DOI: 10.15439/2014F335
ACSIS, Vol. 2

Opening Pandora's Box: Some Insight into the Inner Workings of an Agent-Based Simulation Environment

Daniel Dawson
School of Computer Science,
University of Nottingham,
Nottingham, UK
Email: ddawson4417@gmail.com

Peer-Olaf Siebers
School of Computer Science,
University of Nottingham,
Nottingham, UK
Email: pos@cs.nott.ac.uk

Tuong Manh Vu
School of Computer Science,
University of Nottingham,
Nottingham, UK
Email:
psxtmvu@nottingham.ac.uk

Abstract— **Agent-Based Simulation (ABS) environments are somewhat of a black box to many modelers in Social Simulation or Economics and their inner workings are often only understood by the computer scientists who developed them. We intend to shed some light into the inner workings of such systems. For this purpose we have developed our own simple ABS environment in C++ using hierarchical state machines. In this paper we provide insight into the design of our ABS environment and then test the performance of it by comparing it to that of an "off the shelf" commercial package. While some programming knowledge is required to understand the paper in all its depth we believe that non programming experts will also benefit from this paper as it provides an insight into the underlying mechanisms operating within an ABS using graphical representations and explanations that avoid heavy technical jargon.**

I. Introduction

An Agent-Based Simulation (ABS) environment is a system in which a population of agents (autonomous objects that behave in a predefined way) are created using a template in order to investigate the consequences of these agents acting together in an environment. The application area that we focus on in this paper is Social Simulation [1] and we use examples from computer games (which are related to Social Simulation but much easier to understand) to describe the agents we created during this investigation. The very simple experiments we conduct help us to understand situations which can otherwise be difficult to replicate. ABS experiments can sometimes yield unexpected results, for example an ABS constructed by Bonabeau [2] revealed that placing a column in front of an emergency exit can improve the flow of people out of the exit in an emergency situation, which is not the first idea which common sense would dictate.

This paper will describe and promote the understanding of the inner workings of an ABS environment that has been developed from the ground up. For the software engineering process (i.e. the development of the ABS environment) we take our ideas from the Multi-Agent Systems field. The models we implement during the validation phase are those typically created in the Agent-Based Modelling community (e.g. by Social Scientists). A good explanation about the relevant differences between both fields can be found in [3].

In order to explore how an ABS system works, and subsequently construct one, it is necessary to first explore the concepts that are involved in creating such a system. Simple ABS systems are generally implemented using finite state machines. Once the behaviour of agents gets more complex the introduction of hierarchical state machines becomes necessary to avoid the over-complication of finite state machines, leading to state machines that can be notoriously difficult to fully understand. Object-oriented design principles will be used in the construction of our tool in order to promote its extensibility, allowing anyone to add features or extend classes where it will benefit them. The Unified Modelling Language (UML) is a graphical notation that is often employed in Software Engineering for conducting object oriented analysis and design. AgentUML is an extension of UML that is specifically used for the development of multi-agent systems [4], as for example mobile agents [5], [6]. However, in the field of Social Simulation it is still rarely used for developing agent-based simulation models that represent social processes [7]. In this paper we use the UML on the one hand to show the structure of the proposed ABS environment (in form of a class diagram) and we use statecharts on the other hand for the design of our agent based models (i.e. to represent the behaviour of our agents) as proposed by [8].

In order to provide the reader with the necessary understanding of all of the topics within this paper we first provide a short introduction to object-oriented methods, agent-based modelling and the concepts of state machines. After this we illustrate the design of the ABS system which we have implemented. Finally we focus on the validation of our system in order to demonstrate that its components adhere to the standards of an ABS system. During the validation process we also take a look at the efficiency of our system compared to an "off the shelf" commercial package in terms of memory usage and runtime.

II. Background

A. Object Oriented Methods

Object-orientation is an important concept for designing software in order to promote extendibility of existing systems. The object model encompasses the core principles of abstraction, encapsulation, modularity, and hierarchy [9].

It leads to reusable components, wherever possible, rather than the more bespoke solutions which procedural programming often offers. It also breaks programs down into understandable chunks, and by designing software with object-oriented methods in mind it can be more easily extended and fixed when problems arise.

In order to promote object oriented principles in our design we have taken a number of design patterns into consideration while designing our system. A design pattern is a standardized way of implementing a certain feature in a program [10] and makes it easier to reuse successful designs and architectures [11]. This will be discussed in more detail in Section III-A.

UML is a graphical notation commonly used in software engineering for the purpose of object oriented analysis and design. Through UML it is possible to visualise, specify, construct, and document software applications. It acts as a specification language in which we can precisely and unambiguously capture our design decision [12]. Besides the benefits for software design some of the diagrams (e.g. use case diagrams and state machine diagrams) seem to lend themselves particularly well to Agent-Based Modelling (ABM) [13]. Therefore we use the UML notation not only for designing our ABS environment but also for modelling the agents that we use within our system.

