Modelling The Response Of Cars Travelling Along A Straight Road To An Emergency Vehicle

Abstract

The purpose of this empirical modelling study is to model the scenario of three cars travelling along a straight road and responding to an emergency vehicle, considering not only the model itself but also the process of modelling undertaken and how this could be improved in the future. It will firstly be described how the modelling process was undertaken. We will then discuss the problems of specifying the model in the LSD notation and will suggest solutions to these problems. Finally we will consider the difficulties encountered creating the working model in the Eden environment and possible modifications to the modelling environment which could remove these difficulties in the future. It will specifically be considered how modelling multi-agent systems such as the one created in the model alongside this paper could be facilitated in the future both in by updating the specification methodology and the working modelling environment.

1 How the scenario was modelled

This modelling exercise began by defining the three main agent types in an LSD specification. The road, car and emergency vehicle agents and their respective observables and inter-dependencies were defined using the relevant oracles, states, handles, derivatives and protocols.

The next stage was to create a graphical environment in which the model could be represented and manipulated visually, a task which was undertaken using a series of Scout windows.

Following this, the agents were defined as graphical objects using the Donald notation. The basic properties (states) of the agents were defined as part of this notation.

Once there was a graphical modelling environment to work with, the inter-dependencies between agents were described in Eden, keeping closely to the dependencies specified in the LSD specification. At this stage, a clock was developed to allow agent animation, though this had not been pre-specified using LSD. The processes involved in each of these modelling stages will now be discussed in some more detail.

1.1 Creating the LSD Specification

1.1.1 The road agent

This simple agent is used to describe states which vehicle agents will have as handles, i.e. the purpose of the road agent is to define global variables which both cars and the emergency vehicle will be affected by. These include the dimensions of the road, the speed limit, the horizontal speed at which vehicles should move away and the distance at which cars should slow to match the speed of cars in front.

1.1.2 The car agent

One car agent was described in LSD, though three copies of this were later implemented in Eden. The states and handles of this agent store basic information about the car including its location, dimensions, speed, which lane of the road it is in and the distance between it and other vehicles.

Oracles are required from all other agents. From the road agent, the dimensions, slowing distance and range at which the emergency vehicles' lights can be seen is required. From other vehicles (i.e. both other cars and the emergency vehicle), their speed, dimensions, location and lane must be known in order to determine their location in comparison to the car in question. Specifically from the emergency vehicle, the on/off status of its lights is required.

Derivatives include whether or not cars are currently alert to the emergency vehicle, (only true in a state of emergency when the emergency vehicle's lights are on), as well as the central position of cars in each lane of the road.

Protocols for the car describe mainly the cars' relationships to other cars and the emergency vehicle. The distance between this car and other vehicles must be known. If this car is within the slowing distance behind another vehicle it must slow to match the speed of the vehicle in front. If the car is alert to the emergency vehicle, it must move away in the right direction to allow the emergency vehicle to overtake and must then move back again once the emergency vehicle has passed.

1.1.3 The emergency vehicle agent

The emergency vehicle has the same states and handles as the car (except for the moving away direction). In addition to these, the state of the lights and whether it is stuck behind a car is stored.

Oracles are also similar to those of the car agent, such as the location and dimensions of other vehicles. In addition to these the speed limit is required (from the road agent), as well as whether any car is currently alert to the emergency vehicle. Derivatives define the central position in each lane or the road, as with the car agent.

Some protocols for the emergency vehicle are the same as for the car agent, such as defining when it is necessary to slow behind a car. If the emergency vehicle must slow to avoid crashing in to a car, it records the fact that it is stuck. When any car is alert to the emergency vehicle and so pulls away to make room for it to overtake, the emergency vehicle must also move over so it can pass. It must move in the right direction depending on which lane it is in, and must move back once it has passed the car. Finally, if the emergency vehicle is not stuck behind a car, its speed should be set to the speed limit.

1.2 Creating a modelling environment using Scout and Donald

Before the dependencies described in the LSD protocols for the car and emergency vehicle could be specified, the agents themselves and their states needed to be defined so they could be displayed graphically.

The first stage of achieving this was to create a windowed environment using Scout notation. A large window to display the road was defined, as well as buttons to give the user control over some of the vehicles' states and the clock.

Following this, the Donald notation was used to describe the road and car agents visually. The road was created to fill the road window, with a line down the centre to separate the two lanes. A car was then defined. In addition to the basic states described in the LSD, other information was added to make the car slightly more realistic, such as a line marking the bonnet and boot as well as wheels. The completed car was then copied twice to make three car agents in total and each car was differentiated by changing its body colour.

