Simonin Cédric

Project final report – December 25th, 2006

Game AI (C#) synthesis from statechart

Hoshimi project

Table of contents

I] Hoshimi project presentation

I.A- Story and purpose

Created two years ago in 2005, project Hoshimi programming battle (http://www.project-hoshimi.com/) is one of the challenges proposed by the Microsoft Imagine Cup contest (http://imaginecup.com/). The main purpose of this game is to cure somebody due to the help of nano robots. In 2005, you had to cure Japanese professor Hoshimi, the inventor of this new technology and last year the Indian ambassador in Japan. In fact the nationality of this person always depends on the country where the challenge is going to happen.

Figure 1: Hoshimi posters (2005, 2006 and 2007)

The aim of the player is to define the artificial intelligence of the nano robots that will be deployed and then, act on their own in the ambassador's organism. Two players are running in the same time in the simulator, trying to cure the ambassador and getting points for this. At the end, the player with the highest number of points wins.

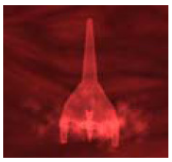
I.B- The simulation environment

The Hoshimi simulation environment is composed of different points, constitution and various kinds of nano robot can operate inside it.

I.B.a- The main points

AZN Point

This point is the location where nano robots can collect AZN molecules which will be used to cure the ambassador. The number of AZN points in the system is limited and they are represented by a small circle in the 2D simulation viewer.



Hoshimi Point

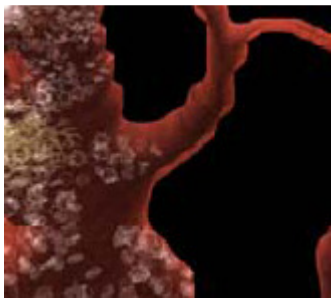
This point is the location where nano robots have to bring the AZN molecules which will be then spread into the whole organism. Players win points doing this. The number of Hoshimi points in the system is limited and they are represented by a small square in the 2D simulation viewer.

Injection point

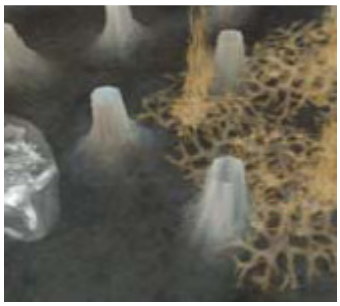
The location where all your nano robots team is injected into the organism of the ambassador. This point is different for each player and is determined by this one at the beginning of the simulation. There is only one injection point by player in the system and they are not displayed in the 2D simulation viewer.

I.B.b- The different environment constitutions

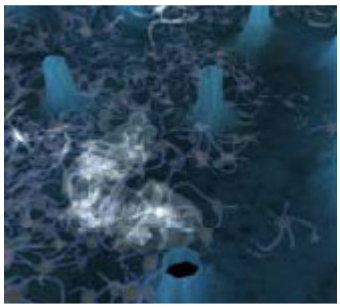
There are three types of environment in the Hoshimi simulation. In fact nano-robots can operate in the blood (in red in the 2D display), the bones (in green) or the nerves (in grey). Depending on the constitution of the environment and its density, the robots move more or less fast and this has a great incidence on your conquest speed.



Blood area
Normal density



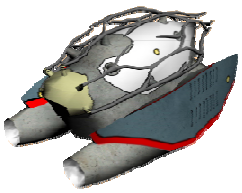
Bones area
High density



Nerves area
Highest density

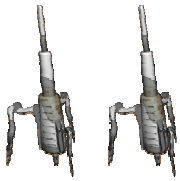
I.B.c- The different actors

In the Hoshimi simulation, your nano robots team can be composed of five types of actor, each one having its own advantages and drawbacks.



Nano AI

This robot is the leader of the team. Being the first injected into the organism, it is responsible for building the other nano robots. Moreover these robots define the whole strategy of the team and as a consequence if this one is destroyed, the game is finished.

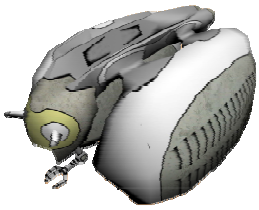


Nano Needle

This robot are created and fixed by the Nano AI on the Hoshimi points located on the blood vessels. Their purpose is just to receive AZN molecules and to deliver them to the entire body. This robot can defend themselves against enemy bots.

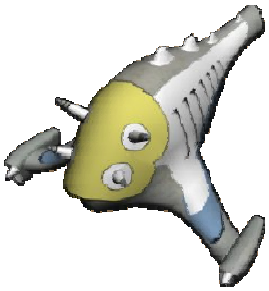
Nano Collector

This actor is the first type of robot that can collect AZN molecules from the AZN points and transfer them to the nearest needle. This robot has a very low capacity (20 molecules) but they can defend themselves.



Nano Container

This is the second type of collector robots. This one has a higher capacity (50 molecules) but it cannot defend itself.



Nano Explorer

This robot is not able to carry AZN molecules but has a very-well developed ability to scan the map. In fact, it is useful to always have information concerning enemy location at any moment of the simulation. This robot can not defend itself.

I.C- The system architecture

There are two types of actors:

- AI developers: beginner or expert, they develop the artificial intelligence of the nano robots in c# language with Visual Studio .NET.
- Community developer: creates maps and missions in order to test the different AI.