B. Agent-Based Modelling

In order to get a good picture of what an ABS environment is, we first need to define what the term agent means, including the principles on which an agent acts and behaves. In the eyes of software engineers agents are simply "objects with attitude" [14] in the sense of them being objects with some additional behaviour added, for instance, mobility, inference, etc. But there are a number of different conflicting views on what an agent is, depending on the situation and discipline for which it is being used [15]. However, often there is a point where the views start to overlap with each other. Castle and Crooks [16] discuss different points of view and merge them together to form a universal definition of an agent, which varying disciplines can agree with. Closely related to their definition we understand an agent to be an autonomous object with some memory, which is able to make individual decisions based on influences from its environment (e.g. messages received from other agents).

The agents which we intend to create have the ability to make decisions based on internal transition trigger rules which might be influenced by the environment they observe. These transition trigger rules which have been programmed into the state machine of the agent and most often fall into one of three categories: condition-based, time-based, or message-based [17]. Details about different transition types can be found in Section III-C.

An agent is often described as having some sort of memory, which comprises of the last state they were in or in the case of composite states, the last super or sub-state they were in. This is the concept of state history, and there are two types: Deep history and shallow history. Deep history goes through multiple levels of composite state, and will return to the last state within a state within a state etc. Shallow history will only return the last state to within a state.

The last major thing to consider with agents is a form of control. As mentioned previously, finite state machines are often used in the creation of agents, as a means to describe the behaviour of an agent, or in other words, a template for how they should act, with conditions specifying when a transition should be made. This leads us onto the topic of Finite State Machines (FSMs).

C. Finite State Machines

A Finite-State Machine (FSM) is conceived as an abstract machine that can be in one of a finite number of states. It can change from one state to another when initiated by a triggering event or condition; this is called a transition [18]. There are different types of FSMs that can be used in a variety of different situations. We distinguish in this paper specifically between deterministic and stochastic FSMs. While deterministic FSMs are based on mathematical formulas and can be formally proven, stochastic FSMs use stochastic rules for deciding about transitions and therefore the exact outcome cannot be predicted; however it may be estimated using models and theories. A turnstile is a good example of a simple deterministic FSM. It can either be locked or unlocked, and there are predefined conditions that determine which state the turnstile is in, and the transitions it can take from each state. Fig. 1 demonstrates this.

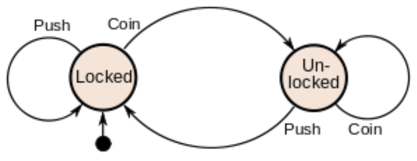


Fig. 1 A simple deterministic FSM (Source: [19])

More complex state machines are often used for economic models [20] and the transition function, which defines the transitions between states, is a lot more complex. For ABM we normally use stochastic FSMs. When behavioural models start to be implemented, FSMs can quickly become complex without some sort of organisation. In such cases Hierarchical State Machines (HSMs) are be introduced in order to keep such a model understandable. In HSMs states may contain other FSMs. This is often programmatically done with nested "if" statements.

D. Hierarchical State Machines

A HSM is similar to the composite state we see in UML state diagrams, and it provides the same functionality. One of the simplest ways to describe a HSM is by showing an example of one of its uses in Game AI design. Non Player Characters (NPCs) must have the ability to act in a complex way in order to give the player a challenge. If all of this behaviour were to be coded with a single if-then-else block, the code would quickly become hard both to manage, and for programmers to read and extend, specifically where the more recent generation of game AI is concerned. Fig. 2

shows the first level of a HSM of a typical NPC enemy in a typical first person shooter game.

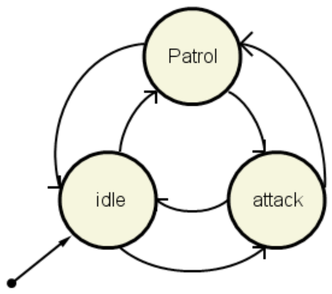


Fig. 2 Typical FSM for an NPC guard in a game

The HSM provides a structural concept for representing behaviour, which can otherwise be complex and resulting in somewhat of "Spaghetti" code when trying to split into logical if statements, in order to structure the functionality into logical categories of behaviour. Fig. 3 shows the hierarchical state "attack" from Fig. 2.

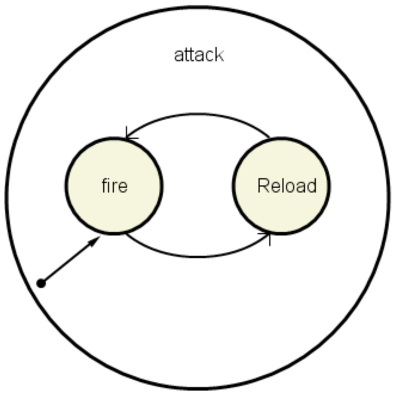


Fig. 3 Hierarchical attack state from NPC guard

Reference [21] talks about an object-oriented software tool that is used for creating the behaviour for NPCs in games by making use of HSMs. The system uses the logical components of a state machine, as well as making use of object-oriented principles such as inheritance and design patterns. This is similar to the work we are conducting, whereby state machines are being used as a template for the behaviour of entities. In our tool, more focus will be placed on the collaboration and communication of agents, unlike NPCs which often have no need to communicate or interact with each other.

