Replication in Agent Based Models using Formal Methods and Multiple Updating Strategies

Joseph Kehoe BSc MSc

A Thesis submitted for the degree of Doctor of Philosophy

Supervisors: Dr. Martin Crane, Prof. Heather Ruskin

School of Computing

Dublin City University

June 2017

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of PhD is entirely my own work, and that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: ______________(Candidate) ID No.: 58118713 Date: 5th May 2015

Abstract

Replication in Agent Based Models using Formal Methods and Multiple Updating Strategies

Joseph Kehoe

Agent based Modelling is now a standard form of modelling in the field of social simulations. One of the outstanding issues in this field is known as the *Replication Problem*. Simply stated this is the inability of researchers to independently replicate the results of Agent Based Models published by other researchers. This thesis outlines a two-part proposed solution to this problem that uses formal methods to precisely specify Agent Based Models and employs multiple updating strategies to demonstrate that any simulation results are independent of updating strategy. A novel synchronous updating algorithm is presented that allows synchronus updating to be extended from Cellular Automata based simulations to Agent Based Simulations. Asymptotic analysis of the novel synchronous algorithm shows that it is $\Theta(n \log n)$ in time and $\Theta(n)$ in space. It is demonstrated that the new algorithm does not suffer from the deficiencies of existing asynchronous algorithms and that it also allows collision detection to be incorporated into agent behaviours. The applicability of this approach is demonstrated by specifying and then reproducing the results of Sugarscape, a well known and complex Agent Based Social Simulation. The first formal specification of Sugarscape is presented and used to identify the ambiguities that have made replication of the original results so difficult. A framework is developed that implements the new algorithm alongside existing asynchronus algorithms and is used to implement Sugarscape and compare the different updating approaches. This is the first synchronous implementation of Sugarscape and the only synchronous implementation of any Agent Based Model of comparable complexity. It is also the first comparison of synchronous and asynchronus updating applied to the same Agent Based Model.

Keywords: Agent Based Models, Agent Based Social Simulations, Replication Problem, Formal Methods, Z Notation, Synchronous Updating, Asynchronous Updating, Algorithms.

Contents

List of Figures

List of Tables

Acronyms

ABM Agent Based Model. 2–15, 21–31, 33, 35–37, 40, 42, 46–50, 65–68, 79, 81–84, 113, 115–117, 119–123, 126, 128–130, 133, 135–137, 154, 155, 157–160, 167, 168, 173, 178, 185, 188–194

ABSS Agent Based Social Simulation. 3, 4, 6, 10, 12–14, 24, 35, 37–40, 48, 66, 67, 115, 116, 119, 126, 128, 132, 150, 157, 159, 160, 164, 167, 169, 173, 178, 188–193

API Application Programmer Interface. 64, 66

AU Asynchronous Updating. 3, 5–7, 48, 49, 107, 120, 121, 125, 128, 129, 131–135, 158, 159, 161, 163, 170, 191–194

BEM Boundary Element Method. 24

BSP Bulk Synchronous Parallel. 46

CA Cellular Automata. 4, 18–20, 36, 38–40, 42–44, 46, 48, 49, 119, 121, 129, 135, 136, 168, 169, 178, 193, 194

CPU Central Processor Unit. 66

CUDA Compute Unified Device Architecture. 66

DES Discrete Event Simulation. 10, 22, 24, 25, 38, 65, 126, 154, 155

DMS Dynamic Microsimulation. 10, 22, 23

EBM Equation Based Model. 9, 11, 12, 22–24

FEM Finite Element Method. 24

GIS Geographical Information System. 16, 28

GPGPU General-purpose computing on Graphics Processor Units. 65, 203

GPU Graphics Processor Unit. 30, 36, 64–66, 118, 157

GUI	Graphical User Interface. 26, 27, 29
HDF5	Hierarchical Data Format, Version 5. 34
HPC	High Performance Computing. 28, 203
IBM	Individual Based Model. 3, 10, 31, 32, 41
IDE	Integrated Development Environment. 26
IPD	Iterated Prisoner's Dilema. 15, 16, 20, 21
MABS	Multi-Agent Based Simulation. 10, 15, 23
MPI	Message Passing Interface. 28
ODD	Overview, Design Concepts and Details. 31–33
ODE	Ordinary Differential Equations. 23
OOS	Object Oriented Simulation. 10
RAD	Rapid Application Development. 27, 30
SIMD	Single Instruction, Multiple Data. 65
SU	Synchronous Updating. 3, 5–7, 35, 48, 67, 101, 120, 121, 129, 131, 132, 139, 158–161, 163, 170, 190, 191, 193, 194
XML	eXtensible Markup Language. 66
XMML	X-Machine Markup Language. 66

Acknowledgements

I would like to thank my supervisors, (in alphabetical order) Martin Crane, Heather Ruskin and Joseph Morris for all their help in completing this thesis. I would also like to thank my family for their support, especially my wife Jacqueline. Obviously, none of this would have been possible without their help. The anonymous referees of the Journal of Artificial Societies and Social Simulation (JASSS), the Multi Agent Based Simulation (MABS) conference and the European Simulation and Modelling (ESM) conference provided helpful feedback and guidance in formulating my arguments.

Chapter 1

Introduction

1.1 Agent-Based Modelling

A simulation is a software based model of a real or hypothetical system [Banks et al., 2005]. Simulations are used in science to model real world systems ranging from simulations of entire economies [Deissenberg et al., 2008] to interactions between molecules [Bezbradica et al., 2014]. In games, for instance, they are used to model hypothetical systems with exotic physics such as the *Half Life* Universe [Valve and Gearbox, 1998].

An Agent Based Model (ABM) [1] is a particular type of simulation. The defining characteristic of an ABM [Helbing, 2011] is that the overall system properties are not programmed in, rather, the system is composed of a number of interacting entities, known as agents. Agents can range in complexity from simple particles in soot clouds to complex animals in ecosystems. These agents interact with each other in some defined manner and the overall system behaviour results from this interaction. A good example of this is flocking [Reynolds, 1987], where each individual in the flock follows three simple rules:

Alignment Each agent aligns itself towards the average alignment of its neighbours;

Cohesion Each agent moves towards the average position of its neighbours;

Separation Each agent avoids getting too close to any neighbour.

Even though each agent is only aware of its immediate neighbours, and reacts only to them, a global and cohesive flocking behaviour emerges as an overall system

[1] We use acronyms such as ABM, to refer both to the plural and singular forms.

property no matter how many agents are in the flock. ABM are used to study how the global system properties can arise from interactions between local groups of agents [Davidsson, 2000].

In an Agent Based Social Simulation (ABSS) we use an ABM to model social systems (usually involving people, that is each agent represents a person) [Davidsson, 2002]. In these cases the behaviour of the agents is usually intentional in nature (*proactive* instead of *Reactive*) and complex. ABM is a cross disciplinary field and as a result there is a variance in the terminology used. Ecologists will use the term Individual Based Model (IBM) in place of ABM, for example.

In an ABM there will be a number of agents all acting simultaneously and this needs to be implemented as accurately as possible. There are two approaches to implementing this termed Asynchronous Updating (AU) and Synchronous Updating (SU) [Huberman and Glance, 1993].

Under the AU approach [Huberman and Glance, 1993] each agent performs its behaviour and updates its state in sequence during each time step. The time step ends as soon as every agent has performed its behaviour or, depending on the AU ordering scheme used, after n behaviours have been performed where n is defined by the asynchronous scheme used. A consequence of this approach is that when we examine an agent's state during AU we can never be sure if this is the state as it was at the beginning of the time step or not.

There are different strategies [Schönfisch and de Roos, 1999] to help decide what the precise ordering of agent behaviours should be during each step. As we might expect each scheme [Radax and Rengs, 2010], [Caron-Lormier et al., 2008] gives different results. The ordering of updates within a step matters [Ruxton and Saravia, 1998] and each scheme tries to alleviate this in some fashion.

The SU approach updates all agents in unison (simultaneously) during each time step. This is normally achieved by each agent holding two copies of its state, known as the *current state* and the *next state*. The *current* state holds the value of its state as it was at the beginning of the time step. This state remains fixed in value throughout the step. If an agent inspects another agent it will see only the *current* state (that is, the state of that agent as it was at the start of this step).

Any agent that updates its state during a step only applies these updates to its *next* state. Only once every agent has finished performing its behaviour does the time step end. At this point every agent copies their *next* state into their *current* state in preparation for the following step. This system works well for systems with simple interactions, such as Cellular Automata (CA), but there has been little work done to implement synchronous updating for more complex interactions such as those found in an ABSS [Grilo and Correia, 2011].

1.2 Ensuring Reproducibility of Results

In this thesis we attempt to answer the question: how can we specify ABMs in a manner that helps solve the *Replication problem*.

The Replication problem, as it is termed in ABM [Sansores and Pavn, 2005] ,[Hinsen, 2011],[Edmonds and Hales, 2003], is simply the issue that it has proven very difficult for independent researchers to replicate or verify the results published by other researchers using ABM. There are a number of suggested reasons for this inability to reproduce results such as the lack of access to the original ABM source code and ambiguously defined models. Despite many attempts at solving this this problem and the supposed identification of its causes there is, as yet, no solution or even an agreed approach.

In effect the issue is one of imprecise specification of simulation models. Access to the source code of a simulation only allows other researchers to rerun the simulation, or exactly replicate the original experiment, but this is a weak form of reproducing experiments. Re-executing existing source code will produce the same results even if those results are due only to errors in the code or model specification.

An example of the weakness of this approach within ABM is when the simulation outcomes are caused by the specific updating strategy employed. These outcomes are termed *artefacts* (e.g. see [Huberman and Glance, 1993]) and have been known about for some time. Rerunning existing code, or for that matter, using new code that employs the same updating strategy will not distinguish between results that are artefacts and those that are independent of the updating strategy employed, that is, inherent properties of the system being modelled.

A stronger form of replication, sometimes known as *reproduction* [Peng, 2011, Drummond, 2009] to distinguish it from the weaker replication, allows researchers to independently redo (or reimplement) the original simulation model based on its definition and compare it to the original. This is the norm in experimental science where experimental results are confirmed by independently reproducing the experiment, based on the description of the experiment, in a different laboratory setting. When we refer to the replication problem it is this second stronger form of replication (reproduction) that we are referring to.

In this thesis we propose an approach to modelling that tackles the replication problem. It does this in two ways. First it employs formal methods, as used in safety critical systems, to precisely define the simulation models at a high level and secondly it allows the model to employ both classes of updating strategy (AU and SU) during implementation.

As we discussed in the previous section there are two broad classes of updating strategy termed AU and SU. Of the two of these the asynchronous approach is predominantly used in ABM [Radax and Rengs, 2010] due to the lack of an algorithm that implements synchronous updating correctly in ABM. In order to demonstrate that simulation outcomes are not simply artefacts of the asynchronous updating strategy employed we have derived novel synchronous algorithms that can handle the full range of complex interaction types present within ABM.

We provide complexity measures for our algorithms and demonstrate that they are comparable (in time and space) to the simpler asynchronous algorithms while also demonstrating weak scalability. As well as having run-time predictability (due to lack of locking) these algorithms have the added properties of:

1. Deadlock Freedom;
2. Livelock Freedom;
3. Deterministic Execution.

It has long been shown that AU and SU of the simple *Spatial Iterated Prisoners' Dilemma* [Majeski, 1984], [Newth and Cornforth, 2009] results in different outcomes and it has been postulated based on this that the fundamental differences between these two strategies means that they give divergent outcomes [Huberman and Glance, 1993], [Radax and Rengs, 2010] with one correct and the other incorrect. In this thesis we demonstrate that this is not necessarily the case and that while they may differ in certain respects they can both lead to the same emergent properties in ABMs.

The effectiveness of the proposed approach is shown by formally specifying Sugarscape a well known and complex ABM that helped jumpstart the use of ABM in the soft sciences (Social Science, Economics, Ecology, etc.). A framework that implements both the novel SU algorithms alongside a range of AU strategies is used to reproduce the original results of Sugarscape and show that AU and SU do not necessarily result in divergent outcomes.

1.3 Outline of Thesis

The next chapter is a survey of Agent Based Models. As well as an overview of the field in general we examine the current literature on the *Replication Problem* in ABM, the differences between synchronous and asynchronous updating and a survey of other approaches to synchronous updating in ABM

Chapter Three reviews the areas of concurrency that are of concern to our algorithms, in particular measuring scalability and the use of asymptotic analysis to produce complexity measures. We also survey concurrency in ABM.

Chapter Four presents an overview of the Sugarscape, a complex and well known ABSS that is widely used for benchmarking but known to be difficult to replicate.

Chapter Five lays out our proposed solution to the *Replication Problem*, that of employing formal specifications to precisely define ABMs independently of updating strategy and showing that the results of an ABM are not dependent on the updating strategy employed.

Chapter Six provides a critique of AU and develops novel algorithms that allow for SU in an ABM. The space-time complexity of these algorithms is derived using Asymptotic Complexity. *Deadlock Freedom*, *Livelock Freedom* and *Weak Scalability* of these algorithms is demonstrated.

Chapter Seven is a case study where we apply our approach to produce a benchmark that can be used in ABM and indicate any issues with our approach as well as its strengths. A framework is developed that implements both synchronous and asynchronous updating strategies and used to implement Sugarscape based on our formal specification. The robustness of Sugarscape is demonstrated by showing that the properties of Sugarscape are independent of the updating strategy applied.

Chapter Eight is the conclusion where we summarise our findings.

Appendix A lists some empirical benchmarking results showing that the actual performance of our algorithms is in line with the predictions of the asymptotic analysis.

Appendix B is a short code tutorial on using the synchronous/asynchronous ABM framework.

Appendix C contains the complete formal specification of Sugarscape.

Chapter 2

Agent Based Models (ABM)

2.1 Introduction

In this thesis we are interested in certain aspects of Agent Based Modelling. ABM is one approach to producing computer simulations of real-world systems. It is used across a variety of fields of science but is possibly most popular in the social sciences.

Here we will give an overview of ABM as it currently stands. In particular we will look briefly at its place in simulation and give an overview of the history of the development of ABM.