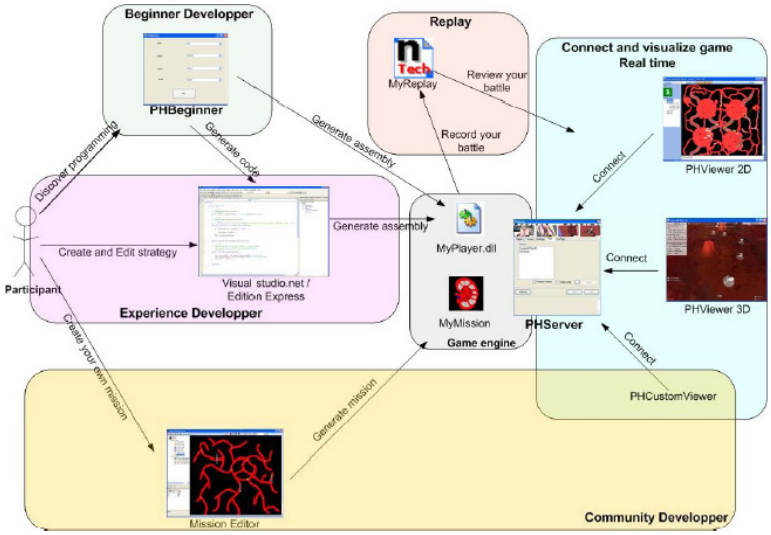


Figure 2: Architecture of the system

II] From Atom^3 to c# code

II.A- The aim of project

In the system previously described, the aim of my project is to develop a tool to help experience developer to define and create their AI. Using Atom^3 tool and the classDiagramv3 formalism, they have the possibility to graphically model their artificial intelligence and to automatically generate the c# code in the end.

This challenge containing a lot of parameters and actors, Hoshimi project is a good example of contest that requires a useful and efficient tool to model before developing.

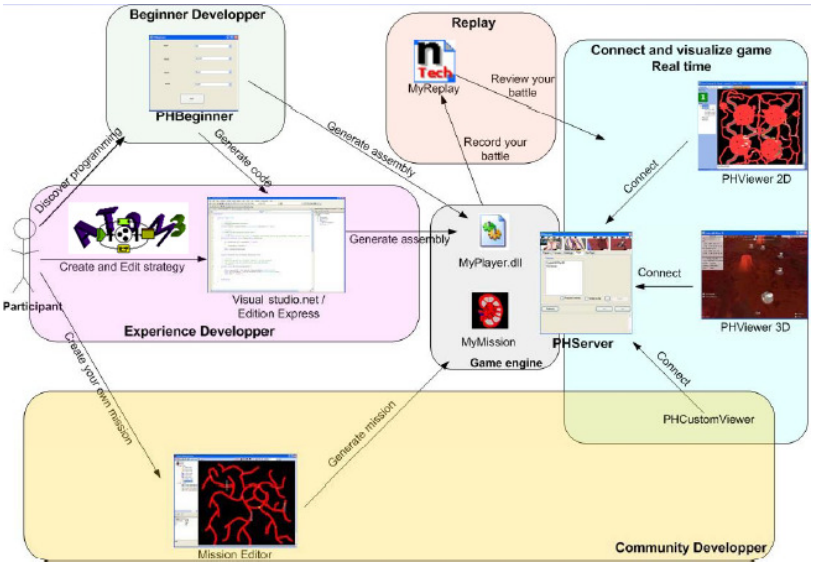


Figure 3: Aim of my project

II.B- The application Atom^3

As mentioned previously, my project consists in creating an extension to the atom3 application and more precisely to the formalism CD_ClassDiagramv3. First, this formalism allows user to model their application using UML class diagram containing variables, methods… The specificity of this application is to be able to add to each class one statechart to describe the behavior of the object defined. This ability will be very useful to map each class of a nano robot to a statechart that describe its different behavior during the simulation.

Secondly, this formalism can automatically generate code from this class and statechart diagrams. When I receive it, the different languages handled were python, C, C++ and Java. One of my purposes was also to add new functionalities to handle the translation into c# code and as a consequence fits with Hoshimi project language requirements.

Concerning statechart, the application allows us:

- To create states. For each one, user has to define if the current state is the default one and can add pieces of code that will be executed when you will enter and exit the state. We call this code: enter and exit code.
- To create edges. For each one, user can define a guard that defines if the transition can be fired or not and piece of code that will be executed when the transition is fired. We name this code: trigger code.

This formalism contains other functionalities (composite state, various parameters …) but they won't be used in this project.

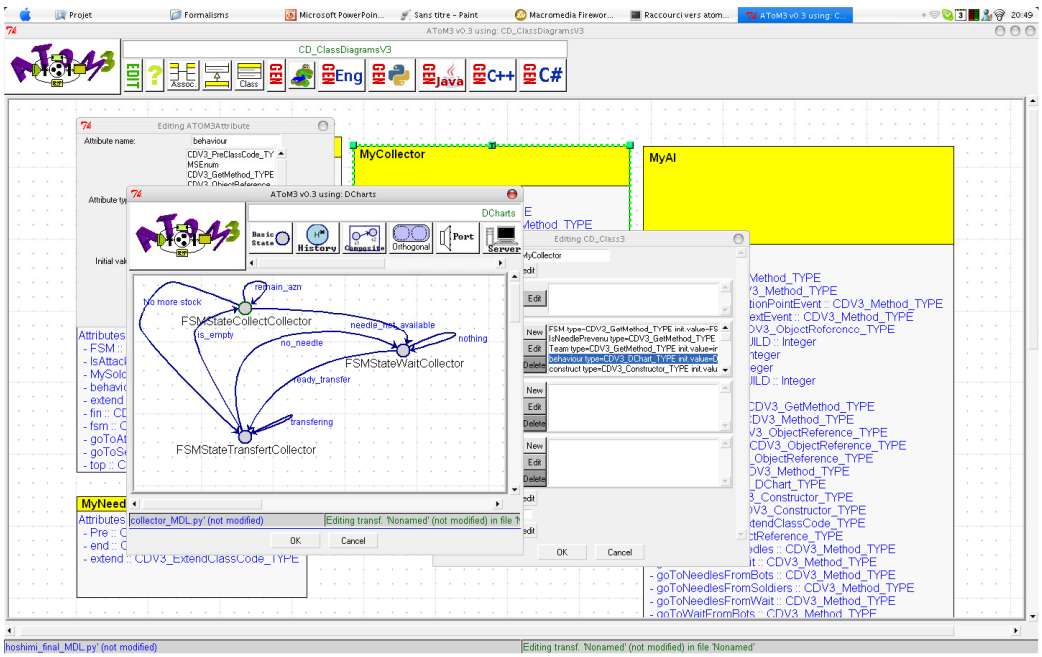


Figure 4: Atom3 GUI

II.C- Specificities of the c# language

As previously mentioned, multiple languages were already handled by the application. There are some specificities of the language:

- namespace: similar to package in java language, definition of namespace at the beginning of a file links all the files of the same project together. The particularity of this word is the fact that it includes the whole code of the file between open and close brackets. As a consequence, we define a new type in this formalism called CDV3_PostClassCode to allow the user to put some code at the end, and in particular a close braket.
- inheritance: being an object-oriented language, c# language often uses inheritance. To handle this type of class definition, we add a new type called CDV3_ExtendClassCode to allow user to add some piece of code just after the name of the class.