E. UML State Machine Diagrams

The UML state machine diagram (also called statechart) is used to depict HSMs. Elements of this diagram are states, transitions, and composite states (which are equivalent to hierarchical states). Fig. 4 shows a UML state machine diagram of an office worker.

The office worker has three main states: "atHome", "elseWhere" and "atOffice". The "atOffice" state shows that while the worker may be at the office, there are still two sub-states that the worker can be in - "working" and "dozing". The statechart entry (the initial state the state machine is initialized into) is represented by the uppermost symbol in

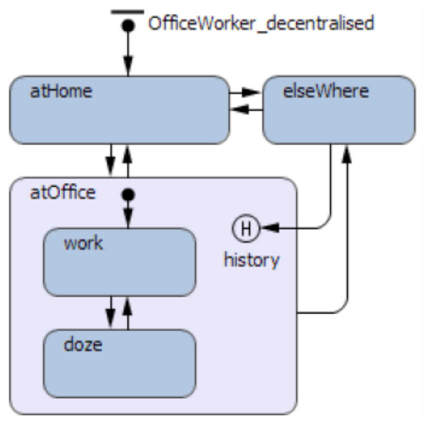


Fig. 4 UML diagram for an office worker (Source: [13])

the state-chart, the circle with an arrow with a line over it. The history state in this example can be described using the following scenario: If a worker was doing work, but then decided to take a break and go elsewhere, they would return to what they were doing before the break. The history state provides this capability. However, if the worker was dozing and then took a break, they would not start at the entry state of working when returning, but instead would return to the dozing state.

III. DESIGN

The design of the finite state machines that run the agents is based on the logical components of a state machine, as is the case in UML, where state machines have states, transitions, and composite states (state machines within states). The agent of this system contains the information relevant to the state machine, which includes the current state, last known state and history states for the super-state it was last in. This is a more memory-efficient way of storing the information, rather than having each agent assigned its own state machine, although the logic behind this is explained later.

A. Design Description

This section will describe the design of the system, including a simple class diagram of the main components of the agent. For the implementation we have decided to use C++ which in some cases has influenced our design decisions (e.g. multiple inheritance is not supported in Java). Fig. 5 shows the classes and initial relationships between each of the classes in our system.

The numbers in Fig. 5 represent the associations between objects, with * representing any number of. For example, the relationship between `Agent` and `StateMachine` is that there is one `StateMachine` to one `Agent`, and the relationship between `StateMachine` and `State` is that one `StateMachine` is composed of one or more `State` objects. The hollow arrowhead represents inheritance, showing that `CompositeState` inherits properties from both a `State` and a `StateMachine`. The `Attributes` of each object are the variables stored within the object, and the `Operations` are a list of methods which are used

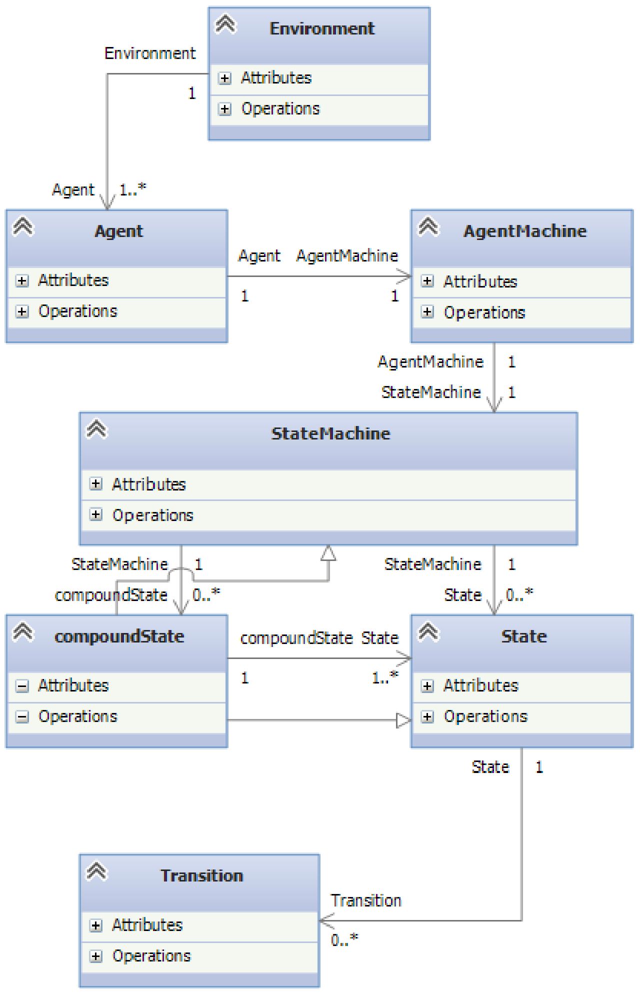


Fig. 5 UML class diagram for our ABS environment

within the object. As such, the object can be accessed as either a State or a StateMachine. The full class diagram including the methods that are used internally is available upon request.