After this we will examine in more detail the characteristics of ABM in order to draw out its essential or defining properties. As we have said there are a number of different approaches to simulation. The choice of approach to simulation that a modeller takes will depend on a number of factors. We will look at these factors and briefly compare the other major approaches to the agent based approach.

Following this we we will look at some of the better known ABM toolkits that are available to developers. In the last section we look at some of the different application areas for which ABM has been used in so as to get an idea of its scope.

2.2 Computer Simulations

2.2.1 What are Simulations?

Computer Simulations are software based models of real or hypothetical systems. We follow the definitions given by [Banks et al., 2005]:

Simulation The process of running a model of a proposed or real system and observing its behaviour over time;

System A collection of entities (e.g., people and machines) that interact together over time to accomplish one or more goals;

Model An abstract representation of a system, usually containing structural, logical, or mathematical relationships that describe a system in terms of state, entities and their attributes, sets, processes, events, activities, and delays;

One of the ways that simulations can be categorised is based on the approach they take to time, either treating it as *Discrete* or *Continuous* [Pollacia, 1989]. A discrete treatment of time results in changes in state that occur only at countable points. Continuous time allows changes of state to occur smoothly and continuously. Of course some simulations are hybrid where some events are discrete and some are continuous. For example, a simulation of an oil refinery may model the flow of oil into, and out of, reservoirs using equations for liquid flow while at the same time modelling the arrival and departure of individual oil tankers dropping off or picking up oil from the reservoirs discretely.

Simulations can be categorised by the approach taken to produce the model. The main categories include [Davidsson, 2000]:

Agent Based Model (ABM) The system as a whole is modelled by simulating individual autonomous agents and their interactions. For example, the simulation of flocking in animals.

Equation Based Model (EBM) The system is defined by a set of variables and a set of equations employing those variables. For example, modelling predator-prey relationships using the *Lotka-Volterra equations* [Lotka, 1909].

Discrete Event Simulation (DES) The system is modelled as a discrete sequence of events in time where each event occurs at a moment in time and marks a change in state of the system. For example, modelling how customers' response time in a bank is affected by the number of tellers available in the bank.

Dynamic Microsimulation (DMS) The system is modelled by the simulation of basic micro units, each of which generally uses some stochastic model. For example [Mazzaferro and Morciano, 2008], analysing the long term redistributive effects of social policies.

Object Oriented Simulation (OOS) Any system that is modelled using the *Object-Oriented* Paradigm. This can include ABM or DES if they are produced using the OO paradigm. As any of the previous approaches can use an OO approach this is the most nebulous category here. A distinctive difference between OOS and ABM is that the entities in Object Oriented Simulation (OOS) tend to be purely reactive while those in ABM are proactive but there is no strict dividing line between them [Davidsson, 2000].

These categories will be detailed further below in section 2.4.

In this thesis we are concerned with ABM. ABM are sometimes known as IBM, mainly in the field of ecological simulations [Grimm et al., 2006, Topping et al., 2010, Altaweel et al., 2010], and also as Multi-Agent Based Simulation (MABS) but these can be taken to be synonyms and we will refer to them simply using the acronym ABM. An ABM that models interaction of individuals is sometimes better known as an ABSS [Davidsson, 2002].

ABSS occupies the intersection of the disciplines of Social Science, Computer Simulation and Agent Based Computing [Davidsson, 2002] and involves the use of agent technology for simulating social phenomena on a computer. In this thesis we will take the term ABM to include ABSS but will use the acronym ABSS whenever we feel it is important to emphasis the social nature of a simulation

The ABM approach to simulation has proven successful in what are sometimes termed the *soft sciences* [Epstein, 2006, Troitzsch, 2009][1] where the simulations consist of many heterogeneous individuals each with high level behaviours that determine how they interact with each other and the world. These include areas such as social science [Epstein and Axtell, 1996] and economics [Deissenberg et al., 2008, Troitzsch, 2009].

A defining characteristic of ABM is that the overall properties of the system being modelled are not directly programmed into the simulation but emerge from the interaction between the individual agents within the system as in, for example, simulations of flocking or swarming.

These areas of interest differ from what are loosely referred to as the *hard sciences*[2], such as chemistry and physics, where interactions are well defined and sometimes more deterministic. Here, because equations defining object interaction are known, the simulations can sometimes be more accurately modelled using

[1]Here the term *soft-science* refers to the dictionary definition [Dictionary.com, 2015] and is not pejorative in any sense.

[2]*Hard Science* usually refer to areas based heavily on mathematics.

EBM [Parunak et al., 1998].

Even though they are known as the *hard sciences* and many of the models used are very complex, it is the case that molecule to molecule interactions are often better understood than interactions between complex organisms and so are, in some respects, simpler and easier to model than organisms. Molecules are reactive in nature while organisms are proactive with high level goals and responses that can change over time [Gilbert, 2004]. We have, except in the simplest cases [Murray, 2002], no equations that accurately model the high level behaviour of people. A person, for example, will often remember previous encounters with other people and use this information to inform future interactions. Each interaction can, therefore, have a unique response that changes over time in response to previous interaction outcomes.

The interactions that occur in social simulations are, in general, less predictable than, for example, those that occur in molecular dynamics where all interactions are precisely specified and well known. This means that the overall simulation outcome in a social simulation is usually less predictable and can require a dependence on probabilistic outcomes. Because of the lack of clarity in how organisms interact, and a lack of equations describing the high level behaviour of individuals, ABM can provide a better to approach to modelling social simulations than EBM.

2.2.2 The Purpose of Simulation

ABM is employed in a wide variety of scientific disciplines [Heath et al., 2009]. As we have noted they are most notably used in the *Soft Sciences* (e.g. Social Science) where the range of suitable mathematical models available to describe their subjects and the component parts of their simulations tend to be less well understood.

An ABSS is defined to have the following properties [Epstein, 1999]:

1. **Heterogeneity** There are different types of agent within the ABM each with its own range of behaviours. For example an economic model may have one type of agent representing individual workers and another representing individual companies employing those workers;

2. **Autonomy** Each agent is autonomous (not centrally controlled) and proactive. Birds in a flock all act autonomously and the overall behaviour emerges out of their individual behaviour;

3. **Local Interactions** Agents can only interact with other agents if they are local to that agent. There is a well defined notion of locality (e.g. adjacent to or within a fixed radius of). For example, the modelling of the spread of *memes* through twitter could define the neighbourhood of a twitter user as those other users that (s)he is following;

4. **Explicit Space** The space inhabited by the agents is explicitly modelled and essential to defining the notion of locality. The space may represent a real world location (building, town or country) or an imagined space. In any event the ability of agents to move around or communicate with neighbouring agents is determined by the properties of this explicitly defined space;

5. **Bounded Rationality** Agents do not have global knowledge and have only limited processing power. Agents are not aware of anything outside of their locality.

ABSS presents a new method of evaluating theories in Social Science. In an ABSS the overall characteristics (or properties) of a population as a whole are not explicitly encoded into the model but emerge from the interactions between individuals within that population, as defined by items 2 to 5 above. This reliance on emergent properties is an important distinguishing feature of the ABM approach [Helbing, 2011]. ABM overcome the limitations of the more traditional simulation techniques. In the social sciences equations defining the behaviours of individuals either do not exist or if they do exist they rely on unrealistic assumptions [Gilbert and Terna, 2000]. If we take, for example, the *Black-Scholes-Merton model* [Black and Scholes, 1973], a mathematical model of a financial market containing certain derivative investment instruments, we can deduce the Black-Scholes formula, which gives a theoretical estimate of the price of European-style options. The model is based on the (ideal) assumptions that include:

- It is possible to borrow and lend any amount, even fractional, of cash at the riskless rate.
- It is possible to buy and sell any amount, even fractional, of the stock (this includes short selling).
- The above transactions do not incur any fees or costs (i.e., frictionless market).

The above assumptions, however, are false which can lead to questions regarding the validity of the model outcomes.

In situations where suitable equations do exist they can still be incorporated into an ABM [Rank, 2010] if required.

ABSS has faced criticism not merely for its lack of governing equations but also because they are not deductive [Epstein, 2006]. This criticism is due to a misunderstanding about how ABM work and what they purport to offer. At a fundamental level a computational model is itself a mathematical formalism based on the theory of recursive functions. It is just that it is not expressible in a more traditional mathematical form. In point of fact many equation-based models are themselves implemented by generating approximate solutions to their defining equations [Epstein, 2006] so this criticism need not be confined just to ABM.

Simulations exist not just to provide predictions but for many other reasons. Epstein lists sixteen (valid) alternative reasons for simulations [Epstein, 2008].

Gilbert and Terna [Gilbert and Terna, 2000] see ABSS as a third way of carrying out social science in addition to argumentation and formalisation, or a new form of hybrid theoretical computational work. It represents a generative approach based on a constructivist philosophy of science. Equation 2.1 [Epstein, 2006] represents the motto of this approach. It can be paraphrased as *If you did not grow it, then you did not explain it.* ABSS provides new insights and a new approach to use simulations in disciplines where traditional simulation methods do not work.

$$\forall x \bullet (\neg\, G(x) \supset \neg\, E(x))^{\dagger} \tag{2.1}$$

2.3 What are Agent Based Models?

2.3.1 Characteristics

ABM are models composed of interacting autonomous agents where agents can be defined as entities with the following characteristics [Macal and North, 2009]:

Autonomous and Self-directed An agent can function independently in its environment;

Modular and Self-contained Agents are discrete and contain individual attributes and behaviours;

[†]For any model x if you did not grow (G), or build, the model then you did not explain (E) it.

Social – interacting with other agents Agents have protocols or rules describing how they can interact with other agents

In addition agents may

Live in an environment Agents are situated within and can interact with their environment. Agent behaviour is situationally dependent. That is, behaviour is influenced by the state of the agent's surrounding environment;

Have explicit goals that drive their behaviour An agent has goals that allow it to compare the outcomes of its behaviours when assessing what actions to take. For example, survival or reproduction;

Be able to learn and adapt An agent may be able to change its behaviour based on past interactions so as to better meet its goals (as in the Iterated Prisoner's Dilema (IPD), for example);

Have resource attributes Agents may have, collect and use stocks of resources. These are collected from its surrounding environment and/or agents.

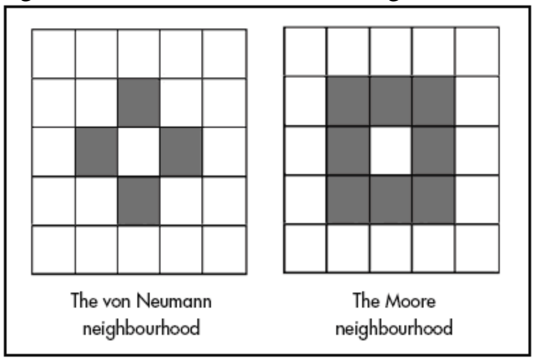
In Computer Science ABM is used in the fields of Artificial Life (the study of man-made systems that exhibit the behavioural characteristics of natural living systems) [Langton, 1986] and Multi-Agent Based Simulation (MABS) (groups of agents designed to solve problems that are beyond the capabilities of each individual agent [Rudowsky, 2004]).

2.3.2 Topologies

As we stated above (section 2.1) the overall properties of an ABM emerge from the interaction of its individual agents. Since agents interact only with other agents in their neighbourhood the composition of, and the relationship between these neighbours is important. This relationship defines the topology of the ABM. Topology defines how agents are allowed to interact with each other and will determine how we implement the models. We can identify five different topology types used in ABM (adapted from [Macal and North, 2009]):

Random Pairing The simplest topology is the aspatial one. Here there is no spatial representation. Agents are kept in a common pool and pairs of agents are

Figure 2.1: Cellula Automata Neighbourhoods



selected randomly[3] and allowed to interact. After each interaction agents are returned to the common pool. In this case the number of neighbours that an agent has is always one but at each turn it may be a different agent. For example, a simulation of the *Iterated Prisoner's Dilema (IPD)*[Poundstone, 1992] (defined in section 2.3.3) might allow agents to pair off randomly against each other and to take a turn playing *Prisoners' Dilemma*. Since any one agent is allowed pair off with any other agent at random there is no spatial topology and opposing agents are just chosen at random.

Discrete Spatial In this case the Cellular Automata based topologies represent space as a lattice or grid of discrete locations. This grid may be bounded in size and if bounded may be toroidal (i.e. there is a wraparound on the grid edges). These grids are generally two dimensional but higher and lower dimensional grids, although much less common[4], can be used. Agents are placed at locations in the grid and allowed to interact only with their neighbours typically using either the von Neumann definition of neighbourhood or the Moore definition as shown in figure 2.1. The best known examples of such simulations are Conway's *Game of Life* [Gardner, 1970], a simple homogeneous Cellular Automata and Sugarscape [Epstein and Axtell, 1996], a much more complex heterogeneous simulation.

Continuous Spatial Continuous spatial models use a Euclidean 2D or 3D topology. These models more closely resemble the real world. Cellular automata

[3]Randomness in a simulation is achieved by employing a *pseudorandom number generator* [NAG, 2006], such as *Mersenne Twister* [Matsumoto and Nishimura, 1998]. In this thesis we will follow this convention in all references to randomness.

[4]In ABSS it is generally possible to model the real world as a 2D surface.

use a discrete model of the world while Euclidean models use a more accurate continuous model. The best known example of this approach is *Reynolds flocking simulation* [Reynolds, 1987]. Continuous models are popular in the field of Artificial Life where a close mapping to the physical world is important.

Geographical Information System Based Spatial More recently Geographical Information System (GIS) have been used as the basis for ABM topologies [Crooks et al., 2008]. Here a realistic geospatial landscape accurately representing a real world location is used as the spatial basis for the model which can be either discrete or continuous. This gives a more accurate mapping between the simulation and the actual space being modelled. GIS can be seen as a special case of either the continuous or discrete spatial topology as its only effect is to initialise the model with values that map closely to some existing real world space. The lattice is basically overlaid on a map of the region being modelled. This is then used to inform the relationship between adjacent locations in the lattice;

Aspatial Network Finally a network topology can be used where the links between agents are defined in a graph that can be static or dynamic (e.g. for time-dependence) in nature. The network need not map to, or conform with, any spatial requirements. An example of this type of model would be a simulation of how trends or ideas spread in online communities such as Facebook [Hummel et al., 2012]. Here, distance between any two agents could be measured as the shortest path of "friend" links between the Facebook pages of any two individuals. *Scale free* or *Small world* networks can be constructed to match the properties of particular social aspatial topologies.