II.D- How does it work ?

Now that we have seen how to model and to create class diagrams under atom[3], we are going to study how to translate this into c# code.

II.D.a- Definition of a state

First of all, we created an abstract class that defines the internal behavior of a state. During this translation into c#, each state of the diagram will extend this class named FSMState. This class contains a variable *owner* to give the possibility to the code of this state to reach the methods of main class. Besides, this class declares four functions:

- enter(): defines the code that will be executed when the simulator will enter this state;
- exit(): defines the code that will be executed when the simulator will exit this state;
- update(): defines the code that will be executed when the simulator will remain in this state;
- getNextTransition(): returns the next possible state or the current one depending on guards.

This is the c# code of this class:

```
public abstract class FSMState
{
    protected object owner = null;

    public FSMState(object owner) {
        this.owner = owner;
    }

    public object Owner
    {
        get { return owner; }
        set { owner = value; }
    }

    public abstract void Enter();
    public abstract void Exit();
    public abstract void Update();
    public abstract Type getNextTransition();
}
```

II.D.b- Definition of a statechart machine

You are now going to define the behavior of the statechart machine. This class named FSMMachine contains a list with all the possible states it can reach and two variables that store current and default states. In addition, this class defines 5 functions:

- Addstate(): adds a FSMState object to the list of possible states;
- SetDefaultState(): initializes the value of the variable that keeps the information concerning the default state;
- getCurrentState(): returns the value of the current state;
- existsState(): checks if a given state exists in our model. If not, the statechart returns to its default state ;
- Update(): running method of the machine, it first checks if the state is going to change via the getNextTransition function of this one. If so, the method runs the exit function of the current state, substitutes this one with the new state and executes the enter function of this one. In all the cases, the method executes in the end the update function of the current state.

```
public class FSMMachine
{
    // Declaration of the variables
    protected List<FSMState> states = new List<FSMState>(5); // List of
all the states
    protected FSMState currentState = null;  // Current state
    protected FSMState defaultState = null; // state by default

    public void AddState(FSMState state) { states.Add(state); }
    public void SetDefaultState(FSMState state) { this.defaultState =
state; }
    public FSMState getCurrentState() { return this.currentState; }

    private FSMState existsState(Type type) {
        foreach (FSMState state in states) {
            if (state.GetType() == type)
                return state;
        }
        return defaultState;
    }

    public void Update()
```

```
        {
            // If there is at least one state
            if (states.Count != 0)
            {
                // If it is the first iteration
                if (currentState == null)
                {
                    // current state = default state
                    currentState = defaultState;
                    // even default state was not define
                    if (currentState == null)
                        // we stop
                        return;
                }

                Type oldStateType = currentState.GetType();
                Type newStateType = currentState.getNextTransition();

                // If the state has changed
                if (oldStateType != newStateType)
                {
                    currentState.Exit(); // exit the current state
                    currentState = existsState(newStateType); // go to the
next state
                    currentState.Enter(); // enter the state
                }
                // update the state
                currentState.Update();
            }
        }
    }
```

II.D.c- Translation into c# code

After having defined these two classes, we are going to study the translation process described in the file exporter_CSharpExporter.py.

The first part of this process is the translation of the pure class diagram into c#. Theses are the main steps:

- Printing of the preClass code that contains imports (using in c#), namespace and characteristics of the class defined;
- Printing of the name of the class with the possible inheritance;
- Declaration of all the variables;
- Declaration of the constructor. In each constructor and if a statechart is defined in the class, it calls the method processChartConstructor. This method reads all the states of diagram, declares a FSMState variable for each, adds them to the FSM Machine states list of the class and defines the default state when it finds it;
- Definition of all the methods;
- Printing of the postClass code.