The way the system is designed ensures object-oriented principles are taken into account so as any one of the objects in the system can be extended. The CompositeState class makes use of inheritance to reuse the code from StateMachine and State, since it exhibits the behaviours of both classes. A number of design patterns such as the Observer pattern, the State pattern, and the Model-View-Controller pattern have been taken into consideration during the design process. For example, the Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [10]. In our case the Agent class is constructed using the ideas of the observer design pattern, allowing agents to constantly monitor transition conditions and allow them to enact the transitions themselves. This removes the need for the transition conditions to be explicitly checked.

In the following sub-sections we provide some more detailed design strategies for the key elements of our ABS environment. Here we look at the environment and agent design, the state machine design and explain the different time models we used.

B. Environment and Agent Design

The Environment in our ABS environment is a container for the agents, which manages their creation and deletion. This class also handles the sending of messages intended for all agents rather than singular agents. The agent class is designed to hold and handle the state machine for the instance of the agent, including telling the state machine to advance a time step in model time. There are two possible ways of controlling the state machine which links to an agent. The first (and simpler) way is to assign a state machine to each agent. The second (more complex) way, proposed by [21], is to use a simplified representation of a state machine without having to create a full machine for each agent; a way which can save a lot of memory. In the latter case the information for the current state of the agent is stored in the agent itself, whilst the state machine stores all of the logic for the states and transitions. The reason for storing information this way, rather than having a machine for each agent, is that even for simple state machines, a high number of objects will take up a lot of memory regardless of how small these objects are. Due to the object-oriented nature of this design, each state machine object requires the creation of every state and transition present in that state machine. The number of objects soon becomes very large, and becomes somewhat of a waste of memory, albeit at the expense of slower processing of state changes. However if a small number of agents were being created or the user were not concerned about memory usage, the gain in state change speed may be preferable.

Let us illustrate this principle with an example. Presume a state machine comprises of a total of 10 states and 10 transitions, and each object takes 1 byte of memory. In order to create 100,000 agents, 1 state machine + 10 states + 10 transitions + 1 agent object will be created for each agent. This totals 22 * 100,000 objects, so 2,200,000 objects in memory. If there is only 1 state machine acting as a template, there are 100,000 agent objects, plus 1 state machine, 10 states and 10 transitions, totalling 100,022 objects in memory. Of course, the larger the state machine is, the more effect it has on the size of the agent. Since we are trying to save memory here, the choice is only logical. We will provide evidence for the memory saving capabilities of this solution in Section IV when we validate our ABS environment.

C. State Machine Design

A state machine has logical components to it, which makes it easy to split up into the objects we talk about in object-oriented programming. The typical state machine is fairly simple and composed of a set of states, and a set of transitions. Composite states however, whilst being regarded as a state within the agent state machine, effectively contain their own state machine. This can be recursive, and there can be many composite states within another state. Fig. 6 shows a UML diagram similar to the NPC in Fig. 2 but more complex.

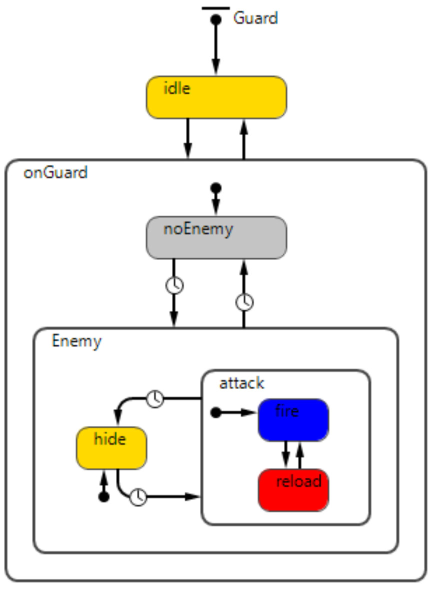


Fig. 6 UML version of a more complex NPC Guard

What we can see in Fig. 6 is that within each composite state there is an initial state pointer. This indicates that a composite state effectively contains a state machine, and so exhibits properties of both, a state machine and a normal state. Therefore it makes sense to reuse both, the state and state machine object, through use of multiple inheritance. Inheriting properties from multiple objects can be a complex problem in object-oriented programming [22]. However, in our case it seems an appropriate solution to the `CompositeState` problem.

Transitions within a state machine have certain conditions which cause the transition to trigger, and in the case of ABM the transitions can be triggered by a number of different things. We have considered the four following triggers which are often associated with transitions: condition; message arrival; rate; timeout. These are quite basic transitions which will cover transitions most people need to implement, however if any other types are needed, the transition class can be extended appropriately.