2.3.3 Neighbourhood

The term *Neighbourhood* is ubiquitous in ABM. Every agent has a defined neighbourhood and when we talk of neighbourhood we are always referring (implicitly or otherwise) to the neighbourhood of one specific agent. This neighbourhood is simply that subset of the system or world being modelled that an agent has immediate knowledge of. Once an event occurs in an agent's neighbourhood that agent is immediately (or instantaneously) aware of that event. An agent is not immediately aware of any event occurring outside of its neighbourhood. It may become aware that the event has occurred at some later point in time as news of the event travels through the simulated world and eventually reaches the agent's neighbourhood (passing through one or more overlapping neighbourhoods along the way). There is an associated time delay in any agent receiving information about an event that occurred outside its neighbourhood. This time delay is problem dependent as each simulation will have rules as to how far information can travel in one time step.

It is also the case that within ABM each neighbourhood represents only a small fraction of the entire world. No agent is allowed immediate knowledge of everything in the world. In other words omniscience is not allowed.

In an ABM of the real world the most common type of neighbourhood is spatially based. For example, in a car traffic simulation each agent (in this case each agent represents a vehicle) has a neighbourhood consisting of only those other agents (vehicles) that are visible to that agent. This will likely consist of a small number of vehicles immediately ahead of the agent and possible one vehicle to its rear (visible through the rear view mirror). As time passes and traffic flows each agents neighbourhood will change dynamically as other vehicles change position and road conditions change (fog might reduce neighbourhood size).

However, neighbourhoods need not be defined in terms of spatial topography. For example in a simulation of Twitter users each user's neighbourhood can be defined as those others being *followed* by that Twitter user. Thus it is possible and even likely that a user's neighbourhood will include people dispersed around the globe, as we tend to follow users based on shared interests rather than physical proximity. This results in information flows that are unexpected. Such a model could be used to explain why traditional government techniques of controlling information flow are unsuccessful, for example in conflict zones such as Syria.

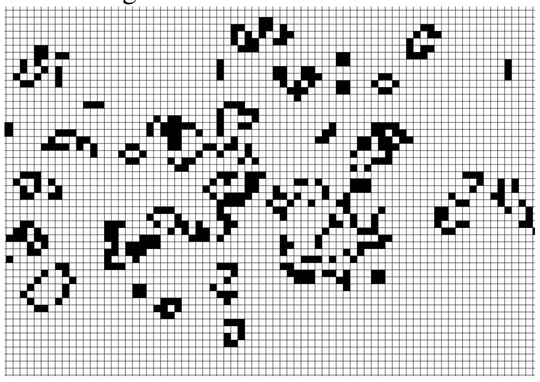
The use of aspatially based ABM extends the definition of neighbourhood from the more traditional one based on physical proximity to a more general information theoretic definition. It allows ABM to be employed in simulating online communities and the more complex interconnections now taking shape in societies.

2.3.4 History of ABM

It is difficult to categorically state when ABM began but the safest place to begin a history of ABM is probably in the Los Alamos national Laboratory in the 1940's [Sarkar, 2000]. Around this time CA were proposed as a formal model of self-replicating organisms. It was shown that CA are capable of universal computation, demonstrated by constructing a universal Turing Machine using CA [von Neumann, 1951].

This means that anything, that can be computed, can be computed by a CA.

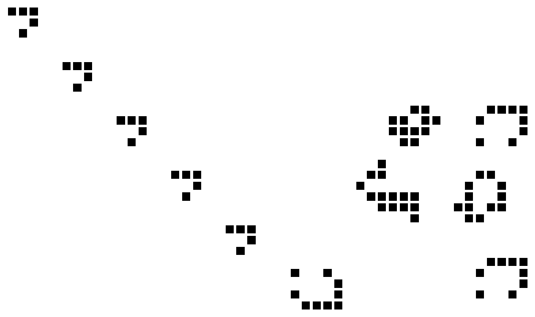
Figure 2.2: Cellula Automata



A CA is defined as an N-dimensional array of cells, where N is some integer greater than zero. While the original work assumed a single dimensional array, two (and occasionally higher) dimensional arrays are more commonly utilised today. Each cell in the array must be in one of a finite number of discrete states. Time is modelled as discrete steps or *ticks* and at each step all cells change state simultaneously. The next state of a cell is completely determined by its current state and the current states of its neighbours. In a one dimensional array neighbours are generally defined as cells to its immediate left and right while a two dimensional array defines neighbours as either those cells immediately north, south, west or east of it, known as the *von Neumann neighbourhood* or its eight surrounding cells – north, south, east, west, northeast, northwest, southeast and southwest, known as the *Moore neighbourhood* (see figure 2.1). In all cases the neighbourhood is a well defined subset of the array. No matter how large the simulation space becomes the size of a neighbourhood remains constant. The rule used to determine the state of a cell is the same for all cells. There is no input to the CA from outside, it is a closed world. At time 0 cells are placed in some initial state and from that point onwards the cells' state change deterministically according to the defined rule.

CA show how individual behaviour can cause population wide properties to emerge (known as *emergent properties*). The best known example of a CA demonstrating emergent properties is John Conway's *Game of Life*, first popularised in a *Scientific American* article [Gardner, 1970]. Here cellular automata were used to study the macroscopic behaviour of a population (e.g. The ”space rake”, which moves orthogonally ten units through a twenty step cycle, emitting one glider per cycle - Figure 2.3), an obvious precursor to the Agent-based approach to modelling within the social sciences.

Figure 2.3: Space Rake Emitting Gliders (Game of Life)



At around the same time that Conway was developing the *Game of Life* the influential paper "Models of Segregation" [Schelling, 1971] was published to demonstrate *the general theory of tipping*. In this case the model demonstrated how individual preferences to live near people of the same colour or even near a certain mixture "up to some limit"[5] could lead to segregation in neighbourhoods. This again showed how individual actions can have macro-affects in large populations. The author demonstrated two things:

1. The law of unintended consequences. Even though no individual within the population wanted segregation their individual behaviour caused that outcome;

2. A small change in people's preferences can have a large macro-level effect, that is, it can form a tipping point that causes a global change in state.

Despite the fact that this is a very crude model of human interaction, it is still used to help explain how segregation can occur in the real world.

While CA began as a formal model of self-reproducing organisms it was not until 1979 that the term *Artificial Life* [Langton, 1986] was proposed. The scope of previous work was extended to study the logic of living systems within artificial environments in order to gain insights into information processing in living systems. Artificial life extended the scope of CA from studying existing systems to studying living systems "as they might be". This can be seen as the start of the switch, from using simulations to make predictions about systems, to using them instead to explain the core dynamics of systems.

[5]For example, an individual might only be comfortable in a neighbourhood where at least half the residents were of the same colour.

Table 2.1: **Prisoner's Dilemma Payoff Matrix**

		Player X	
		Cooperate	Defect
Player Y	Cooperate	Two for each Player	Three for X, 0 for Y
	Defect	0 for X, 3 for Y	One for each Player

In 1981 a simulation of the IPD was developed [Axelrod and Hamilton, 1981]. This marks the change from simple CA based agents to ones where agents have higher level behaviours and motivations. Here the dilemma is phrased in terms of two prisoners trying the make the best decision. It can be stated as follows [Poundstone, 1992]:

> "Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of speaking to or exchanging messages with the other. The prosecutors do not have enough evidence to convict the pair on the principal charge. They hope to get both sentenced to a year in prison on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to: betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. Here is the offer (see also figure 2.1):
>
> - If A and B each betray the other, each of them serves 2 years in prison
> - If A betrays B but B remains silent, A will be set free and B will serve 3 years in prison (and vice versa)
> - If A and B both remain silent, both of them will only serve 1 year in prison (on the lesser charge)"

If two players play prisoners' dilemma more than once in succession and they remember previous actions of their opponent, so as to allow them to change their strategy accordingly, the game is called the *Iterated Prisoner's Dilema (IPD)*. The IPD is used to model individual cooperation within societies. It shows how different strategies might arise and evolve over time in populations and has found many application areas such as, for example, predicting the outcome of arms races [Majeski, 1984]. It has also been used to help to explain how seemingly irrational behaviour such as altruism [Dawkins, 1976] might have evolved. The IPD has been studied across many scientific disciplines since it first appeared.

Soon after the IPD was developed group behaviours in the animal world were modelled to show how flocking behaviours can emerge from individual behaviour [Reynolds, 1987]. It was demonstrated that it was possible for a flock of individuals to act seemingly as a single entity even though no single entity in the flock is in charge or is even aware of more than a few close neighbours. This flocking algorithm works no matter what the size of the flock is without requiring any increase in neighbourhood size. Here again ABM was used to explain rather than predict - an important difference between the ABM approach and other modelling approaches.

More recently the development of the Sugarscape [Epstein and Axtell, 1996] simulation and accompanying book helped jump-start the use of ABM in the social sciences. Sugarscape demonstrated that it is possible to model high level social interactions and how these interactions can effect the outlook of society as a whole. For example it can show how wealth disparities can occur, how disease is transmitted through populations or how cultural memes spread. This was the spring board to using ABSS in areas as diverse as economic policy development [Šperka and Spišák, 2012, Deissenberg et al., 2008], disease transmission [Parker and Epstein, 2011] or even helping archeologists model how ancient civilisations [Axtell et al., 2002, Dean et al., 2000, Lake, 2014, Campillo et al., 2012] functioned and why they thrived or collapsed.

2.4 Alternatives to ABM

2.4.1 Introduction

There are a number of alternatives to ABM when modelling. The most common, as previously mentioned, are Discrete Event Simulation (DES), Dynamic Microsimulation (DMS) and EBM. The choice of approach taken will depend on a number of factors. Questions that a modeller would need to answer include:

- Do they have any expertise in using any one approach?
- At what level is the model defined (or definable), the global level versus local level or some combination of the two?
- How is the model defined: Equation-based, stochastic, etc.?
- Are there any existing simulations that can be built upon?

We will look at each approach in turn and show when each approach is applicable.

2.4.2 Dynamic Micro-Simulation (DMS)

Microsimulation [Li and O'Donoghue, 2013] is popular in the social sciences and involves modelling and simulating the behaviour of the basic micro unit instead of the aggregate behaviour of the system. The definition of a microsimulation and basic micro unit depends on the system being modelled but could be, for example, a household or firm in which case the basic micro units would be the occupants of the household or employees of the firm respectively.

Individual behaviour is usually described by a stochastic model. A stochastic model is a tool for estimating probability distributions of potential outcomes by allowing for random variation in one or more inputs over time. The random variation is usually based on fluctuations observed in historical data for a selected period using standard time-series techniques.

The model can be run using discrete or continuous time. When discrete time is used the time interval is usually a year in length but discrete time steps can be computationally expensive [Li and O'Donoghue, 2013].

DMS is commonly used for pension models [Mazzaferro and Morciano, 2008] and [Li and O'Donoghue, 2013] presents good survey of DMS models.

Two limitations of DMS compared to ABM are identified by [Davidsson, 2000]:

> "Compared to MABS, DMS has two main limitations. First, the behaviour of each individual is modelled in terms of probabilities and no attempt is made to justify these in terms of individual preferences, decisions, plans, etc. Second, each simulated person is considered individually without regard to its interaction with others. Better results may be gained if also cognitive processes and communication between individuals were simulated and by using agents to simulate the individuals, these aspects are supported in a very natural way."

2.4.3 Equation-Based Modelling (EBM)

EBM are used when there is a defined equation that predicts the system behaviour over time. Once the equation is defined, simulation consists of assigning initial values to the equation and iteratively producing the equation output (which usually also serves as the input to the next iteration). This has proved a popular and successful approach but it assumes that there exists an equation that models the system accurately. This is not always the case either because no such equation exists or, it exists but is unsolvable or, because we have not "solved" the system yet [Axtell and Axtell, 2000]. Even if such an equation exists the modeller must be versed in the associated mathematical theory to be able to understand the implications of the equation. An issue with this approach is the fact that, in order to produce solvable equations, sometimes simplifying assumptions must be employed [Black and Scholes, 1973] that can lead to inaccuracies in the simulation.

There are a number of approaches used in EBM. Among the more common are:

Ordinary Differential Equations (ODE) An ODE is a differential equation containing a function or functions of one independent variable and its derivatives. A well known example is the *Lotka-Volterra equations* [Lotka, 1909], also known as the *predator-prey* equations.

Finite Element Method (FEM) FEM is a numerical technique for finding approximate solutions to boundary value problems for (partial) differential equations. It subdivides the problem into simpler parts called finite elements and uses variational methods from the calculus of variations to simulate the system by minimising an associated error function. Briefly FEM amounts to finding approximate solution values at discrete nodal positions on a mesh. The overall solution uses variational methods to minimize an error function by assuming that the overall solution is made up of an piecewise-continuous interpolation through the nodal values. For example the *Navier-Stokes equations* [Girault and Raviart, 1986] can be used to simulate airflow around an obstruction.

Boundary Element Method (BEM) BEM is a numerical computation method of solving linear partial differential equations which have been formulated as integral equations. It is used in fluid mechanics, acoustics and electro-magnetics.

The approach taken will depend on the problem at hand. In some cases [Parunak et al., 1998] the same system can be simulated both by EBM or ABM.

2.4.4 Discrete Event Simulation (DES)

DES is a simulation method that is widely used, particularly in distributed simulation implementations [Siebers et al., 2010]. It contains of an event queue based-simulation where the processes are all well defined. It is a top-down approach that focuses on the model instead of the entities in the model and has a centralised thread of control with passive entities.

DES can be subdivided based on how they treat time. They can use fixed increment time advance where the simulation runs at fixed intervals of time. As can be seen from the approach of the toolkits (in the next section) most of the ABSS use this approach to time as well. The other approach is the more common approach in DES: *Next Event Time Advance* [Banks et al., 2005]. Here a queue of events is stored centrally. Each event has a time stamp that indicates when the event is due to execute. Events in the queue are ordered by time stamp and during each turn the simulation takes the next event(s) from the queue, advances the system time to that event's time stamp and executes that event (or events if more than one event shares that time stamp).