This part of the translation is handled by the following code:

```
    def run(self):

        methodList = []
        constructorList = []
        destructorList = []
```

```
        self.fOut =
exporter_FileOutputer.FileOutputer(self.classDiagram.name + ".cs.txt")

        self.fOut.write("using System;")
        self.fOut.write("using System.Collections.Generic;");
        self.fOut.write("using System.Text;");

        for code in self.classDiagram.preClassCode:
            self.writeCode(code)

        extend = "";
        for code in self.classDiagram.extendClassCode:
            extend = code

        self.fOut.indent()
        self.fOut.write("class " + self.classDiagram.name + extend)
        self.fOut.write("{")
        self.fOut.indent()

        for attribute, type, initValue in self.classDiagram.attributes:
            if type == "Float" :
                self.fOut.write("public float " + attribute + " = " +
initValue + ";")
            if type == "Integer" :
                self.fOut.write("public int " + attribute + " = " +
initValue + ";")
            if type == "Boolean" :
                if initValue == True:
                    self.fOut.write("public bool " + attribute + " =
true;")
                else:
                    self.fOut.write("public bool " + attribute + " =
false;")

        for name, type in self.classDiagram.objectRef:
            self.fOut.write("public " + type + " " + name + ";")

        self.fOut.write()

        for parameters, body in self.classDiagram.constructors:
            if (extend==" : VG.Common.Player\n" or extend==" :
VG.Common.Player") and parameters=="string _name, int _id":
                self.writeConstructor(self.classDiagram.name, parameters,
body,": base(_name, _id)")
            else:
                self.writeConstructor(self.classDiagram.name, parameters,
body,"")

        for name, parameters, returnType, body in
self.classDiagram.getMethods:
            self.writeGetMethod(name, returnType, body)

        for name, parameters, returnType, body in
self.classDiagram.methods:
            self.writeMethod(name, parameters, returnType, body)

        for body in self.classDiagram.destructors:
            self.writeMethod("end", "", "void", body)

        self.fOut.dedent()
        self.fOut.write("}")
```

```
        if self.stateChart != None:
           self.processChart(self.classDiagram.name)

        self.fOut.write()
        for code in self.classDiagram.postClassCode:
            self.fOut.dedent()
            self.writeCode(code)
        self.fOut.write()

        self.fOut.close()

    def writeCode(self, body):
        lineByLine = body.split("\n")
        for line in lineByLine:
            self.fOut.write(line)

    def writeConstructor(self, name, parameters, body, extension):
        self.fOut.write("public " + name + "(" + parameters + ")
"+extension+" {")
        self.fOut.indent()
        self.writeCode(body)
        if self.stateChart != None:
            self.processChartConstructor()
        self.fOut.dedent()
        self.fOut.write("}")

    def writeMethod(self, name, parameters, returnType, body):
        self.fOut.write("public " + returnType + " " + name + "(" +
parameters + ") {")
        self.fOut.indent()
        self.writeCode(body)
        self.fOut.dedent()
        self.fOut.write("}\n")

    def processChartConstructor(self):
        self.fOut.write("fsm = new FSMMachine();");
        for state in self.stateChart.basics:
            if self.stateChart.transitionData[state] != [] :
                self.fOut.write("FSMState "+state.name.getValue()+"_var =
new "+state.name.getValue()+"(this);")

self.fOut.write("fsm.AddState("+state.name.getValue()+"_var);");
                if (self.stateChart.initState == state.name.getValue()):

self.fOut.write("fsm.SetDefaultState("+state.name.getValue()+"_var);")

    def writeGetMethod(self, name, returnType, body):
        self.fOut.write("public " + returnType + " " + name + " {")
        self.fOut.indent()
        self.writeCode(body)
        self.fOut.dedent()
        self.fOut.write("}\n")
```

The second part of this translation happens if a statechart is defined in the class diagram. If so, the function processChart is called just before the printing of the postClass code. This one creates a new class for each states of the diagram extending FSMState class (class nameOfthestate : FSMState) and containing by default an object called owner and having the type of the current main class. By example, if we are generating c# code for a class named robot, the type of this object will be robot.

Then for each state, the function processTransition initializes the different methods of the state previously described. For each edge leaving a state:

- If the destination state is the same than the source one, the trigger code of this arrow will correspond to the update function.
- If the destination state is different than the source one, the following code is added to the body of getNextTransition function:

```
if ([GUARD]) {
     [ trigger code ]
    return typeof([Destination state]);
}
```

This part of the translation is handled by the following code:

```
    def processChart(self,classe):
        for state in self.stateChart.basics:
            # If the state has transition
            if self.stateChart.transitionData[state] != [] :
                self.processState(state,classe);


    def processState(self, state,classe):
        self.fOut.write("public class "+state.name.getValue()+" :
FSMState")
        self.fOut.write("{");
        self.fOut.indent()
        self.fOut.write("public "+state.name.getValue()+"(object owner) :
base(owner) { /* Do nothing */ }");
        self.processTransition(state,classe)
        self.fOut.dedent()
        self.fOut.write("}");

    def processTransition(self, state,classe):
        codes = []
        guards = []
        nextState = []
        transitions = self.stateChart.transitionData[state]
        transitionOk = 0
        for transition in transitions:
            if transition["destinationState"] == state.name.getValue():
                self.fOut.write("public override void Update()");
                self.fOut.write("{");
                self.fOut.indent()
                for code in transition["triggerCode"]:
                    self.writeCode(code)
                self.fOut.dedent()
                self.fOut.write("}");

                self.fOut.write("public override void Enter()");
                self.fOut.write("{");
                self.fOut.indent()
                if len(transition["enterCode"])>0:
                    for code in transition["enterCode"]:
                        self.writeCode(code)
                else:
                    self.fOut.write("//Rien à faire ici");
                self.fOut.dedent()
                self.fOut.write("}");

                self.fOut.write("public override void Exit()");
                self.fOut.write("{");
                self.fOut.indent()
```

```
            if len(transition["exitCode"])>0:
                for code in transition["exitCode"]:
                    self.writeCode(code)
            else:
                self.fOut.write("//Rien à faire ici");
            self.fOut.dedent()
            self.fOut.write("}");
        else :
            transitionOk = transitionOk + 1
            if len(transition["triggerCode"])>0:
                codes.append(transition["triggerCode"])
                guards.append(transition["guard"])
                nextState.append(transition["destinationState"]);

    if transitionOk > 0 or len(transitions)==1:
        self.fOut.write("public override Type getNextTransition()");
        self.fOut.write("{");
        self.fOut.indent()
        self.fOut.write(classe+" "+classe+"_var = ("+classe+")owner;");
        if len(codes)>=1:
            i=0
            for code in codes:
                if guards[i]=="1":
                    guards[i]="true"
                self.fOut.write("if ("+classe+"_var."+guards[i]+") {");
                self.fOut.indent()
                for code in codes[i]:
                    self.writeCode(code)
                self.fOut.write("return typeof("+nextState[i]+");");
                self.fOut.dedent()
                self.fOut.write("}");
                i = i+1
        self.fOut.write("return this.GetType();");
        self.fOut.dedent()
        self.fOut.write("}");
```

II.E- The simulator

The simulator developed by Microsoft is quite easy to understand. The program is based on an infinite while loop. This loop calls at each iteration a function called MyAI_WhatToDoNextEvent() declared in the NANO AI class. With the function MyAI_ChooseInjectionPointEvent() that chooses the injection point of the robots, this function must appear in the definition of your AI. Moreover, this one calls the update function of each robots FSM machine. This is the way how the robots evolve during the simulation.

Finally, the simulation is over after a certain number of turns depending on the requirements of the mission.