Condition-based transitions can be specified by the programmer when a certain condition is met. Since this condition could be any number of things (a block of code which can return true or false), there are two feasible ways of implementing such a transition. The first method would be by introducing time steps to the model. This would be using a synchronous time model, and therefore perhaps not the best way to implement things which reflect the real world, in particular in Social Simulation. We as people are not expected to make decisions every arbitrary unit of time, we simply make decisions when they need to be made. Asynchronous time, which does not specify time steps and instead relies on agents doing things when a condition is met, would reflect the real world more accurately, and is also more computationally efficient. One way of implementing such a time model would be to use the observer pattern. This means that once a condition is needed for a transition, an observer will be added to monitor the condition, and when the condition is satisfied, the appropriate object will be notified. In our case, this will be the lightweight object which is used to represent the agent instance of our state machine.

Time-based transitions are based on either timeouts, or a rate at which agents move from one state to another. The method in which this transition can be implemented is similar to Boolean condition based; however an easier way of implementing such a transition in C++ would be simply to set a timer, which notifies the agent instance of a state machine upon expiry. This eliminates the need for an additional observer to be added to the system.

Message-based transitions are the ones which are triggered upon receiving a message. No particular observer is required for this, as messages should be processed upon arrival at an agent. When a message arrives, it will be stored by the agent and a transition from the current state will be triggered if that particular message matches the condition of the agent.

D. Time Models

Agents need to periodically make decisions, which are based on the transitions in the state machine. There are two main ways of doing this: using an asynchronous and synchronous time model. Both ways have their advantages and disadvantages depending on what type of model is being created.

Asynchronous time models are the most commonly used when trying to model real world situations where an agent acts of their own accord at random time intervals [23]. This reflects best what happens in the real world in social systems. It can also be said that typically asynchronous models are more efficient in terms of computational expense, due to the fact that things are only triggered when they need to be triggered, whereas the same model run in asynchronous time will trigger at every time step regardless of whether it is necessary or not.

Synchronous time models on the other hand are where "time steps" are defined, and each time step triggers an agent to perform an action. This action can directly trigger a transition, or it can signal to the agent that it is time to make a decision on whether to make a transition or not. When using synchronous time models, time steps can notify an agent based on probability, effectively imitating asynchronous time models.

In our system we have the choice of which time model to use. Obviously there are advantages and disadvantages for both and it really depends on the application, which time model should be used.

IV. Validation

Our ABS environment has been implemented in Visual Studio 12 using C++ as the programming language. The "off the shelf" commercial package we have chosen for our validation experiments is AnyLogic 7, which is a multi-paradigm eclipse based simulation IDE that supports graphical model design for all major simulation paradigms (including ABM) [24]. In order to assess run time and memory usage we run the AnyLogic 7 and the C++ implementation on their own with no other processes using significant RAM or CPU time. The experiments were run on the same system to ensure fairness.

For this paper we have conducted two validation experiments which test individual components of the agent, including simple behaviour and composite states. We used the synchronous time model for our tests due to the fact that only time-based conditions were included in the model. We have conducted further experiments using the asynchronous time model as well as all types of transition triggers but due to the limited space we cannot present them here. Results of those experiments are available on request.

A. *Overview of Validation Experiments*

We have constructed a number of experiments to test the individual components of our system. The purpose of these experiments is to verify that the models running in our ABS environment behave as expected and we also compare the performance (in terms of runtime and memory usage) of our system to that of AnyLogic 7.

With regards to creating the models both simulation systems have a very different approach: Our system requires some basic C++ programming while in AnyLogic you can use drag/drop to create UML charts and then simply set up the transition triggers. The AnyLogic model is then automatically translated into Java code and is ready to be executed from within the AnyLogic environment. For our system, in order to construct the state machines for the experiments, an empty class implementing `StateMachine` is created, first listing each of the states as variables, including composite states. The transitions are then defined, and finally all the states and transitions linked together by using an `addTransition` method.

B. *Validation: Experiment 1*

The first experiment focuses on testing the simple decision-making capabilities of an agent. The state machine used to control behaviour here consists of two states: `stateRed` and `stateBlue`. When created, each agent initially is in `stateBlue` and based on probability takes a transition that leaves it to a final state `stateRed`. Once in `stateRed` the agent will not change states any more. The UML statechart for the simple agent is provided in Fig. 7. The initial environment was run using 400 agents.

A counter was used in each instance to count the number of agents which are in each state initially, and after 5 time steps. Fig. 8 shows our system after 5 time steps (with 0 representing `stateBlue` and 1 representing `stateRed`), and Fig. 9 shows the counter variables for each state in AnyLogic.