DES can be used for ABM when the events and processes in the simulation are local instead of global (where a top-down approach is not applied). In this case it just marks a change from a clockwork interpretation of time to an event based one. This has been done with Sugarscape [Zaft and Zeigler, 2002] and it is claimed [Onggo, 2010] that any ABM can be converted into a DES.

The above approach is based on the assumption that a defined computable model of system level behaviour is available for use. In many cases this is not the case and even where it is, the computable model is often based on assumptions that are not strictly true. This approach does not necessarily reflect the behaviour of the systems component parts accurately as these parts are relegated to a secondary role in the model. Such a model does not explain how emergent behaviour can emerge from disparate interacting components to give the overall system properties.

There is no reason why different approaches can not be combined where possible. We might have, for example [Rank, 2010], equations representing certain parts of the system and interactive agents other parts.

2.5 Toolkits

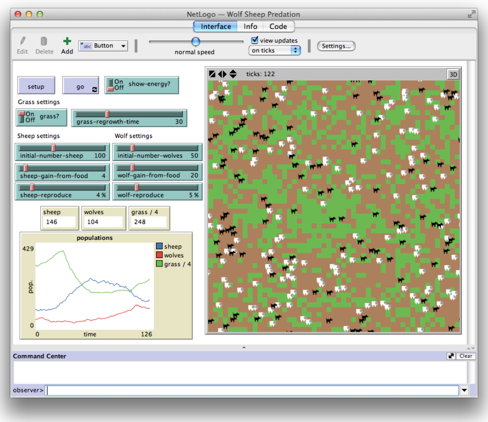
2.5.1 Introduction

There are a large number of toolkits available to aid in the production of ABM. They range from frameworks (or libraries) for particular programming languages to fully self contained scriptable environments. Railsback [Railsback et al., 2006a] compared four of the most popular ABM toolkits (*Swarm, repast, Mason* and *NetLogo*) and found that each had its own particular advantages. Each of these is fully open source, a definite advantage for scientists who need to know the limitations of any models they create. We will look at four of the more popular toolkits. All, bar one (Mason), assume a clockwork definition of time.

2.5.2 NetLogo

NetLogo [Tisue and Wilensky, 2004, Tisue, 2004] consists of a full Integrated Development Environment (IDE) complete with a Graphical User Interface (GUI) capable of displaying a simulation while it is being executed and the results of a simulation once it has completed. It is arguably the easiest of the toolkits to set up, learn and use. It contains the most complete set of documentation of all the available toolkits [Railsback et al., 2006b].

Figure 2.4: NetLogo



Models in NetLogo are developed in the high level *Logo* language. The *Logo* language is interpreted and single-threaded making NetLogo the slowest of the ABM toolkits. This use of a high level language makes the model development process simpler than it is with the other toolkits where a more in-depth familiarity with software development is required. This simplicity may explain why it is the most popular ABM toolkit in use.

NetLogo assumes a discrete approach to ABM and imposes a discrete space representation, specifically a two dimensional toroidal grid. Further NetLogo assumes a sequential execution model with asynchronous interactions. While internal parallelism is not possible it is capable of running the same model simultaneously on different processors.

NetLogo appears to be the least flexible because of its insistence on a lattice based space representation, and is the least amenable to concurrency. Most users of

NetLogo are not programmers so the availability of, for example, the Java Threading Model is of little practical use to them. Simple usage appears to be key for non-computer science based modellers.

NetLogo is the best Rapid Application Development (RAD) environment for ABM. It enables researchers without an in-depth programming skill set to produce usable ABM. The imposed constraints of discrete space and time representation also reduce flexibility in the type of models developed.

2.5.3 Repast

Repast (*RE*cursive *P*orous *A*gent *S*imulation *T*oolkit) [North et al., 2006] is a library of classes, originally written in Java, that was designed specifically for ABMs of complex adaptive systems, particularly within the social sciences. It was based on the *Swarm* simulation toolkit which is also library based and was originally developed in, and for, Objective-C. Although Swarm has since been reimplemented in Java it has been overshadowed by the arrival of Repast and other new libraries so its use is declining.

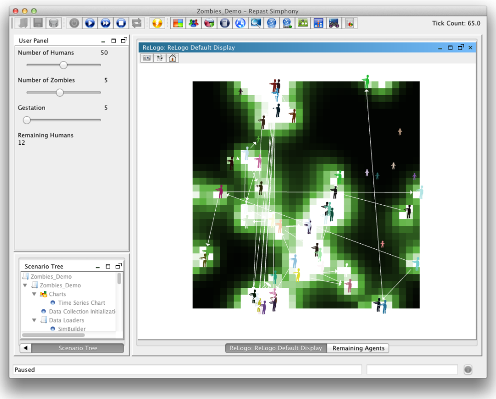
Repast is one of the fastest (in terms of execution speed) ABM libraries, second only to Mason. There are now three versions of Repast [North et al., 2013b]: the original Java implementation, *Repast.NET* – built on the Microsoft .NET CLI framework in $C\#$ [Hejlsberg et al., 2003] and *Repast Py* a Python-based implementation. *Repast Py* comes with a point-and-click GUI and a RAD environment for rapid prototyping of ABMs. It uses a subset of the Python language NQP (Not Quite Python) [North et al., 2013a] as an internal scripting language for building models.

The Repast Simphony Toolkit [North et al., 2013b] (Figure 2.5) builds on top of the Repast framework. It is a reimplementation of Repast with many extras added to make development easier. These extras include a plug-in for Eclipse, a well-known and portable IDE, and the use of the *ReLogo* scripting language to allow for easier and faster coding. *ReLogo* is based on *StarLogo* (which is similar to *NetLogo*) and uses a Logo style syntax based on the Groovy language[6]. It has visualisation tools, GIS integration and allows the use of concurrency through the Java Threading Model.

For large scale simulations there is Repast High Performance Computing (HPC) which is implemented in C++ and uses Message Passing Interface (MPI) to allow

[6] `http://www.groovy-lang.org`

Figure 2.5: Repast Symphony

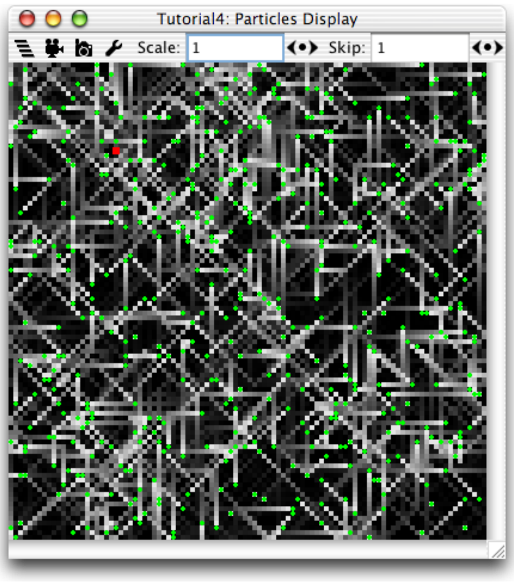


Repast models to run on supercomputing platforms. Repast HPC demonstrates excellent weak scalability.

2.5.4 Mason

Mason (Multi-Agent Simulator Of Neighbourhoods) is a set of Java libraries [Luke et al., 2005, Bigbee et al., 2007] that is designed specifically for speed. It contains both a simulation layer and a visualisation layer (see Figure 2.6).

Figure 2.6: Mason



It assumes a discrete event simulation based model and has been used across a range of different disciplines. It can also use the underlying Java Threading Model for concurrency. *D-Mason* (Distributed Mason) [Cordasco et al., 2013] is another version of Mason with facilities that support distributed ABM execution as well as monitoring facilities and interaction capability within a running simulation. It is designed for HPC platforms and uses MPI to distribute a single ABM across multiple processors.

Mason is designed to be fast and portable. An ABM in Mason will always give the same results regardless of the platform it is running on. This helps ensure that its models are all repeatable and give replicable results. We will return to the issue of repeatability of results in section 2.6.

2.5.5 Ascape

Ascape [Inchiosa and Parker, 2002, Parker, 2001] is a framework for development, visualisation and exploration of ABM. It is based on the Sugarscape ABM and has the design goals of being:

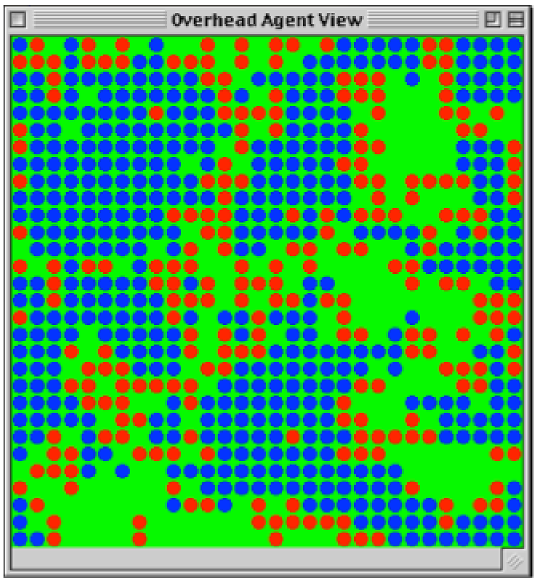
1. Expressive;
2. Generalised;
3. Powerful;
4. Abstract.

It comes with a GUI for control and viewing of an executing model. It is implemented in Java and combines the flexibility of using Java with a wide selection of built-in tools that allow high level point-and-click control over how the ABM is run. It represents a half-way house between the NetLogo RAD environment and the framework based approach of other toolkits (see Figure 2.7).

2.5.6 Other Toolkits

Other more specialised toolkits are available that offer concurrency. However these toolkits do not fully hide the complexity involved in parallelising code (see Chapter Three). Even FlameGPU [Richmond et al., 2010] which is designed specifically for concurrency on the Graphics Processor Unit (GPU) cannot fully hide the complexity involved. Code suitable for GPU execution must still be written by the modeller.

Figure 2.7: Ascape modelling Prisoners Dilemma - Blue are cooperators, Red are Defectors



A comparison [Berryman, 2008] of available ABM toolkits for complex adaptive systems found that Mason and Repast scored highest, but NetLogo was highly recommended for people with little programming experience due to the complexity of the other toolkits.

Other key issues are *repeatability* and *correctness*. When models are used as evidence to either back up existing theories or to help create new hypothesises it is paramount that the modeller is confident of the results provided by the ABM and that those results are reproducable by other researchers. In this regard determinism (as guaranteed by Mason) is a major advantage. We will look at the repeatability of simulations next.

2.6 The Replication Problem

The replication problem is simply stated: although simulations are developed in quantity and their results peer reviewed and published it has proven difficult for researchers to replicate these published results [Hinsen, 2011], [Sansores and Pavn, 2005], [Edmonds and Hales, 2003].

The reasons for this difficulty in replicating published results is that published papers do not include the actual code that produced those results or, in many cases, the raw data used to quantify them. If we wish to check that the published results are accurate we must re-implement the simulation ourselves, seldom an easy task. Even then, if our results do not match the original we cannot be sure that it is our implementation that is correct. Modellers need to provide a clear and precise description of the model to work from and this is not always present.

This can be a particular issue for modellers in the natural sciences who may not be as familiar with computer science as is required to ensure that their implementation is a correct interpretation of the model.

If the results of a simulation cannot be replicated then we can have little trust in the published results. Being able to repeat or reproduce scientific results is an essential part of science. There have been a number of suggested solutions to this problem.

2.6.1 Overview, Design concepts, and Details

Overview, Design Concepts and Details (ODD) [Grimm et al., 2006] is a protocol for specifying IBM and/or ABM. ODD was introduced to overcome shortcomings in published IBM definitions, specifically the issue that IBM are not specified precisely enough to allow for replication of the simulations.

The ODD protocol consists of three blocks (Overview, Design concepts and Details) divided into seven sections:

1. Purpose;
2. State variables and Scales;
3. Process overview and scheduling;
4. Design Concepts;
5. Initialisation;
6. Inputs;
7. Submodels.

By enforcing a particular structure on simulation descriptions it (i) makes specifications easier to read and (ii) ensures that modellers do not forget to specify any aspect of the model. ODD was originally tested by 28 modellers within the field of ecology to test its usefulness. Since its introduction it has been used in over 50 publications [Grimm et al., 2010]. The long term aims of ODD are stated as:

> "modellers could describe their IBM on paper using some kind of language that (1) people can understand intuitively, (2) is widely used throughout ecology, (3) provides 'shorthand' conventions that minimize the effort to describe the IBM rigorously and completely, and (4) can be converted directly into an executable simulator without the possibility of programming errors."

These aims seem quite naive from a computer science perspective where such a goal has been attempted for many years without success [Brooks, 1987].

ODD has been criticised for not being formal enough [Galan et al., 2009] but the chasm between the current approach to specifications taken in IBM and those more formal models used in computer science is clear. For example it is stated that it is often the case that the specifications produced by each role in the simulation development process (four are identified: *Thematician, Modeller, Computer Scientist* and *Programmer*) might *stay in the realm of mental models, and never reach materialisation*! In other words it is considered acceptable for the developers to produce no design for one or more phases of the development process.

ODD has been extended by adding an algorithmic model to specify the behaviours in the IBM [Hinkelmann et al., 2011]. Although interesting, it is lacking any of the modularisation features used by computing specification languages such as Z [Spivey, 1989].

2.6.2 Model Alignment

An early paper to point out and address this problem [Edmonds and Hales, 2003] concluded that we cannot trust simulation results that have not been replicated. Simulations should be treated like experiments in this regard. They illustrate this by taking a published simulation model and producing two independent implementations of it. They then compared the results of their two independent implementations against the original to find differences in the outcomes. They state that producing two different implementations gave them confidence in their results, where they differed with the original results, that they would not have had with only one implementation.

The process of re-implementing a model they called *model alignment* and they suggest that, as most simulations are not amenable to formal analysis, experimentation is the only route to verification.

While we agree that experimentation is useful it does not solve the problem of specifications that are vague or are missing important information. If we do not agree the rules of the ABM with which we are experimenting it will not help.

2.6.3 A Model Driven Approach

Following from the conclusion of [Edmonds and Hales, 2003] that experimentation is the only route to verification, [Sansores and Pavn, 2005] list two closely related approaches:

Re-implementation The model is rewritten following the original authors instructions;

Alignment or Docking We implement a conceptually identical model *but* using different tools and different methodologies.