At this stage interdependencies between cars were defined (described in section 1.3) before the emergency vehicle was created.

The emergency vehicle was initially modelled by making a fourth car agent with no differences to other cars. A light was then added and the boot taken away to differentiate its appearance.

Finally, the Scout buttons were made functional, so they could move vehicles between left and right lanes, and change variables to store whether the clock was started and the lights of the emergency vehicle switched on.

1.3 Modelling inter-agent dependencies in Eden

Dependencies between vehicles in general and specifically between cars and the emergency vehicle were described in Eden, based on the LSD protocols for the car and emergency vehicle agents.

Because in practice there are three car agents, each of the protocols for the car agent had to be repeated multiple times for each car. Some of the protocols referred to other vehicles in general so also had to be repeated to take the emergency vehicle into consideration. As an example, in the case of calculation of the distance between vehicles, this had to be repeated a total of twelve times, as there were four vehicles which each having to relate to the other three.

Before the protocols for inter-vehicle interaction were defined, a clock was described in Eden to allow agent animation. This clock could be started and stopped by the user via a Scout button.

Modelling the vehicle inter-dependencies was not a simple task of entering the logical syntax of a particular protocol once. Rather, an idea was input in to Eden for how to model a dependency such as when cars should be alert to the emergency vehicle. The result of this was then viewed in practise, which at first was never quite as expected. A process followed of updating the code for a dependency and then testing it and re-modifying it until the desired responses could be clearly observed in the graphical model simulation.

2 Difficulties in describing a multi-agent model in LSD

In describing the multi-agent system in LSD as described in section 1.1, there were three main difficulties which had to be overcome.

The first of these difficulties arose from the fact that three identical car agents needed to be defined, each of which had relationships to each other. One option in defining them would have been to copy the same LSD agent three times and rename variables to reflect their ownership to a particular car.

However it was felt that a more appropriate way would be to 'cheat' by separating implementation from instantiation. In other words, it was decided that a generic car would be defined, with variables owned either by (this) car or by an (other) car. A dot notation such as 'this.width' or 'other.width' was used to show whether a width was owned by this vehicle or another one.

Another problem was in showing ownership of a particular variable to a particular type of agent. In some cases different agents had states of the same name (such as the width of a car or of the road). It was thus necessary to show which agent was being referred to in a statement involving states of several agents. Therefore, the dot notation was also used in the form 'road.width' (rather than this.width) to show the width was owned by the road (rather than by this vehicle).

The final problem was in defining the emergency vehicle. The problem arose from the fact that a large amount of the emergency vehicle's properties were identical to those of the car because they were both road vehicles. In practise, the emergency vehicle was defined separately from the car with similar properties being repeated. However in any specification (or implementation) repetition gives more room for error and redundancy. The ideal situation would have been for the emergency vehicle to automatically inherit specified properties of the car agent, but this is not possible in LSD. A suggested solution is discussed in section 3.2.

3 Suggested improvements to LSD to aid the specification of multi-agent systems

3.1 Ability to specify the ownership of variables in LSD

The first two problems described in Section 2 are of the inability in standard LSD to specify ownership of a variable to either a particular agent type, or to a particular instantiation of an agent type. The LSD specification for the car, emergency vehicle and road model cheated by introducing a dot notation to specify this ownership. Perhaps such a notation could be incorporated in to standard LSD practise as a solution to this problem. However the dot notation used in this car/road model would need to be more fully defined to be used in practise for real world empirical modelling specifications.

If the idea of separating a generic agent definition to a reference to a specific instantiation of that agent is to be taken further, (as in the class / object idea of object-orientated programming), it would be necessary to be able to create definitions and instantiations separately in LSD notation. The current inability to do this, corresponds to the same inability in Eden. It might be counterproductive to increase the conceptual separation between LSD and Eden, and so Eden would need a similar modification to the way agents are specified to be worthwhile. See Section 5.1 for a discussion of how this could be accomplished.

Once this separation would be implemented, oracles could then be defined either to belong to an agent type or to an agent instantiation. This idea creates further problems in how to show precisely what variables logical statements are referring to, and this is addressed in Section 3.3 below.

3.2 Ability to allow one agent in LSD to inherit the properties of another

The final problem discussed in Section 2 was of the inability for one agent to inherit specific properties of another, causing the necessity for identical properties to be repeated for each agent in which they are defined.

The idea of inheritance comes from object inheritance in object-orientated programming. Objects have a parent, from whom they inherit all the data and methods, and can have children who inherit all of their properties, as well as the properties of each object higher up in the hierarchy.