It can be seen that there is a slight difference in the number of agents in each state, due to random number generation in each instance yielding different results. This shows that the agents in our system behave as expected. The

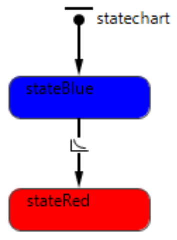
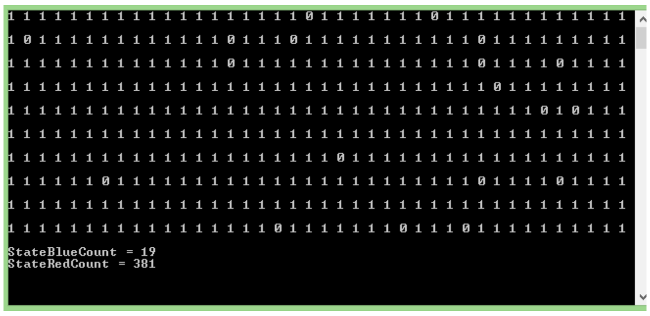
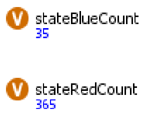


Fig. 7: UML state-chart for a simple state machine agent



```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1
StateBlueCount = 19
StateRedCount = 381
```

Fig. 8: Our system model after 5 time steps



```
stateBlueCount
35

stateRedCount
365
```

Fig. 9: AnyLogic model after 5 time steps

experiment was then altered to run in virtual mode in AnyLogic, which simply executes transitions as fast as possible, and was also run without any GUI in our system. The results in each instance are shown in Tables I and II. Since the time taken cannot be easily recorded in AnyLogic, the number of time steps executed in virtual time over 10 seconds was gathered, and averaged per second. The time steps per second on our model was calculated using the time taken for all of the transitions to complete.

TABLE I.
EXPERIMENT 1 RESULTS USING OUR SYSTEM

Agents	Ram	Timesteps
500	0.5 MB	1000 / sec
100,000	14.4 MB	2.92 / sec
250,000	34.1 MB	1.68 / sec
1,000,000	134.5 MB	0.38 / sec

TABLE II.
EXPERIMENT 1 RESULTS USING ANYLOGIC

Agents	Ram	Timesteps
500	90 MB	13063 / sec
100,000	180 MB	19.7 / sec
250,000	320 MB	6.4 / sec
1,000,000	1,260 MB	1.34 / sec

As can be seen in Table I, the number of time steps executed per second is linearly affected by the number of agents present in the model, and the memory used also grows fairly linearly with the amount of agents when concerned with a large agent population. Table II shows that AnyLogic also exhibits the same linear increase of memory usage with agent population size; however, this memory usage is far higher than our tool. The amount of time steps that is executed per second is also quite drastically affected in AnyLogic when a large number of agents are put into the simulation, although it is faster than with our tool. This could be due to the use of the lightweight state object which is used to save memory, and a performance increase may be obtained with the expense of more memory usage by assigning an individual state machine to each agent. It should be noted that AnyLogic can in fact execute multiple transitions per time step, so the number of transitions taken may be more than the time steps executed. This shows us that an improvement needs to be made to our tool in terms of the speed of the model, focusing on the use of the lightweight `AgentMachine` object used to save memory.

C. Validation: Experiment 2

This experiment implements the idea of composite state machines. In addition to the simple agent behaviour experiment, the `stateRed` state will now contain a clone of the previous state machine. The agent should move into the stateRed state and move between the states within this composite state. The UML statechart for the more complex agent is provided in Fig. 10.

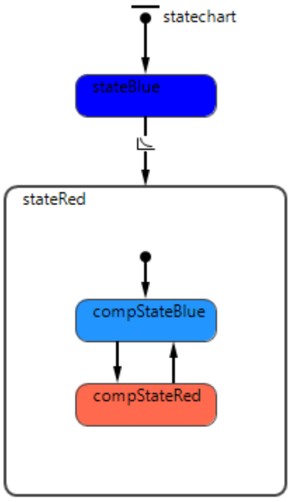
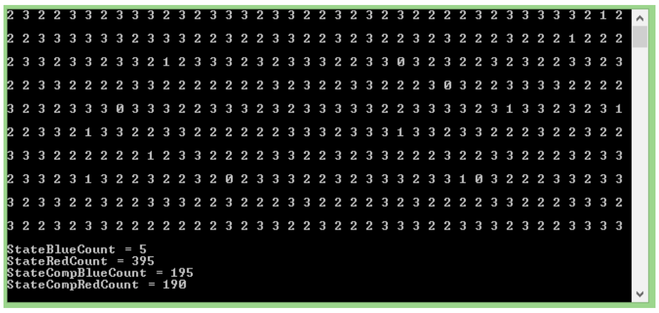


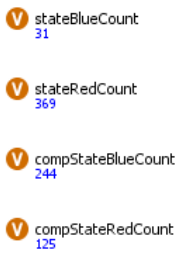
Fig. 10: UML state-chart with a composite state machine agent