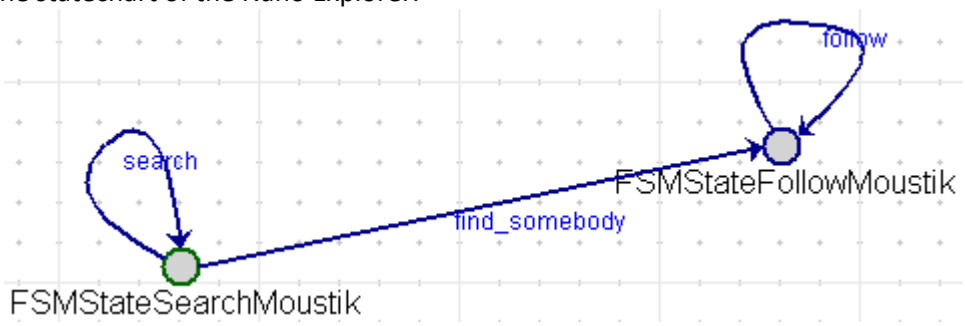
III] Modelisation of the artificial intelligence

III.A- Modelisation of the explorer

The first behavior that I have modeled was the one of the nano explorer. This robot has only two states:

- **FSMStateSearchMoustik**: state by default, the nano explorer is looking for someone randomly, traveling around the map.
- **FSMStateFollowMoustik**: the nano explorer found an enemy and decides to follow it. In my implementation it follows the enemy Nano AI.

This is the statechart of the Nano Explorer:



This is the c# code produced:

```
class MyMoustik : VG.Common.NanoExplorer
{
  …
    public MyMoustik()  {
        fsm = new FSMMachine();
        FSMState FSMStateSearchMoustik_var = new
         FSMStateSearchMoustik(this);
        fsm.AddState(FSMStateSearchMoustik_var);
        fsm.SetDefaultState(FSMStateSearchMoustik_var);
        FSMState FSMStateFollowMoustik_var = new
         FSMStateFollowMoustik(this);
        fsm.AddState(FSMStateFollowMoustik_var);
    }

    public bool goToFollowFromSearch() {
        …
    }

}

public class FSMStateSearchMoustik : FSMState
{
  …
    public override void Update(){
        …
    }
    public override void Enter(){
    }
    public override void Exit(){
    }
    public override Type getNextTransition(){
        MyMoustik MyMoustik_var = (MyMoustik)owner;
```

```
            if (MyMoustik_var.goToFollowFromSearch() == true) {
                return typeof(FSMStateFollowMoustik);
            }
            return this.GetType();
        }
    }
    public class FSMStateFollowMoustik : FSMState
    {
      …
        public override void Update(){
            …
        }
        public override void Enter(){
        }
        public override void Exit(){
        }
        public override Type getNextTransition(){
            MyMoustik MyMoustik_var = (MyMoustik)owner;
            return this.GetType();
        }
    }
}
```

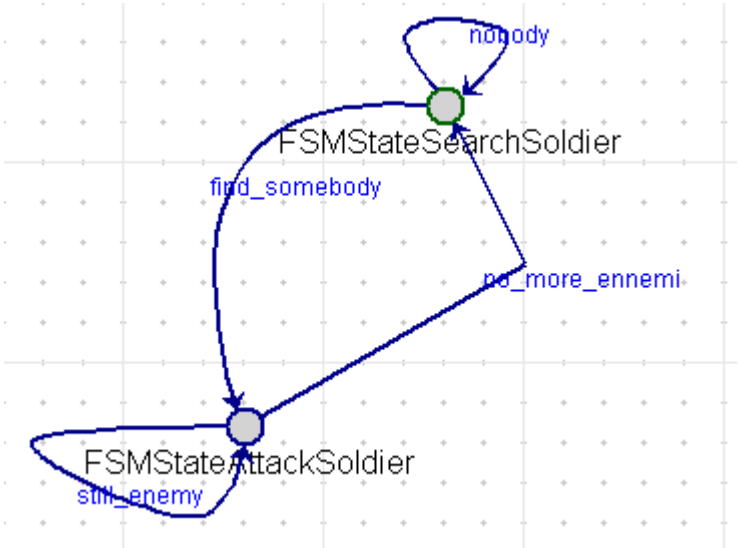
III.B- Modelisation of the nano collector

For the purpose of the project, the nano collector is only used as a soldier having the possibility to attack its enemy. Like the nano explorer, this robot has two states:

- **FSMStateSearchSoldier**: state by default, the soldier is wandering in the map without being attacked by an enemy.
- **FSMStateAttackSoldier**: the soldier is defending itself, attacking the enemy.

Contrary to the nano explorer, this robot can come back to the search state if the enemy disappears.

This is the statechart of the nano collector:



This is the code produced:

```
    class MySoldier : NanoCollector
    {
        public bool IsAttacking = false;
        …
        public MySoldier(){
```

