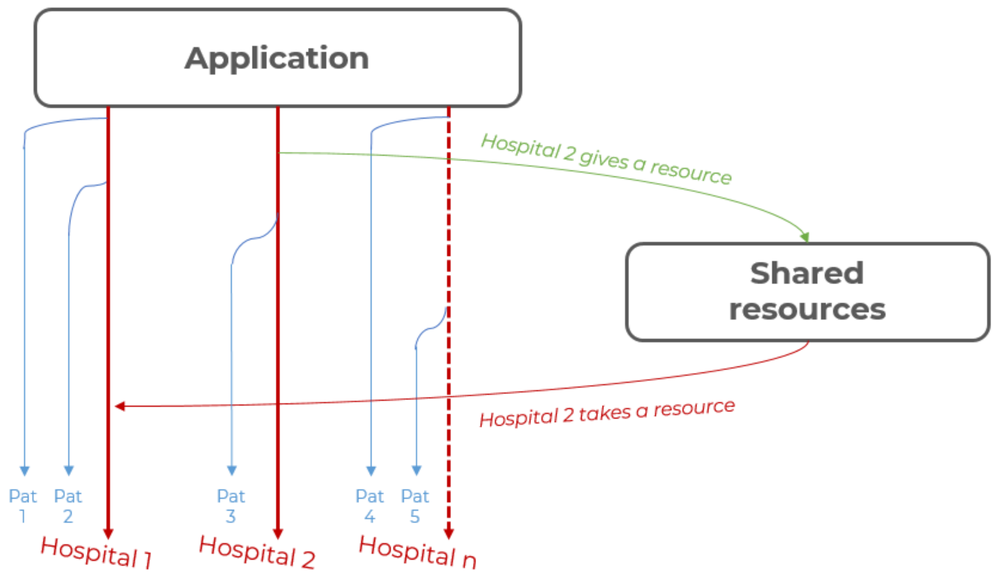
For each of the hospital, we launch a thread ; its only task is to create a new Patient and to wait for a random amount of time before creating the next.


Create a Patient's thread
Create a Patient's thread
Create a Patient's thread
Create a Patient's thread
Hospital

When a patient is created, it iterates through the petri-net's states, waiting at each step a given random amount of time, and needing / freeing resources if any.

About the **resource sharing** : when we free a resource that we have plenty of, we check if we can give it to the pool of resources shared by all the hospitals : when we need a resources that we are running low of, we check if we can borrow one.

Hence, the overall project looks like this :


Application
Hospital 2 gives a resource
Shared resources
Hospital 2 takes a resource
Pat 1
Pat 2
Pat 3
Pat 4
Pat 5
Hospital 1
Hospital 2
Hospital n