They favour alignment as the better approach and propose using an independent modelling process that can be automatically implemented using a number of different ABM toolkits. This has the advantages that it becomes easier for modellers to provide multiple implementations and does not require in depth programming skills from them.

Such an approach would be useful in that a number of implementations can be compared but it still only gives use replication not reproducibility. We can replicate (copy) their results but it does not allow us to reproduce their results. In other words if there is some error in their approach then *Alignment* will not find that error, only repeat it.

2.6.4 Executable Papers

Hinsen [Hinsen, 2015, Hinsen, 2014, Hinsen, 2011] identified the problem as belonging to the entire computational science field. His proposed solution is to produce *executable papers*, termed *ActivePapers*. That is, published papers, as well as containing the usual text, will also contain the full simulation model, executable code, source code, etc. all in the correct formats in a single Hierarchical Data Format, Version 5 (HDF5)[7] format. Then from the paper one could extract all the necessary information required to replicate the simulation.

While he recognises and accepts the difference between replicability and reproducibility in science he states that the current state of affairs is unsatisfactory

[7] `http://www.hdfgroup.org/`

and a symptom of a lack of exchange between the natural sciences and computer science.

2.6.5 Data Sharing

A similar proposal to *ActivePapers* (above) is made by [Kitchin, 2015] for embedding data with existing formats (pdf, etc). Here the suggestion is that no new formats are required and what is missing is only the toolset to allow easy insertion of different data attachments to existing formats. The authors have developed some tools to this end.

2.6.6 Reproducibility versus Replication

What is needed is a higher level approach to replication that allows us the freedom to interrogate the original ABM and reproduce the model independently in a verifiable manner. This distinction between just replicating an experiment and reproducing an experiment is first mooted in [Drummond, 2009] where all the previous approaches to replication are explicitly criticised for being too narrow in scope.

All the approaches listed above are useful and give more information to experimenters but all have the same weakness. They only allow us to repeat an experiment exactly as it was done originally. Therefore such approaches do not deal with errors that are hidden in the original experiment. It has been shown that occasionally the emergent properties of a simulation can be directly caused by the updating strategy employed [Huberman and Glance, 1993] instead of being intrinsic properties of the system being modelled. If the results of the simulation are artefacts of the updating approach then simply copying or replicating this simulation will give us the same erroneous results. A better approach would be to *reproduce* the results with a different program and different updating strategy. Reproduction of a simulation is considered a stronger result than simply replicating it.

In this thesis such a solution is proposed. In this thesis the production of a high level formal description of an ABSS is proposed that deals with the lack of precision normally associated with ABM descriptions. Alongside this new SU algorithms that can be used in ABM are derived that allow more updating strategies to be used and hence rule out simulation outcomes that are artefacts of any one updating strategy. This allows not just replication but reproducibility.

2.7 Application Areas

2.7.1 ABSS

ABM have been used successfully in a number of different fields of enquiry. [Macal and North, 2009] lists a number of more recent ABM in fields such as:

Anthropology Simulating the Kayenta Anasazi people (circa 1800 BC to circa 1300AD) [Gumerman et al., 2003] to help determine how the society died out;

Biomedical Research CA based models have been used to help with early drug design [Bezbradica et al., 2014] and simulate *Mycobacterium Tuberculosis* [D'Souza et al., 2009] on GPUs

Crime Analysis A model of the effect of policing on civil violence [Epstein et al., 2002b]

Ecology: [Coakley et al., 2006] shows how ABM can be used in social insect modelling (as well as Tissue Modelling and Molecular Modelling)

Epidemic Modelling ABM have simulated the containment of a Smallpox outbreak [Epstein et al., 2002a] and a global pandemic [Parker and Epstein, 2011].

Market Analysis Simulation have been used to predict how the introduction of a Tobin tax [Šperka and Spišák, 2012] would have a beneficial effect on the economy.

Organizational Decision Making An ABM [van Dam et al., 2007] simulating negotiations to achieve global objectives, in this case the location of *intermodal freight hubs*.

ABM offers a bottom-up approach that can be based on more realistic assumptions (fewer simplifications) such as replacing the notion of total rationality in people with the more believable bounded rationality and also dismissing the idea of people having complete information of the market and replacing this with people only having access to local (or incomplete) information. A number of large scale economic ABM are now in use, for example EURASE [Deissenberg et al., 2008] an economic model of the European economy.

The ABM approach allows for a discovery-based process to take place. Emergent system properties are discovered by running the simulation whereas with other approaches to simulation these properties must be known in advance of the simulation being created and then must be explicitly programmed in. The bottom-up approach of ABM allows us to simulate less well understood systems and discover their properties through simulation.

ABM open up simulations to sciences that have until recently never had these tools available to them, such as archeology [Campillo et al., 2012], and provides an alternative and more flexible approach to sciences which have until recently relied on equation based models, such as economics [Troitzsch, 2009].

2.7.2 Computer Games

Computer games are an almost ubiquitous part of modern life [Association, 2015]. They can be very complex simulations containing large numbers of interacting agents with high level behaviours [Nareyek, 2004], [Kleiner, 2005] such as path finding [Graham et al., 2003], [Cui and Shi, 2011], planning sequences of actions to achieve goals [Orkin, 2006], [Orkin, 2005], [Nau et al., 1999] or even learning new behaviours [Tang et al., 2009] all in real time. While the goals of computer game simulation may differ from those of ABSS they contain many similarities.

Sweeney [Sweeney, 2006] divides the game code in the level game loop into three different categories: Simulation, Numeric and Shading. At the moment, when developing a game the numeric code is written in a low level imperative language, usually C but possibly C++. This numeric code handles items that require heavy numeric processing such as the physics simulation, collision detection, scene graph traversal and sound propagation.

The second component, shading, is written in a high level shading language and uses specially designed graphic processing units. These processing units encode many of the shading algorithms at hardware level and allow for very high performance.

The third part of the game is the simulation. The simulation involves a sometimes large number of entities that interact and display behaviours. This simulation part follows the same bottom up approach as ABM. Behaviours occur between individual agents in the simulation and the overall system properties emerge through their interaction. Games use a *clockwork* approach to time with every agent performing an action during every step. One difference is that each step in an ABSS can represent different amounts of real time, for example, in Sugarscape a step where every agent employs the *Combat* rule will represent a different amount of time than one where the *Pollution diffusion* or *Inheritance* rule is employed. In a game each step represents a fixed amount of time, usually less than one twentieth of a second of real time.

There has been some crossover between games and ABSS. For example, the flocking algorithm [Reynolds, 1987] has been commonly applied in games [Shopf et al., 2008, Silva et al., 2010]. The crossover goes in both directions with games technology starting to be applied in ABSS as in [CHŃG, 2008] where games engines are used to help with archaeological simulations.

2.8 Comparing Synchronous and Asynchronous Updating

When a model is simulated it is done in a time-stepped manner. In an ABM this is sometimes known as *clockwork*. In clockwork, each step in the simulation occurs at a fixed time interval and during each step every agent performs some behaviour. In DES time is *event-based*. Event-based simulation events are scheduled to occur at particular times. These events are kept in a time ordered queue. The next event scheduled to occur is taken from the front of the queue and the clock time advanced to match that of the event. DES relies on event-based time while CA and ABSS use clockwork. As we have seen clockwork has been adopted by the major agent toolkits such as NetLogo, Repast, Mason and Swarm [North et al., 2013b, Luke et al., 2005, Berryman, 2008]. It is the clockwork approach that is of interest here.

The synchronous and asynchronous approaches to updating define how we handle multiple events occurring simultaneously or at the same time, i.e. during the same step.

Asynchronous Updating To accurately simulate actions that may occur simultaneously it is sufficient that during each time interval all agents are restricted to acting one at a time in some well defined sequence. That is, this sequencing of actions is in effect equivalent to those actions occurring simultaneously.

Synchronous Updating SU holds that, in contrast to AU, to accurately simulate a world where events can happen simultaneously it is necessary to perform those simultaneous actions concurrently.

We will look at both approaches in more detail and show the issues associated with each.

2.8.1 Asynchronous Updating

The asynchronous approach assumes that there is no *global clock* that forces agents to interact in lockstep [Huberman and Glance, 1993] with each other. We can get this same effect, it is argued, by allowing each agent behaviour to occur in sequence within a step. That is, agents acting simultaneously during a time step can be accurately simulated by just letting each agent act in sequence. While it is conceded that using a particular ordering of behaviours might cause *artifacts* in the simulation it is assumed that these can be overcome by employing a different ordering during each step. In any case overall behaviour can be ascertained by running the simulation multiple times and looking for patterns in the outcomes.

There are two different approaches to asynchrony *Step Driven* and *Time Driven* and they can be further subdivided into [Schönfisch and de Roos, 1999]:

Fixed Direction Line-By-Line The locations in the lattice representing the simulation space are updated in the order they appear in the lattice (usually left to right, top-down). This is a deterministic ordering in that the ordering remains the same during every simulation run. When a simulation employs this approach it is impossible to tell if the emergent properties of the simulation are due in whole or part to the particular ordering used. Properties that are due to the ordering used are known as *ertifacts*. Because this determinacy can cause artifacts it is the least favoured approach;

Fixed Random Sweep The order that is used is determined randomly at the start of the simulation and this order is constant for every step in the simulation. Here the ordering is specific to each simulation run. In any particular simulation run the ordering is deterministic and does not vary from step to step. As each simulation run uses a different ordering this approach allows different runs to be compared and contrasted to see if any emergent properties are artefacts due to the particular orderings employed or not. This is a major improvement over *Fixed Direction LineBy-Line*;

Random New Sweep The order that the agents are updated in is determined randomly at the start of each step (each step uses a different order). This approach is, in a weak sense, fairer than the previous approaches because the ordering is randomly chosen from time step to time step. No one location has a lasting advantage over any other location over time. Artifacts appear to be less common under this approach;

Uniform Choice Each agent has an equal probability of being chosen. If there are n agents, then n agents are chosen randomly during a step. During any single step an agent may not be picked at all or may be picked more than once (in contrast Random New Sweep guarantees every agent is picked exactly once per step);

Exponential Waiting Time This is a *Time Driven* method, all the others are *step driven*. Every agent has its own clock which rings when the agent is to be updated. The waiting times for the clock are exponentially distributed (with mean 1). The probability that an event occurs at time t follows e^{-t} where t is a real number, $t \geq 0$. This is most similar to *Uniform Choice*.

Most of these introduce nondeterminacy into the simulation and so every run of the simulation will use a different set of orderings and will most likely produce different outcomes. Determining the outcome of a simulation might therefore require us to produce multiple runs and compare their outcomes for similar patterns of overall system level behaviour.

There is no established SU algorithm for ABM that overcomes the technical difficulties associated with concurrent agent interaction (for example, two or more agents attempting to move to the same location). As a result of this the asynchronous approach is almost exclusively used in ABM. We will look at SU and its associated problems next.

2.8.2 Synchronous Updating

Adherents of the synchronous approach argue that, as the events are occurring simultaneously, they should be implemented in that manner. This is achieved by letting each agent see the state of the world only as it was at the start of the step. They then perform their behaviours based on that information. Any updates that the behaviours make to an agents state do not get applied until the end of the step, once all agents have completed their behaviours. This makes the updates appear as if they were applied simultaneously.

For simple CA-based simulations where each agent only reads the state of adjacent agents and can only update their own state, synchronous updating is easily implemented. However for an ABSS where behaviour can be complex and agent behaviour may update not just the agent performing the behaviour but also its surrounding agents the synchronous approach is more difficult to implement. If we take the example of simple agent movement, an agent must update the status of two locations, during a move the location where the agent currently resides is vacated while the destination location changes from being vacant to containing the moving agent. If two or more agents try to move to the same location then we have a collision and a decision must be taken as to how this is handled.

Any SU approach must have a sensible way of handling collision detection and since detection of collisions is not enough it must also provide some collision resolution mechanism that resolves the issue properly. These mechanisms become part of the agent behaviour definitions as they will affect how the simulation proceeds.

If we return to the Movement example we can see that there are numerous possible ways of handling agent collisions:

1. One agent is chosen at random to be the winner;
2. The agent who was nearest the destination can be deemed the winner (he would have gotten there first);
3. The strongest agent wins (by some agreed measure of strength).

Even once a resolution mechanism is decided upon there is the question of how the losing agents are death with:

1. The losing agents return to their starting position. In effect they miss a go and remain stationary for this move. A punishment for attempting to move to the wrong destination;
2. Losing agents get another go. Here we assume they would have spotted to imminent collision and moved to a different destination (side stepping the potentially colliding agent).

Each choice will lead to a different dynamic and so needs to be justified in the context of the simulation. Any collision resolution strategy must also be aware of and be able to avoid potential *deadlock* situations. Consider a trading behaviour. Each agent chooses an agent that they wish to trade with and then performs some trade. Under SU, agent A might choose to trade with agent B while agent B wishes to trade with agent C who, in turn, wants to trade with agent A. In this case we have circular waiting and unless there is some method of breaking the circular dependency each will wait for the other agent forever (A waits to hear back from B before replying to C's request, B waits to hear back from C before replying to A's request and C waits to hear back from A before replying to B's request).

AU avoided all these problems by forcing a sequential ordering on actions. Only one agent (under AU) can act at a time so there can be no collisions.

It is the difficulties that SU has in handling these collisions in a sensible manner that has resulted in AU being almost exclusively favoured in ABM. There have been a number of attempted approaches to handling the collision resolution issue and we will look at these proposed solutions in the following subsection.

2.9 Synchronous ABM

Although it has been understood for some time that there are differences between the outcomes of synchronous and synchronous interpretations of ABM [Radax and Rengs, 2010, Caron-Lormier et al., 2008, Huberman and Glance, 1993] and some work has been undertaken in the area of CA to try understand these effects [Radax and Rengs, 2010], there has been little similar work undertaken in ABM [Grilo and Correia, 2011]. As we intend to prove that the asynchronous approach gives incorrect results in ABSS, and produce our own synchronous updating algorithms, we now briefly survey other attempts made to apply synchronous updating to ABM simulations.