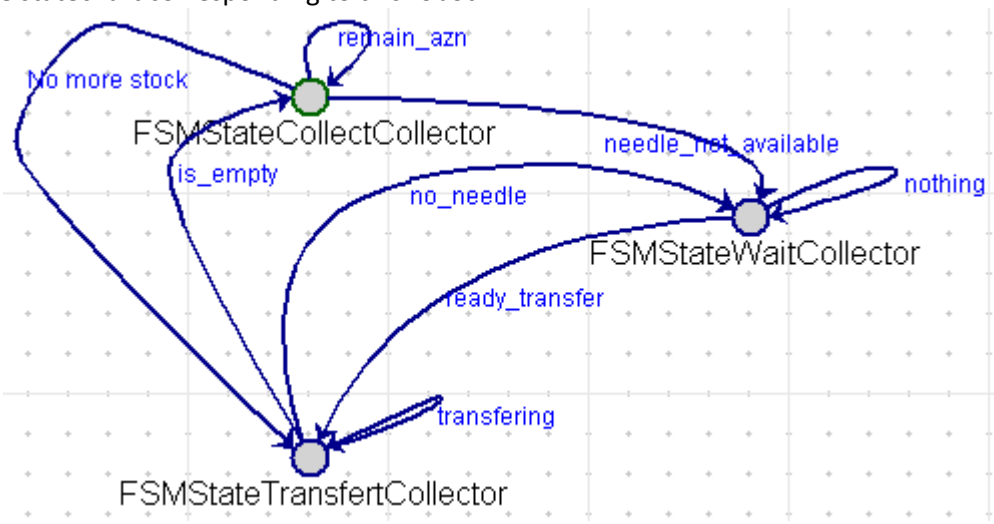
```
        this.IsAttacking = false;
        fsm = new FSMMachine();
        FSMState FSMStateAttackSoldier_var = new
         FSMStateAttackSoldier(this);
        fsm.AddState(FSMStateAttackSoldier_var);
        FSMState FSMStateSearchSoldier_var = new
         FSMStateSearchSoldier(this);
        fsm.AddState(FSMStateSearchSoldier_var);
        fsm.SetDefaultState(FSMStateSearchSoldier_var);
    }
    …
    public bool goToAttack() {
        …
    }
    public bool goToSearch() {
        …
    }
}

public class FSMStateAttackSoldier : FSMState
{
    …
    public override void Update(){
        …
    }
    public override void Enter(){
    }
    public override void Exit() {
    }
    public override Type getNextTransition()
    {
        MySoldier MySoldier_var = (MySoldier)owner;
        if (MySoldier_var.goToSearch() == true) {
            return typeof(FSMStateSearchSoldier);
        }
        return this.GetType();
    }
}

public class FSMStateSearchSoldier : FSMState
{
    …
    public override void Update(){
        …
    }
    public override void Enter(){
    }
    public override void Exit(){
    }
    public override Type getNextTransition(){
        MySoldier MySoldier_var = (MySoldier)owner;
        if (MySoldier_var.goToAttack() == true) {
            return typeof(FSMStateAttackSoldier);
        }
        return this.GetType();
    }
}
```

III.C- Modelisation of the container

This is the third robot I modeled. The complexity of this one was the fact that there were multiple leaving edges. This robot has three different states:

- **FSMStateCollectCollector**: state by default, the nano collector collects or is going to collect some AZN molecules from an AZN point;
- **FSMStateWaitCollector**: after collecting cure, the robot is waiting for a needle to be built on an Hoshimi point;
- **FSMStateTransferCollector**: after collecting cure, the robot is transferring its AZN molecules to an available needle.

This is the statechart corresponding to this robot:



This is the code produced:

```
class MyCollector : VG.Common.NanoContainer
{
    …
    public MyCollector()  {
        …
        fsm = new FSMMachine();
        FSMState FSMStateTransfertCollector_var = new
         FSMStateTransfertCollector(this);
        fsm.AddState(FSMStateTransfertCollector_var);
        FSMState FSMStateCollectCollector_var = new
         FSMStateCollectCollector(this);
        fsm.AddState(FSMStateCollectCollector_var);
        fsm.SetDefaultState(FSMStateCollectCollector_var);
        FSMState FSMStateWaitCollector_var = new
         FSMStateWaitCollector(this);
        fsm.AddState(FSMStateWaitCollector_var);
    }
    public bool goToCollectFromTransfert() {
        …
    }
    public bool goToTransfertFromCollect() {
        …
    }
    public bool goToTransfertFromWait() {
        …
     }
```

```
        public bool goToWaitFromCollect() {
            …
        }
        public bool goToWaitFromTransfert() {
            …
        }
    }

    public class FSMStateTransfertCollector : FSMState
    {
        …
        public override void Update(){
            …
        }
        public override void Enter(){
        }
        public override void Exit(){
        }
        public override Type getNextTransition(){
            MyCollector MyCollector_var = (MyCollector)owner;
            if (MyCollector_var.goToCollectFromTransfert() == true) {
                return typeof(FSMStateCollectCollector);
            }
            if (MyCollector_var.goToWaitFromTransfert() == true) {
                return typeof(FSMStateWaitCollector);
            }
            return this.GetType();
        }
    }

public class FSMStateCollectCollector : FSMState
    {
        …
        public override void Update() {
            …
        }
        public override void Enter(){
        }
        public override void Exit(){
        }
        public override Type getNextTransition(){
            MyCollector MyCollector_var = (MyCollector)owner;
            if (MyCollector_var.goToTransfertFromCollect() == true) {
                return typeof(FSMStateTransfertCollector);
            }
            if (MyCollector_var.goToWaitFromCollect() == true) {
                return typeof(FSMStateWaitCollector);
            }
            return this.GetType();
        }
    }

public class FSMStateWaitCollector : FSMState
    {
        …
        public override void Update(){
        }
        public override void Enter(){
        }
        public override void Exit(){
        }
```

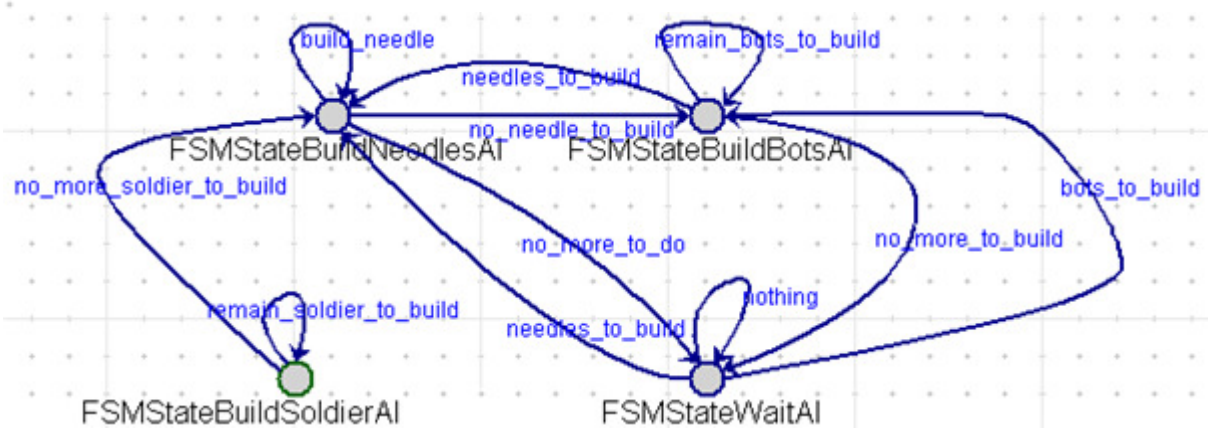
```
        public override Type getNextTransition(){
            MyCollector MyCollector_var = (MyCollector)owner;
            if (MyCollector_var.goToTransfertFromWait() == true) {
                return typeof(FSMStateTransfertCollector);
            }
            return this.GetType();
        }
    }
```

