Describing Non-Player Character Behaviour using Statecharts

Kevin Wyckmans

University of Antwerp

Abstract

In contemporary computer games, users demand and expect realistic behaviour of Non Player Characters (NPC's). Coding this behaviour is hard and often leads to complex code. On top of this, the people that design the behaviour are not necessarily the ones coding it, which can lead to errors. Previous work has shown that it is possible to use Statecharts to model this behaviour on a higher, more appropriate abstraction level. In this paper, this technique is used to model NPC behaviour in the context of agent-based spreading of an infectious disease. Finally, we let a statechart compiler synthesize python code to use this in a simple custom made game.

Keywords: artificial intelligence, statecharts, game development, modelling

1. Introduction

Present-day computer games are becoming more and more realistic, not only in the way they look, but in the way computer-controlled characters, called *Non-Player Characters* (NPC's) behave. Whatever the kind of game may be, users expect NPC's to behave in a realistic way. Transforming the specifications of this kind of behaviour to code is not easy. This code is often not modular or easy adaptable to new situations.

In (Kienzle et al., 2007) it is shown that *Statecharts* can be used to specify the behaviour of NPC's in real time games at a more appropriate abstraction level, in a reusable way. On top of that, the *Statechart* formalism is not hard to learn and can be used more easily by people who are not software experts. In this paper we follow the structure proposed to describe the behaviour of NPC's in the context of agent-based spreading of an infectious disease. This

means that the actions and interactions of autonomous agents are described in the presence of a highly contagious disease. This can be used to simulate how a disease spreads in a small group of people, or it can be used as the foundation for a computer game in which a player has to survive.

Section 2 of this paper describes how the modelling of game AI is approached, and gives a concrete implementation by designing the AI for healthy and sick people in the setting mentioned above. Section 3 explains how the code generated from this model is used in a concrete game. Finally, we come to a conclusion in section 4.

2. Modelling Behaviour using statecharts

As mentioned before the behaviour of NPC's in the presence of an infectious disease is modelled. More specifically, in this system there are a number of healthy people, as well as a number of sick people present. If a healthy person spots a sick person, he will try to run away, while the sick people will do just the opposite, they will chase the healthy people if they see one. Once a healthy person is touched by a sick one, he will become sick himself. The sick people die after a certain amount of time.

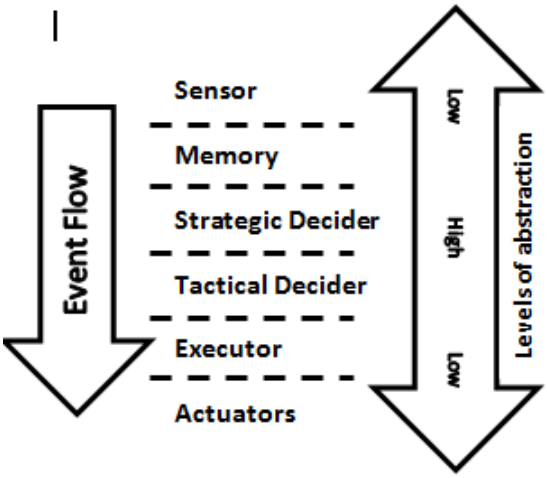

Figure 1: Different levels of abstraction used

The behaviour we need is divided into different levels of abstraction as shown in 1. This will make sure our statecharts stay simple and concise and make sure that every statechart has exactly one goal. As stated before this is based on Kienzle et al. (2007), but not all levels were needed. We will not repeat the how and why of every level, we will just explain how we specified our needed behaviour using this structure.