2.9.1 Influences and Reaction

In [Ferber and Müller, 1996] a model of situated multi agent systems, *Influences and Reaction*, is derived from *Situation Calculus* [Lespérance et al., 1996]. It extends Situation Calculus to enable it to deal with simultaneous interactions. This is achieved by splitting state into two components. The first represents the state of the system (as per Situation Calculus) while the second represents any *influences* on the system that might cause the state to change. For example, state might represent the location of a particle while the influences might contain the set of forces acting on that particle. The state update function is divided into two separate functions:

React: $State \times Influences \rightarrow State$ Takes in the current state and influences and produces the next (updated) state;

Exec: $State \times Influences \rightarrow Influences$ Takes in the current state and influences and produces the next (updated) set of influences.

This is a theoretical model and it is interesting to note that it is synchronous not by design but because that was what was required to deal with simultaneous interactions. No algorithm to implement this approach is given so it is unclear how collisions (for example, two agents try to move to the same location or take the same resource) can be detected and resolved.

2.9.2 Synchronous IBM

An IBM [Caron-Lormier et al., 2008] in the field of ecology was developed using both synchronous and asynchronous updating so as to compare the results. The model is a simple IBM with three types of agent:

Primary Producer These agents produce resources continuously (each step);

Primary Consumer These agents take resources from primary producers whenever they meet (move to the same location);

Secondary Consumer These agents take resources from primary consumers whenever they meet (move to the same location);

The agents move around a shared landscape at random. It was shown that the outcomes differed for asynchronous and synchronous updating with the changes being more continuous in nature (smoother) in the asynchronous version. The authors also noted that the differences were more pronounced at higher population densities.

However it must be pointed out that although the changes in state (resource updates) were handled synchronously in this model (through the use of *dual state*) the agent movement was handled asynchronously leading to incorrect behaviour. If a primary consumer is visited by two or more primary consumers in a single step then each consumer will see the producers resource level as it was at the start of the step (the updates are hidden until the step end) and so could possibly get more resources from the producer that actually were held by the producer. For example, a producer with a resource level of 100 units could be visited by three consumers during a turn. Assuming a consumer takes 50% of a producers resources then at the end of the step the three consumers will have extracted a combined total of 150 units from the producer. This might also help explain the fact that state changes were less smooth in the synchronous version. A synchronous model of movement is required before this model can be correctly called synchronous.

2.9.3 Transactional Cellular Automata

Following on from [Ferber and Müller, 1996], the *Influences and Reaction* model was used as a basis for a framework that converts ABM into equivalent synchronous CA [Spicher et al., 2010]. The resulting CA is called a *Transactional CA*. There were two main reasons for attempting this:

1. To reduce the bias due to asynchronous updating schemes;
2. The ease the development of parallel ABM implementations.

In the Transactional CA model updates occur using a three step approach [Spicher et al., 2010]:

1. Request *source* cells express their needs to their neighbours.

2. Approval-rejection *target* cells accept or not their neighbours requirements; this decision is taken with respect to an exclusion principle policy (for example, an empty cell is an available target if and only if there is exactly one particle requesting to move to this cell).

3. Transaction *sources* and *targets* separately evolve.

The authors used this to implement *Diffusion-Limited Aggregation*. For example, if two particles (agents) wish to move to the same location then each sends a request to that location in step one. Then the location responds to these requests with either approval or rejection. In the final step the particles update based on the replies they received from the second step.

The model chosen to illustrate this is a very simple model. More work would be required for it to handle more complex interaction types. For example, if interaction required agreement amongst overlapping groups of N agents how are the requests handled? Is it possible to get consistent agreement between all the group participants in a single exchange of messages? What if we have, for example, three agents a, b, c and a wants to attack b while b wants to attack c?. Agent b cannot work out whether to approve or reject in one message passing step. It must wait for a reply to its request to attack c before it can approve/reject a's action. This requires (at least) two steps.

We note its similarity to the synchronous objects model [Kehoe and Morris, 2011] proposed for computer games which instead proposes a multi-pass exchange of messages between objects, thus allowing it to handle complex interactions correctly.

2.9.4 Pedestrian Traffic

In [Burstedde et al., 2001] a stochastic CA is used to simulate pedestrian traffic. The simulation uses a two dimensional lattice where each location can hold a maximum of one agent. Agents are allowed to move one cell in any direction (Moore Neighbourhood) in a single step provided they move to an empty location. Agents have a preferred direction of travel and from this a *matrix of preferences* is produced containing the probabilities of a move in each direction. Updating in this model is synchronous.

> "In each update step, for each particle a desired move is chosen according to these probabilities. this is done in parallel for all particles. If the target cell is occupied, the particle does not move. If it is not occupied and no other particle targets the same cell, the move is executed. If more than one particle share the same target cell, one is chosen according to the relative probabilities with which each particle choose their target. This particle moves while its rivals for the same target keep their position." [Burstedde et al., 2001]

Although the authors do not give the run time complexity of their algorithm they do state that "it should still be way faster than continuous models". The authors note that this model displays oscillations in agent movement and other overall behaviours that do not appear in other models. We also note that they state that synchronous updating is the most widely accepted approach in CA based traffic models.

2.9.5 Turmites

In computer science, a turmite is a Turing machine which has an orientation as well as a current state and a *tape* that consists of an infinite two-dimensional grid of cells. The terms *ant* and *vant*[8] are also used. Langton's ant is a well-known type of turmite defined on the cells of a square grid. Figure 2.8 shows a single *vant's* progress 8342 steps after starting from an empty grid, the turmite (a red pixel) has exhibited both chaotic and regular movement phases.

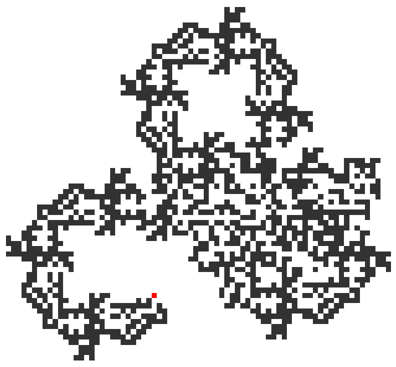
In [Fatès and Chevrier, 2010] a synchronous implementation of the multi-turmite model is presented. It is used to demonstrate that different updating schemes have

[8]vant stands for Virtual Ant.

large effects on simulation outcomes. The model itself is very simple with only one type of agent and only one type of behaviour [Langton, 1986]:

- The *vant* moves on a square lattice where each cell can be blue or yellow; initially they are all blue.
- If it encounters a blue cell, it turns right and leaves the cell coloured yellow.
- If it encounters a yellow cell, it turns left and leaves the cell coloured blue.

Figure 2.8: A 2-state 2-color turmite on a square grid.



They find out that *Deadlock* is possible between agents. With multiple agents competing for the same resources (cell space) we would expect this. What is a more interesting question is why does deadlock *not* occur in the asynchronous system.

While we agree with the authors that updating schemes can lead to different evolutions we do not agree that the updating regime is completely independent of the behaviour definitions. Rather we feel that the updating scheme should be part of the definition. The definition, in this case, is ambiguous as to how certain situations are handled. The updating method refines the definition by imposing its updating regime on this definition. Different updating methods will handle the same situation in different ways. The problem, in this case, is that the definition is incomplete. By not defining what happens when, for example, two agents try to occupy the same location we leave undefined an important aspect of the rule. An asynchronous update handles this by randomly choosing one agent to occupy the location above another (based on the sequencing of moves) but a synchronous rule could use a different strategy (perhaps closest agent to the destination location 'wins'). The implicit assumption that asynchronous updating will be used leads many modellers to ignore the collision resolution issue without justifying that decision.

2.9.6 Synchronous Objects

The Synchronous Objects [Kehoe and Morris, 2011, Kehoe and Morris, 2013] model of programming was developed for Computer Games and not ABM based simulations. However, as we have seen in section 2.7.2, there are many similarities between ABM and computer games simulations. The model used in computer games is based on many interacting heterogeneous agents with overall system properties emerging from their interactions. The Synchronous Objects model was inspired by the Bulk Synchronous Parallel (BSP) [Valiant, 1990] and Active Objects [Briot et al., 1998, Hernandez et al., 1994] approaches to concurrency and adapted for development of simulations (albeit Games Simulations). Its purpose was to make concurrent programming of game simulations easier by removing issues of *Deadlock*, *Livelock* and *Data races*[9]. It also introduces deterministic execution so as to insure debugging is still possible for programmers not experienced in concurrency.

It employs the dual state common in synchronous updating along with message passing similar to that proposed in the *Transactional CA* [Spicher et al., 2010] model. However it allows for multiple passes of message between objects if required for collision resolution (as we saw in the combat example previously). It also assumes each step represents a small fraction of a second in real time (corresponding to frame rates in games of up to 100 frames per second). That is step duration is fixed and all actions must have a duration less than or equal to the step duration length.

The Synchronous Objects model is a precursor to the synchronous algorithms proposed in chapter 6 but the algorithms we are proposing are even simpler than that of the Synchronous Objects model and is more applicable to ABM simulations.

[9]covered in the following chapter.

2.9.7 Combined Synchronous and Asynchronous

In [Sahay et al., 2014] a model that combined agents using synchronous updating with agents that used asynchronous updating was introduced. This model of a supply chain used the synchronous/asynchronous distinction to model the flow of information, or rather the delay in information propagation, within a network of agents. Although the authors state that synchronous interactions are not always easy to implement the interactions in this simulation simple read-dependent interactions. That is an agent interaction can involve an agent reading the state of a neighbour but not writing to the state of a neighbour. These agent interactions are less complex than interactions such as movement.

All of the synchronous updating approaches deal only with simple interactions. The most complex example given in any of the models is movement. It is not clear how, or if, they handle complex actions such as combat or mating where agents can need to exclusively update themselves and other agents in a single step.

In chapter five we will produce synchronous algorithms that can handle the most complex of agent interactions. We demonstrate the algorithms in action in chapter six.

2.10 Summary

We have provided a brief overview of ABM. We examined briefly the different possible approaches to general simulation so as to show the possible alternatives to ABM and place it in context. The approach that a modeller uses will depend on the context.

From our overview of ABM it should now be clear that ABM as an approach to simulation should be given serious consideration if any of the following statements are true.

1. The primarily interest is in emergent properties;

2. There are no solvable (computable) equations that describe the overall system properties;

3. Any equations we have rely on simplifying assumptions that we cannot or do not wish to stand over;

4. Of interest are the individual behaviours of agents within the system;

5. The individual behaviours are high level behaviours and the agents are proactive.

An ABSS will have the following properties (section 2.2):

1. Heterogeneity;
2. Autonomy;
3. Local Interactions;
4. Explicit Space;
5. Bounded Rationality.

If a model does not have these properties then it is not suitable for ABSS.

During each step of a simulation all agent interaction takes place as if simultaneously in time. We note that there are two different approaches to implementing this condition, known as the *synchronous* and *asynchronous* approaches. The asynchronous approach proceeds by executing each interaction in a sequential order. to remove any simulation artefacts that may occur due to the ordering used we randomise the order for each step. This is currently the preferred approach in ABSS.

Asynchronous has become the prevalent updating method used in ABSS. The ability to execute a sequence of agent actions in a random order, an essential part of asynchronous simulations and forms part of *StupidModel* [Railsback et al., 2005], a suite of models designed to test the suitability of any toolkit for ABM development. Many assume that asynchronous is more realistic for real world based simulations [Caron-Lormier et al., 2008], [Cornforth et al., 2005], [Newth and Cornforth, 2009]. However [Fatès, 2013] states that the asynchronous approach only makes sense if the interactions are instantaneous, something that is seldom true in the real world. These contradictory conclusions drawn from different published sources indicate that more research into the effects of AU and SU are needed.

The alternative approach is favoured more in CA based simulations. Here simultaneous interactions within a step are achieved by allowing agents to only see the state of the world (neighbouring agents and locations) as it appeared at the start of the step. All interaction outcomes are calculated based on these values. Any updates that occur during a step (as a result of agent interactions) are hidden and only applied at the end of the step (as if simultaneously).

The differences between these two approaches and how they affect the overall ABM properties are at the heart of this thesis. There has been some work on the differences between synchronous and asynchronous ABM updating but much less work on how to extend synchronous updating to complex ABM interactions. It has been shown that even in simple CA-based simulations the synchronous and asynchronous interpretation of identical models can lead to completely different results [Huberman and Glance, 1993, Schönfisch and de Roos, 1999], especially in higher density simulations [Caron-Lormier et al., 2008] and asynchronous is generally less restrained [Cornforth et al., 2005] in behaviour outcomes. While [Ruxton and Saravia, 1998] says that the approach taken should be chosen with care and specific regard the process being simulated, in [Caron-Lormier et al., 2008] it is stated that most modellers choose an approach based on personal experience rather than based on evidence of its correctness.

Conway's *Game of Life* has been implemented as an asynchronous CA [Lee et al., 2004] but it has further been proven [Nehaniv, 2003] that any synchronous CA can be implemented as an asynchronous CA. This is achieved through local synchronisation instead of global synchronisation. Each cell in the CA keeps track of its previous state and its current state and when chosen to update will only update itself if its neighbours are all operating on the same timestamp. So while this allows asynchrony every cell is always either one time step behind, one time step ahead or at the same time step as its neighbours. That is, it is still synchronised but only locally. This, it could be argued, is an example of synchronous updating without a *Global clock*.

The work that has been done concentrates only on the simplest of ABM interaction: basic movement across locations. More complex interaction types, such as those in Sugarscape have been ignored. Modellers seem to have uncritically accepted the arguments presented for asynchronous updates.

We will examine the assumptions behind AU in Chapter Six and propose new synchronous algorithms that do not suffer from the flaws inherent in the asynchronous approaches.

In the next chapter we survey the issues introduced by concurrency. We will show how to measure the performance of an algorithm (concurrent and sequential) and show how concurrency is currently employed in ABM.

Chapter 3

Concurrency

3.1 Introduction

Concurrent computing is a form of computing in which programs are designed as collections of interacting computational processes that may be executed in parallel [Ben, 2006]. If we call a sequential program a *process* then we can define a *concurrent program* as a set of processes that are executed in abstract parallelism. In a minority of cases a concurrent program is composed of completely independent processes and these can all be run in parallel without any problems. Such problems are termed *embarrassingly parallel*. However, in the vast majority of cases concurrent programs cannot be broken into completely independent processes. There will exist interdependencies between these processes that constrain the relative speeds of their execution.