III.D- Modelisation of the artificial intelligence

This is the last artificial intelligence that I have defined and the most complex one. This robot has four possible states:

- **FSMStateBuildSoldierAI**: state by default, the nano AI begins the simulation by building the soldiers (nano collector, nano explorer);
- **FSMStateBuildNeedlesAi**: the nano robot is building a needle on the nearest Hoshimi point from the location it had when it reached this state;
- **FSMStateBuildBotsAI**: the nano robot is creating non-soldier robots (container);
- **FSMStateWaitAI**: The nano robot is waiting for the possibility to build another bot or another needle.

This is the statechart of this robot:



This is the c# code produced:

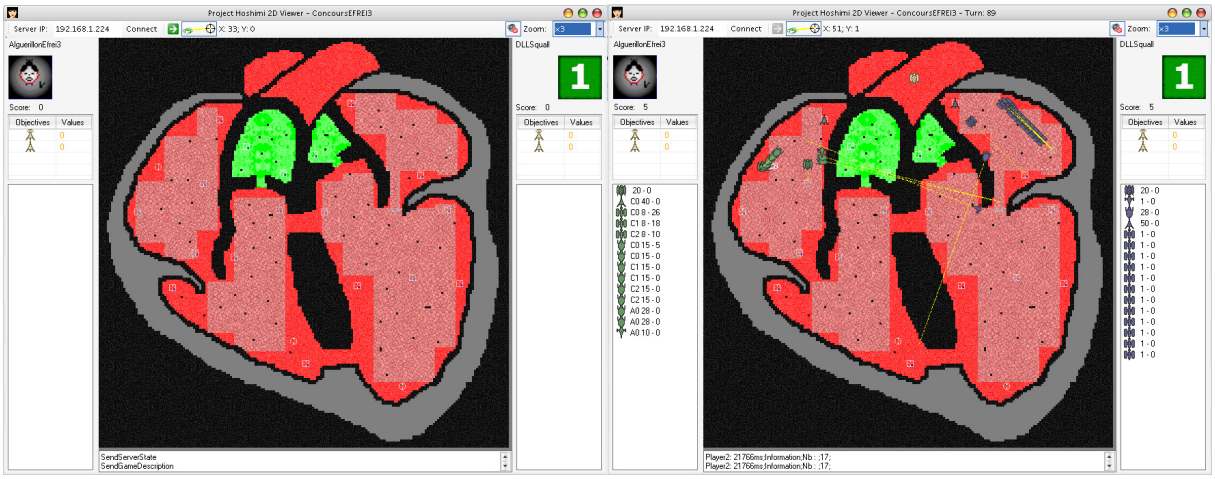
```
    class MyAI : VG.Common.Player
    {
        public MyAI(string _name, int _id) : base(_name, _id) {
             …
            fsm = new FSMMachine();
            FSMState FSMStateBuildSoldierAI_var = new
             FSMStateBuildSoldierAI(this);
            fsm.AddState(FSMStateBuildSoldierAI_var);
            fsm.SetDefaultState(FSMStateBuildSoldierAI_var);
            FSMState FSMStateBuildNeedlesAI_var = new
             FSMStateBuildNeedlesAI(this);
            fsm.AddState(FSMStateBuildNeedlesAI_var);
            FSMState FSMStateBuildBotsAI_var = new
             FSMStateBuildBotsAI(this);
            fsm.AddState(FSMStateBuildBotsAI_var);
            FSMState FSMStateWaitAI_var = new FSMStateWaitAI(this);
            fsm.AddState(FSMStateWaitAI_var);
        }
        …
        public void MyAI_ChooseInjectionPointEvent() {
```

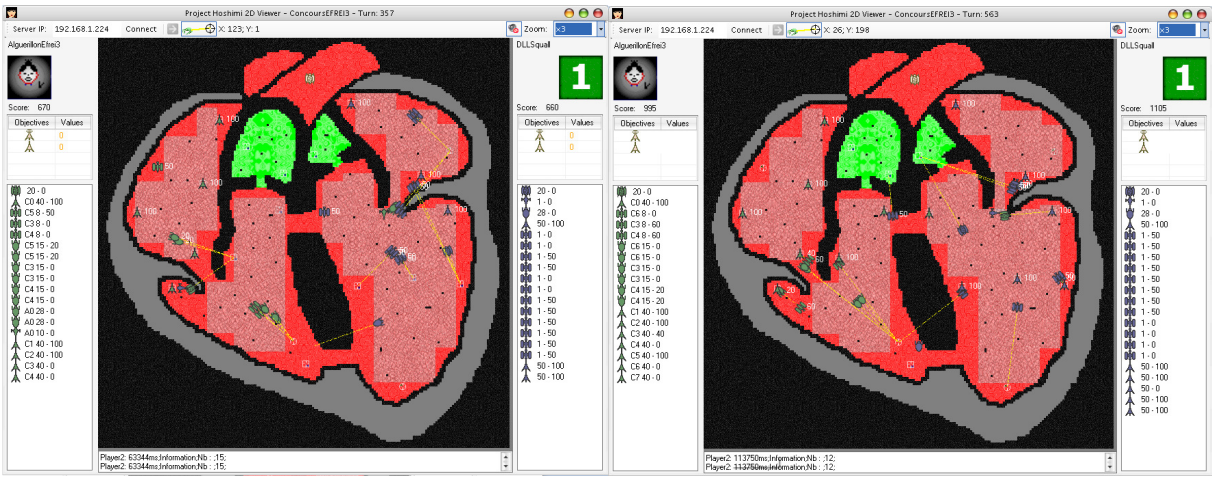
```
    …
}

public void MyAI_WhatToDoNextEvent() {
    …
    this.nbCollector = 0;
    this.nbNeedle = 0;
    this.nbSoldier = 0;
    this.nbMoustik = 0;
    foreach (NanoBot bot1 in base.NanoBots)
    {
          if (bot1 is MyCollector)
          {
                this.nbCollector++;
                if (((MyCollector)bot1).Team == -1)
                {
                      this.UpdateTeams();
                }
                ((MyCollector)bot1).FSM.Update();
          }
          else if (bot1 is MyNeedle)
          {
                this.nbNeedle++;
          }
          else if (bot1 is MySoldier)
          {
                this.nbSoldier++;
                ((MySoldier)bot1).FSM.Update();
          }
          else if (bot1 is MyMoustik)
          {
                this.nbMoustik++;
                ((MyMoustik)bot1).FSM.Update();
          }
    }
    this.fsm.Update();
}

public bool goToBotsFromNeedles() {
    …
}

public bool goToBotsFromWait() {
    …
}

public bool goToNeedlesFromBots() {
    …
}

public bool goToNeedlesFromSoldiers() {
    …
}

public bool goToNeedlesFromWait() {
    …
}

public bool goToWaitFromBots() {
    …
}
```