It could be possible to implement some form of inheritance in agent orientated programming. Thus in the example of the model described in Section 1, a parent agent called a vehicle could be defined with properties owned by all vehicles such as width, length, left, bottom, speed and lane. Two subagents could then be defined called car and emergency vehicle which would inherit all the properties of the vehicle agent.

This would mean that states, protocols or other properties would not need to be repeated for all agents which are similar in some way. For example the distance between one vehicle and another could be defined in the vehicle agent rather than in its children. In the car agent, only properties specific to cars would be defined, such as whether they are currently alert to an emergency vehicle, or whether they need to pull away to the side of the road. Similarly the emergency vehicle agent would only need to describe properties relevant to itself, such as the status of its lights, or whether it needs to adjust its speed to overtake a car.

3.3 Ability to use First Order Predicate Logic in LSD

The solution discussed in Section 3.1, of defining both a generic agent and instantiations of it creates other problems in how logical statements can be expressed. Introducing the possibility of lots of versions of an agent means that it needs to be possible to refer in logic to either a specific, more than one of, or all such agents. Thus propositional logic, which can only refer to specifically named variables, is unable to meet these requirements.

A sensible solution to this problem would be to introduce the use of First Order Predicate Logic into LSD specifications (particularly into the derivatives and protocols of agent definitions). This would allow either universal of existential reference to instances of a particular type of agent allowing greater flexibility in what can be described.

As an example of a statement requiring this, let us consider the model road system described in Section 1. A statement might want to be made as a protocol to the emergency vehicle such as: ‘if there is any car that is alert then move to overtake it’. The notion of ‘any’ could not be made with propositional logic alone, but instead a reference would have to be made to each and every car agent, connected with ‘ands’ to specify universality, (if all cars…), or ‘ors’ to specify existentiality, (if a car exists such that…). The quantifiers of First Order Predicate Logic would solve this problem, but would need a way to be implemented in a modelling environment such as Eden, (discussed in Section 5).

4 Difficulties in describing a multi-agent model in Eden

The difficulties that were experienced in modelling the multi-agent car / emergency vehicle / road system using Eden were the same in nature as those that were found in initially describing the model in LSD. Namely, the difficulties were found of specifying multiple versions of the same agent type, and of finding a way to implement agent inheritance.

The other problem of specifying ownership was partly solved by the Donald system of being able to define data items within a ‘shape’ item. Thus a notation was used such as ‘_car1_height’ and ‘_car2_height’ to specify whether the height in question was owner by car 1 or car 2 respectively. In cases where agent specific variables were created in Eden rather than Donald, the same notation was used without the initial underscore.

5 Suggested improvements to Eden to aid the implementation of multi-agent systems

5.1 Ability to create multiple instances of a single agent type

In the current implementation of the Eden environment, each agent is both a type of agent and an instance of an agent. Each agent is unique and contains both descriptions of its properties and its current state.

Unlike in LSD, the concept of an agent in Eden is a very loose one. There is no pre-defined data type called ‘agent’. Rather an agent is a collection of variables which the modeller has chosen to create and associate with one another.

Others have discussed whether the Eden system should be strongly or weakly typed. However what is being suggested here is more of a ‘construct’ than a type; the notion of being able to collect all the definitions for a type of agent into one construct which can itself be referenced. For example, a file could be created called ‘car’ which defines in full a generic description or a car agent. Another place will then be needed to create and manipulate specific instantiations of these ‘agent types’. In this case, three cars would be created of a car type and related to an instance of a road type.

This idea raises many questions which this paper does not have enough scope to consider. These questions would need to be answered before an implementation of ‘agent typing’ could be considered. One question would be to consider the effect on the versatility of empirical modelling tools if agent typing is enforced. Another would be in choosing a notation to refer either to a generic ‘agent type’ or an instance of it. The way logical statements are created would have to be extended to include quantifiers, as discussed in relation to LSD in Section 3.3.

5.2 Agent inheritance in Eden

As described in Section 3.2, it would be useful to be able to extend an agent type so that a ‘sub-agent’ could inherit its properties. This is assuming the idea of an ‘agent type’ from Section 5.1 would be implemented in Eden. Syntax to specify agent extension would be required, as would a set of rules to define precisely what is allowed and what is not, such as multiple inheritance or circular definitions where an agent tries to become a descendant of one of its own descendants.

6 Conclusion

The solutions described above are in essence a union of Object Orientated and Agent Orientated Programming. Specifying ownership, multiple instances and inheritance are all problems solved in an Object Oriented framework. Considering the difficulties and suggested solutions discussed for this modelling task, perhaps an amalgamation of the two paradigms would serve to bring them closer. The challenge would be to maintain the ability to define inter-agent dependencies and the flexibility of the modelling environment.

Acknowledgements

N/A

References

N/A