This extra complexity, brought about by the introduction of concurrency [Sutter and Larus, 2005, Lee, 2000], means that it is difficult to produce correct solutions even for small problems. The processes that make up a concurrent program need to communicate with each other in order to share data. There are two approaches to communication between processes: *shared memory* and *message passing* [Klaiber and Levy, 1994].

All concurrent programs must deal with the issues of contention and communication. Concurrency is only introduced to reduce overall time required by a computation to complete (that is, *speed-up* a piece of code). The speed-up that comes from changing a serial computation into a functionally equivalent concurrent computation will depend on how much concurrency can be introduced and the dependencies that exist between the concurrent tasks. Generally we want to achieve as much speed up as possible and this requires us to maximize opportunities for parallelization. Optimal speed up is linear but the achievable speed-up will vary from computation to computation. Here we ignore the possibility of *superlinear*[1] speed increases that can be caused by, for example, the concurrent algorithm being more efficient than the original sequential algorithm it is replacing. We address these issues in more detail in the following sections.

3.2 Issues in Concurrency

Concurrent code is more difficult to produce than the functionally equivalent sequential code. The first difficulty is trying to define the constituent processes. Once we have identified a set of processes we must identify all the dependences that exist between the processes and decide on the appropriate parallelization strategies (For example, the *producer-consumer* pattern [Downey, 2005]).

Dealing with the dependences between processes can result in contention and communication problems. In shared memory models of concurrency we can have *data races* [Breshears, 2009]. This is where one or more processes are reading data while simultaneously one or more other processes are updating that same data. In order to prevent this, some form of *mutual exclusion* is required. *Mutual Exclusion* is usually implemented by locking [Downey, 2005, Herlihy and Shavit, 2012]. A process can use a *lock* to guarantee exclusive access to data. A lock can only be held by one process at a time. We associate a particular lock with a specific piece of data and then we ensure that any process that wants access to that piece of data must hold that lock before they are allowed access to the data. Once a process has finished with the data it must relinquish the lock so as to allow other processes to acquire it. The most common form of lock is the *semaphore*. The semaphore was first introduced by Dijkstra [Dijkstra, nd] and has simple properties. A semaphore is an abstract data type that contains an integer value and has two operations:

Wait If the semaphore's value is > 0 then subtract 1 from it otherwise suspend the calling process;

Signal If there are processes suspended on this semaphore wake up one of the suspended processes otherwise add 1 to the value held by the semaphore[2].

[1]A speedup of more than p when using p processors.

[2]Originally these two operations were called P and V, respectively, from the first letters of the Dutch words for *Signal* and *Wait*

When used incorrectly, locks can cause deadlock: where two or more tasks are waiting for each other to release a resource. Absence of deadlock is an important *safety* property that must be shown for any program that uses locking. Deadlock can arise if all of the following conditions hold [Breshears, 2009]:

Mutual Exclusion One or more processes holds a resource in a non sharable mode (hold a lock on this resource);

Resource Holding A process holds one resource while simultaneously waiting to lock another resource;

No Preemption Locks can only be released by the processes holding them;

Circular Wait A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes such that each process in the set is waiting for a resource held by another process in the set.

A programmer implementing locks needs to show that at least one of the above properties cannot hold in order to prove that a program is deadlock free.

Another issue is *Livelock* which is similar to deadlock except the processes involved in livelock constantly change state with regard to one another but with no process progressing. Consider the example of two people trying to cross a rope bridge in opposite directions. When they meet in the middle they are blocked. Both go back the way they came but when they exit the bridge they both now see it is clear and both again try to cross, again meeting in the middle. As they continue this back and forth procedure both are doing work but neither is actually progressing.

We also need to show that fairness, in some form, exists within concurrent programs. Fairness specifies how contention is resolved. Generally we must ensure that it is resolved in such a way that everyone will eventually get access to the resources they need. There are different recognised types of fairness [Ben, 2006]:

Weak fairness If a process continuously makes a request, eventually it will be granted;

Strong fairness If a process makes a request infinitely often, eventually it will be granted;

Linear waiting If a process makes a request, it will be granted before any other process is granted the request more than one;

FIFO (first-in first-out) If a process makes a request, it will be granted before that of any process making a later request.

The ability to prove properties, such as *fairness* and *absence of deadlock*, of concurrent programs is both complex and difficult. Concurrent programmers must learn how to handle these new types of error that are just not present in sequential programming.

A final issue we face is that of debugging concurrent programs. Execution of concurrent code is generally non-deterministic. Each run, or execution, of the same code can, and generally will, result in a different scheduling or interleaving of the instructions within the processes. An error in the code may result in an incorrect output only under a certain scheduling. Since there is no reliable way of repeating the exact relative speed of processes on demand we can find ourselves unable to repeat the conditions that lead to specific errors. Without this repeatability the correction of errors in concurrent code is more difficult than that for sequential code.

3.3 Measuring Concurrent Performance

3.3.1 Concurrency Metrics

There are many different reasons for using concurrency in an algorithm. Some of the possible reasons include:

1. Reducing the time it takes to get a result (*latency*);
2. Increasing the rate at which a series of results is produced (*throughput*);
3. Reducing the power consumption of a computation;
4. Increasing the amount of data our algorithm can handle (*weak scalability*);
5. Simplifying the algorithm design process;
6. Reduce the cost of the platform required for the computation.

How we choose to measure the performance of a concurrent system depends on our reasons for using concurrency. Central to many of these reasons are the ideas of latency, throughput, speed-up and efficiency.

Latency

This is the time taken to complete a task. It is measured in units of time. Obviously the time required will vary depending on the processor speed. We keep time independent of any one particular processor by, for example, using complexity classes (see section 3.3.4).

Throughput

Throughput measures the rate at which a series of tasks can be completed. It is measured in units of work per time unit.

Although latency and throughput are clearly related they remain two distinct measurements. For example, a task T with a latency of one (time unit) can vary its throughput. If we compute two such tasks in parallel then the latency of one results in a throughput of two tasks per time unit. This relationship is quantified through Little's formula [Little, 1961]. If throughput is R and latency is L then concurrency, C, is $R \times L$.

$$C = R \times L \tag{3.1}$$

This shows how concurrency can hide latency and maximise throughput. In practice consideration will have to be given to any required synchronisation between the tasks. The underlying processor architecture (issues such as cache size) also has a role to play and must be taken into account.

Speed Up

Speed up in a concurrent system generally depends on the number of processors, or workers, at our disposal. Ideally the addition of more workers will result in more speed up. Therefore we measure speed up in terms of processors employed. Speed up using P processors (S_P) is defined as the latency of the task when completed using one worker (L_1) divided by the latency of the task when completed using P workers (L_P)

$$S_P = \frac{L_1}{L_P} \tag{3.2}$$

Efficiency

Efficiency is a measure of how well we use our workers. It is calculated as the speed up achieved using P workers divided by P.

$$\text{Efficiency} = \frac{S_P}{P} = \frac{L_1}{P \times L_P} \tag{3.3}$$

In an ideal world we would get an efficiency of one but this is unusual. When we divide a task between multiple workers we introduce an overhead. This overhead includes the communication that must occur between the various workers allowing them to coordinate their work on the tasks. An efficiency of one is known as linear speed up. Occasionally we get a *superlinear* speed-up (> 1) which can be caused by [McCool et al., 2012]:

1. The parallel algorithm making better use of cache memory (cache memory is at least an order of magnitude faster than non cache memory;
2. The parallel algorithm may be more efficient than the original sequential algorithm.

Absolute speed-up is achieved by comparing the original sequential algorithm latency (L_1) against the new concurrent algorithm. *Relative speed-up*, in contrast, is attained by comparing the new concurrent algorithm run with one worker as the baseline L_1. Since concurrent algorithms run with only one worker tend to run slower than the equivalent sequential algorithm absolute speedup is considered the more useful measurement.

3.3.2 Strong Scalability: Amdahl's Law

Strong scalability is measured by showing the speedup attained by the introduction of more workers to a problem whose size is fixed. The limits of strong scalability are shown by Amdahl's Law [Amdahl, 1967].

For any given problem A with a latency of L_1 we can divide this latency into two separate components. The time spent on the nonparallelisable part (L_{seq}) and the time spent on the parallelisable part (L_{par}). We can see that:

$$L_1 = L_{seq} + L_{par} \tag{3.4}$$

L_{seq} is, by definition, unaffected by the introduction of more workers so

$$L_P \geq L_{ser} + \frac{L_{par}}{P} \tag{3.5}$$

The reason we use "$\geq$" and not "=" in this equation is that there may be extra overhead introduced to coordinate the activities of the concurrent components. Speedup can now be written as:

$$S_P = \frac{L_1}{L_P} \tag{3.6}$$

Amdahl showed that:

$$S_P \leq \frac{1}{f + \frac{(1-f)}{P}} \tag{3.7}$$

The maximum possible speedup, with infinite workers ($P = \infty$) becomes:

$$S_\infty = \frac{1}{f} \tag{3.8}$$

If, for example, only 1% of the code remains sequential we get a maximum possible hundredfold speedup. Amdahl's Law puts hard limits on what speedups are possible using concurrency. However we need to note that:

1. It is a theoretical based limit that assumes nonsuperlinearity. A concurrent algorithm could still achieve better speedups through improved utilisation of the underlying hardware;
2. It ignores the possibility that a concurrent algorithm might be many times faster than the original sequential algorithm. In effect it measures relative speedup;
3. It assumes the problem size is fixed.

3.3.3 Weak Scalability: Gustafson-Barsis' Law

Strong scalability assumes that the problem size remains fixed in size even as we add new workers. This is seldom the case. Concurrency is more often employed to help us tackle bigger problems than those we have been solved before, for instance larger simulation spaces. Modern games consoles are not used to run old 8-bit arcade games at faster frame rates but instead to run bigger, more complex and more realistic games. As the hardware improves it changes the problem set we wish to solve.

Gustafson-Barsis' Law [Karbowski, 2008] is used to show the limits of *weak scalability*. *Weak scalability* is defined as the measure of speedup by increasing the problem size. Although known as *weak scalability* it is just as useful (or perhaps even more so) a measure as *strong scalability*. We demonstrate *weak scalability* limits using the *Work-span model*. Again we ignore the possibility of superlinear speedup. The *Work* of an algorithm is defined as the amount of time a serial execution of the algorithm would take.

$$Work = T_1 \tag{3.9}$$

Span is defined as the amount of time an algorithm would take on an ideal machine with an infinite number of workers. This is equivalent to the critical path through the algorithm. Span is also known as the *step complexity* or *depth*.

$$Span = T_\infty \tag{3.10}$$

Speedup is calculated as:

$$S_P = \frac{T_1}{T_P} \tag{3.11}$$

$$T_P = \frac{T_1}{S_P} \geq \frac{T_1}{P} \tag{3.12}$$

Substituting terms we can deduce that:

$$S_P \leq \frac{T_1}{\frac{T_1}{P}} (= P) \tag{3.13}$$

Also as $T_P \geq T_\infty$ we can deduce that:

$$S_P \leq \frac{T_1}{T_\infty} (Work\, over\, Span) \tag{3.14}$$

This provides an upper bound on speedup. We can also provide a lower bound using Brent's Lemma [McCool et al., 2012].

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty \tag{3.15}$$

We can see from this that imperfectly parallelisable work will always take T_∞ (the critical path length) no matter how many workers are available. The remaining work (that which is perfectly parallelisable) is $T_1 - T_\infty$ that is total work minus the imperfectly parallelisable work. As this is perfectly parallelisable we can simple divide this by the number of available workers to see how long it takes. Therefore:

$$S_P \geq \frac{T_1}{\frac{T_1 - T_\infty}{P} + T_\infty} \tag{3.16}$$

If it turns out that the span is much smaller than the total work then we can simplify further:

$$S_P \geq \frac{T_1}{\frac{T_1}{P} + T_\infty} \tag{3.17}$$

In practice this simplified equation works well [McCool et al., 2012] but in all cases we are assuming:

1. There is no speculative work: that is, work that we undertake before knowing for certain whether it will actually be needed. For example, faced with an *if-else* where we have spare processor power we could decide to do both branches before we have worked out which will be taken;

2. Scheduling of tasks to workers is *greedy*[3][Malik et al., 2013];

3. Memory bandwidth is not a limiting resource.

Overall the Gustafson-Barsis' Law (equation 3.17) give a more optimistic view of what can be achieved with concurrency. It is also more in keeping with general practice as concurrency is usually used to increase the size of problem we can handle rather than speedup a fixed size problem.

In chapter six we give *speed-up* and *efficiency* measures for our new synchronous algorithms derived in chapter five.

3.3.4 Asymptotic Analysis

Asymptotic analysis [Greene and Knuth, 2007] is an approach to analysing algorithms to determine their theoretical space-time complexity. Space and Time Complexity are defined as follows:

Time Complexity How the completion time of an algorithm grows with the problem size. That is, what is the relationship between the size of input to a

[3]A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

computation and the amount of time it will take that computation to run to completion;

Space Complexity How much memory an algorithm uses with respect to the size of its input data set.

In this section we will refer to *time complexity* but the general arguments and application applies to space complexity equally. When measuring complexity we generally measure:

Best Case (Lower Bound) The fastest possible time (smallest amount of memory usage) taken for an algorithm to run to completion;

Worst Case (Upper Bound) The slowest possible time (largest amount of memory usage) taken for an algorithm to run to completion;

Average Case The average time (memory usage) taken for an algorithm to run to completion. This is the most useful measure as the best and worst cases may only occur very infrequently. It may be, for example, that the best case time is very good but the likelihood of the circumstances causing the best case to occur are very low.

Asymptotic analysis provides measures for the best and worst cases. Other methods must be used if we want to compute the average case.