2.1. Sensor

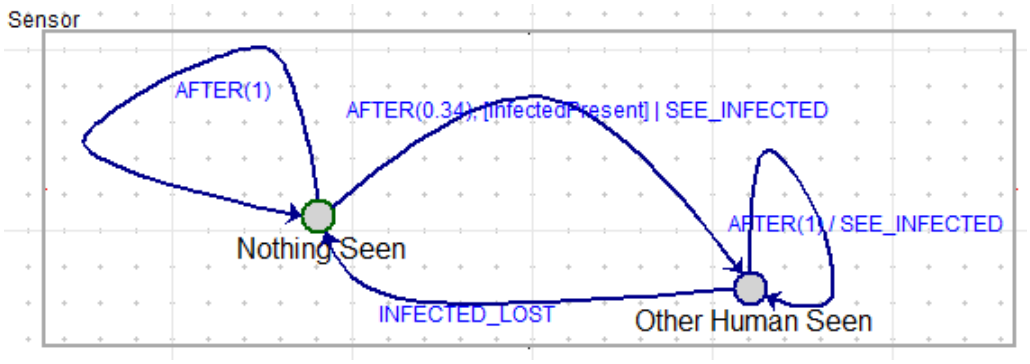

Figure 2: The statechart representing the sensor.

A sensor in this case represents the eyes of a human. Sick or not, a human uses its eyes to look around and spot things. This can be a wall, a healthy human or a sick human. Depending on what is seen, an event is generated.

2.2. Memory

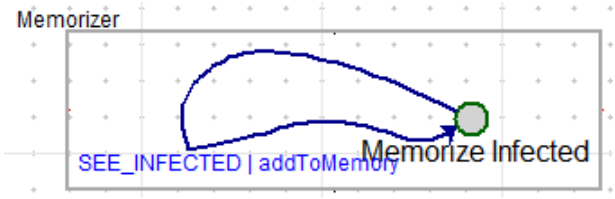

Figure 3: The statechart representing the memory.

Once something is spotted, this is remembered and can be used in later stages, such as for pathfinding.

2.3. Strategic Decider

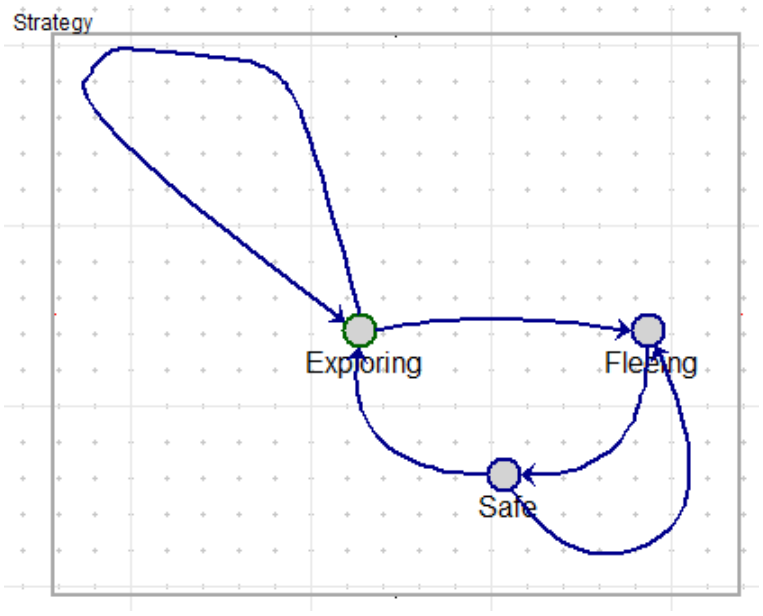

Figure 4: The statechart deciding the strategy to follow.

The general strategy is decided by the strategic decider using information gathered by the sensors and the current state of the human. A human starts in the exploring state. This means that its just walking around randomly. What happens next depends whether the human is sick or not. If its not, once a sick human is spotted, it goes into the fleeing state, it will try to run away. If left alone for a while, it will go back to exploring.

A sick person on the other hand will go to the attacking state if a healthy human is spotted. This just means that he will try to get as close as possible to a healthy human, for example to look for help. If it looses track of a human, he will go back to exploring.