As with the previous experiment, a counter was used in both AnyLogic and our tool to count the number of agents in each state. Fig. 11 and 12 illustrate the number of agents in each state after 5 time steps in both, our system and in AnyLogic. In Fig. 11, the numbers 0 and 1 represent `stateBlue` and `stateRed`, respectively, and the numbers 2 and 3 represent `compStateBlue` and `compStateRed`, respectively. As can be seen in Figs. 11 & 12, the number of agents in each state varies between our tool and AnyLogic due to random number being used to determine whether a transition should be made, however the agent behaves as expected with a composite state.



```
2 3 2 2 3 3 2 3 3 3 2 3 2 3 3 3 2 3 3 2 2 3 2 3 2 3 2 2 2 2 3 2 3 3 3 3 3 2 1 2

2 2 3 3 3 3 3 3 2 3 3 3 2 2 3 2 2 3 3 2 2 3 2 3 2 2 3 2 3 2 2 2 3 2 2 2 1 2 2 2

2 3 3 2 3 3 2 3 3 2 1 2 3 3 3 2 3 2 3 3 3 2 2 3 3 0 3 2 3 2 2 3 2 3 2 2 3 3 2 3

2 2 3 3 2 2 2 2 3 3 2 2 2 2 2 2 2 3 2 3 2 2 3 3 2 2 2 3 0 3 2 2 3 3 3 3 2 2 2 2

3 2 3 2 3 3 3 0 3 3 3 2 2 3 3 3 2 3 2 3 3 3 3 3 2 2 3 3 3 3 2 3 1 3 3 2 3 2 3 1

2 2 3 3 2 1 3 3 2 2 3 3 2 2 2 2 2 2 3 3 3 2 3 3 3 1 3 3 2 3 3 2 2 2 3 2 2 3 2 2

3 3 3 2 2 2 2 2 2 1 2 3 3 2 2 2 2 3 3 2 2 3 2 3 3 2 2 2 3 2 2 3 3 2 2 2 3 2 3 3

2 3 3 2 3 1 3 2 2 3 2 2 3 2 0 2 3 3 3 2 2 3 2 3 3 3 2 3 3 1 0 3 2 2 2 3 3 2 3 3

3 2 3 3 2 2 3 2 2 3 3 3 2 2 3 2 2 2 3 3 2 2 2 2 3 2 3 2 2 2 2 3 3 2 2 2 3 3 3 2

3 2 2 3 2 3 3 2 2 2 2 2 2 2 3 2 3 3 2 2 3 2 2 2 3 3 3 2 2 3 3 3 2 3 2 2 3 3 3 3

StateBlueCount = 5
StateRedCount = 395
StateCompBlueCount = 195
StateCompRedCount = 190
```

Fig. 11: Our system model after 5 time steps



stateBlueCount
31

stateRedCount
369

compStateBlueCount
244

compStateRedCount
125

Fig. 12: AnyLogic model after 5 time steps

As previously, the experiment was then altered to run in virtual mode in AnyLogic, which simply executes transitions as fast as possible, and without any GUI in our system. The results are presented in Table III for our system and in Table IV for AnyLogic.

TABLE III.
EXPERIMENT 2 RESULTS USING OUR SYSTEM

Agents	Ram	Timesteps
500	0.5 MB	1000 / sec
100,000	14.4 MB	2.99 / sec
250,000	34.1 MB	1.22 / sec
1,000,000	134.7 MB	0.29 / sec

TABLE IV.
EXPERIMENT 2 RESULTS USING ANYLOGIC

Agents	Ram	Timesteps
500	75 MB	9135 / sec
100,000	346 MB	16.5 / sec
250,000	540 MB	5.7 / sec
1,000,000	1,615 MB	1.5 / sec

Table III shows that the performance for our system has remained almost the same as in Experiment 1 in terms of memory usage and time steps executed per second. The same is true for AnyLogic in relation to the time steps executed per second, as can be seen in Table IV. However, memory usage has gone up quite significantly. This demonstrates the advantages of our state machine design. One can save potentially a lot of memory when constructing models that feature more complex statecharts.

V. CONCLUSIONS

This paper provides an insight into the often unexplored inside world of ABS environments. We have defined the different components (classes) of such a system, including `Environment`, `Agent`, `StateMachine`, `State`, `Transition`, and `CompositeState`. For each of these, we have defined what is expected of them in an ABS system, the links between them, and in addition we have explored a variety of different ways in which these can be implemented whilst fulfilling the requirements of an ABS system. We have also designed and implemented some basic tests which showed that the components work as expected.

In Section IV we have compared the performance of our system to that of the "off the shelf" package AnyLogic. The results demonstrate that with a synchronous time model in mind, memory usage of our tool is much lower than that of AnyLogic. This demonstrates the value of our ABS environment when considering larger models where memory usage can be of high importance, and where large agent populations are being implemented. However, there is still much room for improvement in terms of performance, which has become apparent from our experimentation.

We have extensively used object oriented analysis and design principles to create a system that is easy for others to use and extend. This should allow others to easily implement features they might want to include. Through providing UML diagrams we are hopeful that we have helped non computer scientists to understand, perhaps for the first time, how ABS works internally. Through our tests we have demonstrated that our C++ implementation is a promising solution for use within the area of ABS when memory usage is of a higher priority than features.

Currently we are continuing our validation efforts by building more complex real world models in our ABS environment and comparing their performance to the performance of existing models that we have previously built in AnyLogic, e.g. [25]-[27]. With regards to extending the ABS environment we are considering to provide researchers with an easier way of creating simple state machines. XML would be a useful file format to consider for storing models of finite state machines, as all the information on states and transitions could be stored in a structured way, and many tools which are used to draw UML diagrams support the XML file format. Here we follow the idea of [21] where a custom file format is used to interpret HSMs.