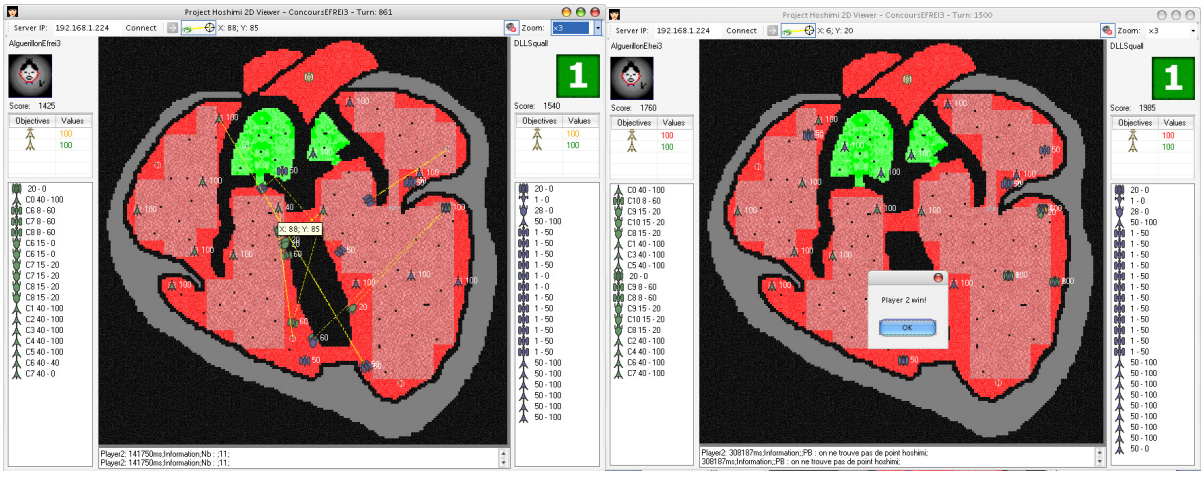
```
    public bool goToWaitFromNeedles() {
        …
    }

}

 public class FSMStateBuildSoldierAI : FSMState
{
    …
    public override void Update(){
        …
    }
    public override void Enter()
    {
    }
    public override void Exit()
    {
    }
    public override Type getNextTransition(){
        MyAI MyAI_var = (MyAI)owner;
        if (MyAI_var.goToNeedlesFromSoldiers() == true) {
            return typeof(FSMStateBuildNeedlesAI);
        }
        return this.GetType();
    }
}

public class FSMStateBuildNeedlesAI : FSMState
{
    …
    public override void Update() {
        …
    }
    public override void Enter() {
    }
    public override void Exit() {
    }
    public override Type getNextTransition(){
        MyAI MyAI_var = (MyAI)owner;
        if (MyAI_var.goToWaitFromNeedles() == true) {
            return typeof(FSMStateWaitAI);
        }
        if (MyAI_var.goToBotsFromNeedles() == true) {
            return typeof(FSMStateBuildBotsAI);
        }
        return this.GetType();
    }
}

public class FSMStateBuildBotsAI : FSMState
{
    …
    public override void Update() {
         …
    }
    public override void Enter() {
    }
    public override void Exit() {
    }
    public override Type getNextTransition() {
```

```
        MyAI MyAI_var = (MyAI)owner;
        if (MyAI_var.goToWaitFromBots() == true) {
            return typeof(FSMStateWaitAI);
        }
        if (MyAI_var.goToNeedlesFromBots() == true) {
            return typeof(FSMStateBuildNeedlesAI);
        }
        return this.GetType();
    }
}

public class FSMStateWaitAI : FSMState
{
    …
    public override void Update() {
    }
    public override void Enter() {
    }
    public override void Exit() {
    }
    public override Type getNextTransition() {
        MyAI MyAI_var = (MyAI)owner;
        if (MyAI_var.goToNeedlesFromWait() == true) {
            return typeof(FSMStateBuildNeedlesAI);
        }
        if (MyAI_var.goToBotsFromWait() == true) {
            return typeof(FSMStateBuildBotsAI);
        }
        return this.GetType();
    }
}
```

IV] Overview of the simulation







Conclusion

We successfully create an intelligent and efficient player that wins against another one. As a consequence, the aim of this project is fulfilled: we extended Atom3 formalism to give to developers an easy solution to define and model their artificial intelligence and to automatically generate c# code.

This project gives me the will to keep on developing my artificial intelligence powered by Atom3. This is the reason why, I have registered for the next session of the project Hoshimi programming battle and why the code source of the player realized for this project doesn't appear online.