2.4. Tactical Decider

The tactical decider specifies how a specific strategy will be performed. So for every strategy present in the strategic decider we need a corresponding statechart in the tactical decider. In our case this means we need one for

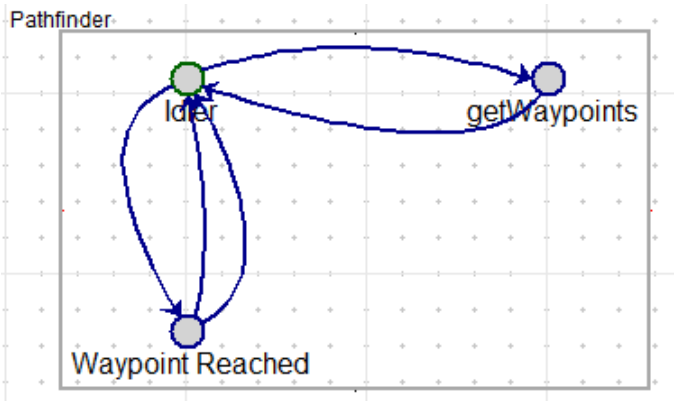

Figure 5: The pathfinder.

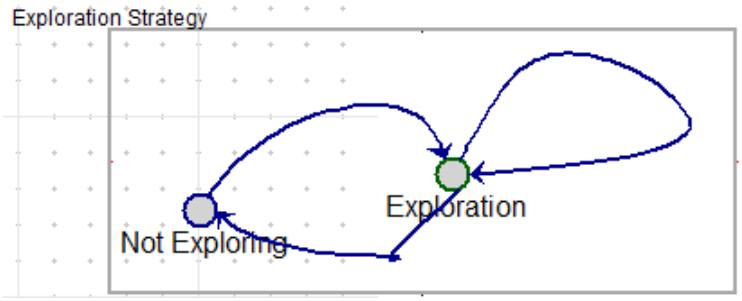

Figure 6: Exploration Strategy.

exploring, fleeing and attacking. The exploring state is the same for healthy and sick humans. A destination is randomly choosen, the human walks to that location. Fleeing is unique for healthy humans. A location as far away as possible from all the sick people it remembers and sees is chosen in a certain radius around the human. The pathfinder translates this destination into smaller waypoints. If new sick people are spotted, this location is adjusted. Attacking on the other hand is unique for sick people. They choose the human closest by and look for the shortest path to that human. If a human is encountered that is closer, the destination is changed. A pathfinding component is used to translate the chosen destination to waypoints.

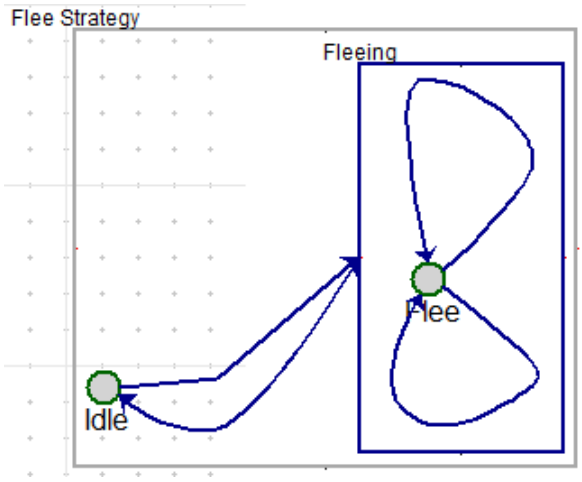

Figure 7: Flee Strategy.

2.5. Executor

Executors translate the events from the tactical decider to events the actuators understand. When it recieves a new waypoint, it will tell the actuators to start, if a destination or waypoint is reached it will tell the actuators to stop and wait for a new waypoint or destination.

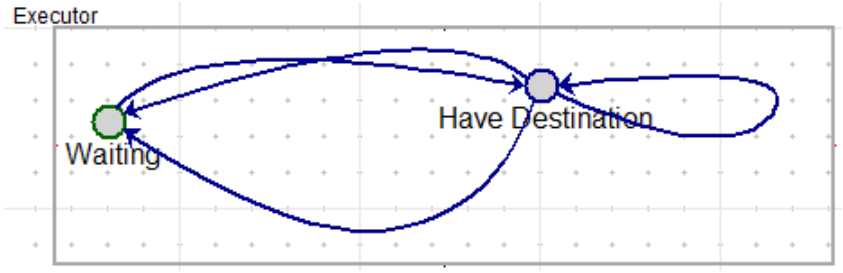

Figure 8: Executing the strategy

2.6. Actuator

The actuator represent the legs and feet of a human. It starts and stops the human if needed, depending on the events it recieves.