REFERENCES

[1] M. W. Macy and R. Willer, "From factors to actors: Computational sociology and agent-based modeling", *Annual Review of Sociology*, 28(1), pp. 143-166, 2002. DOI: 10.1146/annurev.soc.28.110601.141117

[2] E. Bonabeau, "Agent-based modeling: Methods and techniques for simulating human systems", *Proceedings of the National Academy of Sciences of the United States of America*, 99, pp. 7280-7287, 2002. DOI: 10.1073/pnas.082080899.

[3] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, Cambridge University Press, 2008.

[4] B. Bauer, J. P. Müller, and J. Odell, "Agent UML: A formalism for specifying multiagent software systems", *International Journal of Software Engineering and Knowledge Engineering*, 11(03), pp. 207-230, 2001. DOI: 10.1142/S0218194001000517

[5] G. Fortino, W. Russo, and E. Zimeo, "A statecharts-based software development process for mobile agents", *Information and Software Technology*, 46(13), pp. 907-921, 2004. DOI: 10.1016/j.infsof.2004.04.005

[6] G. Fortino, W. Russo, "ELDAMeth: An agent-oriented methodology for simulation-based prototyping of distributed agent systems", *Information and Software Technology*, 54(6), pp. 608–624, 2012. DOI: 10.1016/j.infsof.2011.08.006

[7] Engineering Agent-Based Social Simulations, CfP for a JASSS Special Issue, http://www.cs.nott.ac.uk/~pos/docs/pos-CfP-JASSS_EngineeringABSS.pdf [last accessed 12/04/2014]

[8] B. Hugues, "UML for ABM", *Journal of Artificial Societies and Social Simulation*, 15(1) 9, 2012.

[9] G. Booch, R. A. Maksimchuk, M. W. Engle, B.J. Young, J. Conallen, and K.A. Houston, *Object-Oriented Analysis and Design with Applications*, 3rd Edition, Pearson Education, 2007.

[10] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head First Design Patterns*, O'Reilly Media, Inc., 2004.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, 1994.

[12] K. Barclay and J. Savage, *Object-oriented Design with UML and Java*, Butterworth-Heinemann, 2003.

[13] P.-O. Siebers, and B. S. Onggo, "Graphical representation of agent-based models in Operational Research and Management Science using UML", in *Proceedings of the 7th OR Society Simulation Workshop (SW14)*, 1-2 Apr 2014, Worcestershire, UK.

[14] J. Bradshaw (ed.), *Software Agents*, MIT Press, 1997.

[15] J. Ferber, Multi-Agent Systems: *An Introduction to Distributed Artificial Intelligence*, Vol. 1, Addison-Wesley, 1999.

[16] C. J. E. Castle and A. T. Crooks, "Principles and concepts of agent-based modelling for developing geospatial simulations", *Working Paper 110*, London: University College London, Centre for Advanced Spatial Analysis, 2006.

[17] M. Buckland, *Programming Game AI by Example*, Jones & Bartlett Learning, 2005.

[18] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, 2006.

[19] Wikipedia, "File:Turnstile state machine colored.svg.", http://en.wikipedia.org/wiki/File:Turnstile_state_machine_colored.svg [last accessed 12/04/2014]

[20] J. D. Farmer and D. Foley, "The economy needs agent-based modelling", *Nature*, 460(7256) pp. 685-686, 2009. DOI: 10.1038/460685a

[21] J. Ricardo, E. B. Passos, W. Esteban, C. Bruno and C. Pedro, "Dynamic game object component system for mutable behaviour characters", in *Proceedings of the VII Brazilian Symposium on Computer Games and Digital Entertainment*, 10-12 Nov 2008, Brazil.

[22] R. C. Martin, "Java and C++; A critical comparison", *ObjectMentor*, 1997.

[23] A. Borshchev, *The Big Book of Simulation Modeling: Multi-Method Modeling with AnyLogic 6*. AnyLogic North America, 2013.

[24] XJ Technologies, http://www.anylogic.com/ [last accessed 12/04/2014]

[25] P.-O. Siebers and U. Aickelin, "A first approach on modelling staff proactiveness in retail simulation models", *Journal of Artificial Societies and Social Simulation*, 14(2) 2, 2011.

[26] T. Zhang, P.-O. Siebers, and U. Aickelin, "Modelling Electricity Consumption in Office Buildings: An Agent Based Approach", *Energy and Buildings*, 43(10), pp. 2882-2892, 2011. DOI: 10.1145/2422531.2422535

[27] T. Zhang, P.-O. Siebers, and U. Aickelin, "Modelling the effects of user learning on forced innovation diffusion", in *Proceedings of the UK OR Society Simulation Workshop 2012 (SW12)*, 26-28 Mar 2012, Worcestershire, UK.