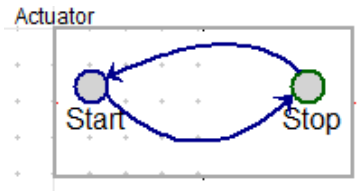

Figure 9: The actuator

2.7. General

Independent of the state a sick person is in, after a certain amount of seconds, it dies. This is accomplished by combining all orthogonal modules mentioned above in one general component. After a certain amount of seconds, it goes into the dead state and disappears out of the world. The same technique is used to model the transition from a healthy person to a sick person.

3. From statecharts to a game

To get from the statecharts specified above to a working game, I made use of the framework provided by Gino Wuytjens that he is developing in function of his master's thesis. SCC, a stachart compiler is used to compile the statecharts into python code. This code is then attached to an actor representing a certain class of objects. In this case there was an actor for a healthy human and one for a sick human. The statecharts themselves are made in atom3

Every pass of the game loop, that is every 33 milliseconds, the statecharts are updated and all events propagated and executed. This means that the statechart must be finished in this time frame. If not, the game will wait for the statechart to be finished which will give the impression that the game hangs. This was not encountered during the development of this encounter though. While the events are propagated, the necessary code is executed. When the statecharts are finished, the sprites are updated and the loop starts anew.

One minor downside of this approach is that a statechart detects certain events slightly later than actually happening. For example detecting if a

waypoint is reached. The statechart only discovers that a waypoint is reached 2 or 3 passes of the main loop later then when this actually happens. Ofcourse the object keeps moving during these loops, so by the time the arrival at the waypoint should be detected, it might have passed it already and this might go unnoticed. This is not a big problem, but one does need to take this into account and, if necessary, take appropriate steps to counter this or it might cause bugs that are hard to find.

3.1. Possible improvements

A lot of possible improvements can be made. First and foremost, a refactoring of the code is needed. It does not adhere to appropriate coding and design standards. If extensions are to be made, the code needs to be cleaned first. On top of this, there is still too much logic present in code. There are some aspects that can and should be specified in statecharts. The only code present should be a general game engine and the algorithms needed by the specifications made in the statecharts.

I made the mistake of starting in code and designing the statecharts later, so, subconsciously I was adapting my statecharts to my code and it should be the other way around. It would have prevented bugs and made development time shorter.

It is also easily possible to add new behaviour to the game. For example a soldier class, that hunts down sick people, or a doctor class, that heals them. Or the simulation can be made more realistic by giving the healthy humans a certain amount of stamina, so they need to rest once in a while. These additions only need new statecharts on the strategic and tactical level.

4. Conclusion

Using statecharts to specify the behaviour of NPC's works very well. From section 2 it is obvious that components are very reusable. In our case, a healthy and a sick human are the same, except for the strategic and tactical decision. This means that every part can be left alone, except for those two. If for example we would make some kind of soldier, we could use the attacking strategy from the sick person.

Some behaviour is actually simpler to implement using statecharts than code. Using statecharts, we can, on a high level describe what actions need to be taken to achieve a certain goal. A programmer than just needs to implement specific algorithms, for example: in a statechart we state that we want the shortest path, for this, the A* algorithm is implemented in code. This greatly reduces the complexity of the needed structures in code and prevents bugs when certain components need to work together.

Another advantage is that non-programmers can use this technique. Once again we use the pathfinding example. A game designer may not know the algorithm to calculate the shortest path from one point to another. So he can just say, in this state we need the shortest path.

To conclude, I had the same positive experience using statecharts as in Kienzle et al. (2007) and it seems they are really helpful in this kind of context.

5. Bibliography

Kienzle, J., Denault, A., Vangheluwe, H., 2007. Model-based design of computer-controlled game character behavior. In: Engels, G., Opdyke, B., Schmidt, D., Weil, F. (Eds.), Model Driven Engineering Languages and Systems. Vol. 4735 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 650–665.