Asymptotic analysis provides general measures that are independent of any specific implementation issues. For example, the measure should be independent of the speed of any actual processor as all processors run at different speeds and these speeds change from year to year. It achieves generality by ignoring constant differences in the estimates (constant differences in speed can be overcome by waiting for faster processors to arrive). Under this scheme a measure of $10n^2$[†] is considered to be equal to n^2. That is, we ignore the constant multiplier which we assume to be insignificant. Here 'equal' is defined to mean that it belongs to the same *complexity class*. In computational complexity theory, a complexity class is a set of problems of related resource-based complexity. Once this principle is accepted it becomes clear that we can also ignore all lower order powers in an expression. For example for any constants a, b, c, d:

$$a \times n^3 + b \times n^2 + c \times n + d$$

[†]The n variable refers to the input data set size. We will stick to this convention here.

$$< a \times n^3 + b \times n^3 + c \times n^3 + d \times n^3$$

$$= (a + b + c + d) \times n^3$$

$$= n^3 (ignoring\, constants)$$

More formally, if two algorithms belong to two different complexity classes then there exists input sets of some size such that one of the algorithms will always be faster than the other regardless of the speed of the processor [Greene and Knuth, 2007]. The main complexity classes of interest to us, listed from best to worst are [4]:

C **or** 1 Constant time required, regardless of input set size;

$\log n$ Logarithmic time. Here $\log$ is assumed to be $\log_2$. However, as different logarithmic bases differ only by constant amounts, the base is not important;

n Linear. Algorithm size is directly proportionally to input set size;

$n \log n$ known simply as "nLog n". Algorithms in this complexity class are viewed as acceptably fast;

n^C Polynomially bounded. There are separate complexity classes for each value of C. (n^2 is faster than n^3 is faster than n^4 et cetera);

C^n Exponential. Here C is some constant. These algorithms display unacceptable (exponential) rates of growth of time requirements as the data set increases.

There are other complexity classes worse than exponential. All the algorithms worse than polynomially bound, known as non-polynomially bound are considered bad[5].

Formal Definitions

The three main bounds measured by asymptotic analysis are formally defined as follows [Knuth, 1976]:

Upper bound or worst case analysis: $O(-)$ [6]:

$$O(f(x)) = \{g(x) \mid \exists\, x_0 > 0, c > 0\, \forall\, x > x_0 : g(x) < c \times f(x)\} \qquad (3.18)$$

[4]A full list is available at: `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`

[5]To paraphrase the rule from the $\alpha\beta$ search algorithm if it is bad enough to belong to a non-polynomial bound complexity class then there is probably not much point spending any time working out just how bad it is.

[6]also known as Big-"O"

Lower bound or best case analysis: $\Omega(-)$

$$\Omega(f(x)) = \{g(x) \mid \exists\, x_0 > 0, c > 0\, \forall\, x > x_0 : g(x) > c \times f(x)\} \tag{3.19}$$

Bound above and below: $\Theta(-)$

$$\Theta(f(x)) = \{g(x) \mid \exists\, x_0 > 0, c_1 > 0, c_2 > 0\, \forall\, x > x_0 : c_1 \times f(x) < g(x) > c_2 \times f(x)\} \tag{3.20}$$

As we can see $O(-)$ provides an upper bound or worst case analysis, $\Omega(-)$ provides a best case analysis and $\Theta(-)$ is both measures combined into one complexity class. Of these three measures Θ is the most useful as it combines both other measures in the same complexity class. Providing a Θ value is not always possible as it requires proving that both the upper and lower bounds are in the same complexity class.

Issues

While asymptotic analysis has proven to be very useful in practice there are issues we need to beware of. First even if we produce best and worst case measures for an algorithm this does not tell us how likely the best or worst case scenarios are. For example, *Quicksort* is O(n^2) [Hoare, 1961] but the expected completion time is actually $n \log n$ as the worst case scenarios giving n^2 are so unlikely.

Another issue is the ignoring of constants. If the constants being ignored are sufficiently large then they can still have a large impact on the algorithm completion time. Sufficiently large is determined by the problem input set size we are dealing with so it varies from situation to situation. It is good practice to back up asymptotic analysis with some benchmarks that demonstrate how the algorithm translates on actual processor architectures.

We follow this procedure by providing $\Theta(-)$ measurements for our new algorithms in chapter five and then following this up with benchmarks in chapter six to indicate how well they operate on current processor architectures.

3.3.5 OpenMP

OpenMP [Chapman et al., 2007, Dagum and Menon, 1998] is an Application Programmer Interface (API) for portable multithreaded application development that was created in 1997. OpenMP is designed for shared memory multiprocessors and so is perfectly suited to modern multi-core processors. It emphasises a incremental approach to parallelism and is often used to take existing code and parallelise it *in situ*.

OpenMP provides a platform independent set of compiler directives in the form of *pragma* directives, function calls and environmental variables that explicitly instruct the compiler how and where to use parallelism. It is designed to be easy to use in that the programmer does not need to worry about creating, synchronising, load balancing or destroying threads [Akhter and Roberts, 2006]. The more recent version of OpenMP [OpenMP Architecture Review Board, 2013] now includes tasks [Ayguadé et al., 2009] and the ability to offload concurrency to coprocessors (such as a GPU) [Bertolli et al., 2014].

It is possible to give a scripting language, such as NetLogo, access to the OpenMP C API but there are two issues with this approach. First the OpenMP library would need to become part of the language infrastructure and this could make it unwieldy. Even if this was not a problem there is a greater issue. OpenMP operates at the code level. The coder or model developer writer would still need to explicitly decide which parts of their code could be parallelized. While the process would be easier than using threads directly it is still an added burden for the modeller. They still must reason about whether their code can deadlock and explicitly spot all possible data races.

In implementing our new algorithms we employ OpenMP for concurrency (see chapter six).

3.4 Heterogeneous Processing in Agent Based Models

Early work by [Perumalla, 2006] showed that DES style execution is possible on GPUs and could give a 2 fold improvement over sequential code but it was also noted that much work was required to convert DES code from a multicore based model to a streaming multiprocessor model. This work was followed up in 2008 [Perumalla and Aaby, 2008] where the GPU performed 2 to 3 orders of magnitude faster than that achievable using Agent Based Model (ABM) toolkits (which were sequential) when benchmarked on three homogeneous models (Mood Diffusion, Game of Life and Schelling Segregation). It was noted [Perumalla and Aaby, 2008] that the GPU code gained speed at the cost of modularity and reusability when compared to the ABM toolkits. Programability was also more difficult and there were worries about model correctness - artificial biases can be introduced for performance optimisation.

[Lysenko and D'Souza, 2008] also identified a GPU implementation of version 16 of the *StupidModel* benchmark [Railsback et al., 2005] (again a homogeneous model) that was an order of magnitude faster than ABM toolkit implementations. They demonstrated some data based algorithms (Single Instruction, Multiple Data (SIMD)) suitable for ABM and felt that the area showed promise.

Richmond [Richmond et al., 2009] developed an agent based GPU framework that can be scripted using a C++ like scripting language and used this to benchmark the flocking algorithm [Reynolds, 1987]. The results were promising but no direct comparisons were made with the equivalent sequential or multicore code.

Park [Park and Fishwick, 2010] showed that General-purpose computing on Graphics Processor Units (GPGPU) could be used for simulations where the time steps were irregular (similar to the DES approach) and developed a framework that gave an order of magnitude speed up but also gave results that were approximate and result in numerical error, thus confirming Perumulla's worries over the same issues.

More recently again, Seok [Seok and Kim, 2012] compared homogeneous cellular models implemented both on multicores using OpenMP and GPUs using Compute Unified Device Architecture (CUDA). The GPU was 2.5 times as fast as a quad core Central Processor Unit (CPU) when simulating the largest model. He made no reference to the difficulties in developing for GPU or CPU. Around the same time the Flame ABM [Richmond et al., 2010] framework was developed. It was designed to be easy to program and hides as much of the CUDA API from the user as possible. It describes agents and messages using X-machines [Holcombe, 1988]. X-machines are designed using formal techniques and encoded using the X-Machine Markup Language (XMML) while messages are described in eXtensible Markup Language (XML). Message implementation is still in the low level CUDA C subset with some extra restrictions on what the code is allowed to do (e.g. functions can only output one message). The GPU version of Flame shows a 250 times speed up over the non GPU version. Although no direct comparisons between the speed of Flame and other ABM toolkits were found it is hard to imagine that it is not many times faster. For best results it is recommended that Flame is used for homogeneous models with many agents and simple messages [Richmond et al., 2010]. For many ABSS these conditions do not hold so actual attainable speed-up will be lower but the authors provide no details as to how much lower they will be.

3.5 Summary

Concurrency introduces extra complexity [Sutter and Larus, 2005, Lee, 2000]into program development (such as *deadlock, livelock* and *nondeterminacy*) but these difficulties can sometimes be offset by *speed-up*. The maximum attainable *speed-up* is outlines in Amdahl's and Gustafson-Barsis' Laws [Amdahl, 1967], [Karbowski, 2008]. In ABSS Gustafson-Barsis' Law is more applicable as concurrency is used to allow modellers to produce larger and more complex simulations. We can measure the theoretical space-time complexity of algorithms using asymptotic analysis.

While work has been undertaken in parallelising ABSS before, this work suffers from two deficiencies. The first is that this work tends to parallelise only the simpler agent interaction types (for example, the *Stupidmodel* benchmark [Lysenko and D'Souza, 2008]). It is difficult therefore to judge how their approach to concurrency will transfer to the more complex ABSS.

In chapter six we present new algorithms for synchronous updating in ABSS that can cope with the most complex agent interaction type and use asymptotic analysis to demonstrate that their space-time complexity belongs to the same complexity classes as asynchronous updating algorithms. We also prove that our new algorithms are *deadlock* and *livelock* free and that they are deterministic in execution.

The second problem is that results tend not to be repeatable or even when they are repeatable (for example, through availability of original source code) they cannot be easily compared against other approaches [Edmonds and Hales, 2003].

What is required is a precisely defined benchmark that contains a wide range of interaction types of varying degrees of complexity. Producing such a benchmark requires tackling the replication problem in ABSS and the definition of a suitable range of agent behaviours.

In the next chapter we look at *Sugarscape*, a well known ABSS that contains a wide range of agent interaction types. These vary in complexity from the very simple to the very complex. In chapter five Sugarscape is used as an example to show how to formally specify an ABM. The formally defined version of Sugarscape contains the precision required to allow its use as a benchmark. In appendix A this benchmark is used to test our new SU algorithms in chapter six to demonstrate that, in practice, they achieve the performance predicted by the theoretical complexity measures we derive in chapter six.

Chapter 4

Sugarscape

4.1 Sugarscape's Place in ABM

Sugarscape is the simulation that demonstrated how ABM could be applied to the social sciences. It first appeared in the book *Growing Artificial Societies*[Epstein and Axtell, 1996] and it remains influential today. Almost every major simulation toolkit (Swarm, Repast, Mason and NetLogo) [Railsback et al., 2006a, Berryman, 2008, Inchiosa and Parker, 2002] comes with a partial implementation of Sugarscape that demonstrates that toolkit's approach to simulation. Different concurrency researchers [Lysenko and D'Souza, 2008, D'Souza et al., 2007] have used the Sugarscape model as a testbed for benchmarking different approaches to parallelising ABMs.

Although the rules of Sugarscape have been defined in [Epstein and Axtell, 1996] there is no general agreement on their exact meaning [Bigbee et al., 2007, Gilbert, 2014]. These difficulties hamper the ability of researchers to:

1. properly compare their approaches;
2. provide complete implementations of Sugarscape;
3. replicate their results.

Originally the rules were stated with an explicit assumption that the underlying implementation would be sequential. Concurrency was simulated through randomisation of the order of each rule application on the individual agents. Models that follow this regime are termed asynchronous although sequential is a more accurate term.

Sugarscape was originally implemented in twenty thousand lines of code and is a lattice based simulation.

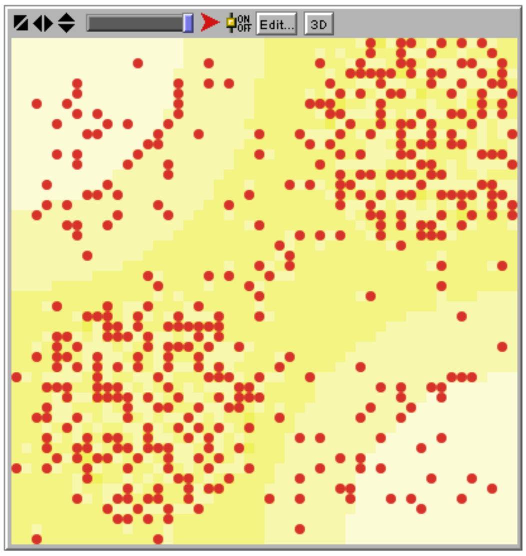
4.2 The Definition of Sugarscape

There are thirteen different rules in Sugarscape [Epstein and Axtell, 1996]. The rules, although stated simply, are lacking in clarity. Most of the rules can only be fully understood after a careful reading of the book's text and even then some aspects of the rules remain unclear.

There are gaps in the Sugarscape definitions, many pertaining to how rules should be extended to cope with multiple resources. In some cases we can be confident as to how these gaps can be filled in but in other cases we have been forced to pick one out of a competing pool of alternatives, and in these cases we chose what we believed to be the simplest solutions.

4.3 The Lattice

Figure 4.1: Sugarscape



The lattice consists of a 40 by 40[1] two dimensional toroidal grid. Each location on the grid can hold at most one agent and has a number of associated properties:

1. An N-dimensional vector of resource levels;

2. An N-dimensional vector of resource capacities;

[1]In theory any size lattice can be used but in practice size is determined by available computing resources

3. An M-dimensional vector of pollution;
4. An M-dimensional vector of pollution fluxes.

On initialisation these properties are set to random values from within a specified range. Figure 4.1 shows a graphical representation of the lattice where the red dots indicate agents and the amount of sugar at a location is indicated by yellow shading.

4.4 The Agents

Locations represent the space that the mobile agents move on, have the properties listed above and are stationary. The agents can change location and have different attributes to the locations. These are initialised to random values within set ranges. These properties include:

Vision The number of locations that the agent can see (or sense) in each of the four cardinal directions. Sugarscape uses the *von Neumann* definition of neighbours (North, South, East and West only);

Metabolism N-dimensional vector indicating the resources an agent consumes during each step. Sugarscape was developed with a maximum of two resource types, called *sugar* and *spice*. An agent who does not have the resources required by its metabolism will die;

Resources N-dimensional vector of resource allocations. This indicates the amount of resources that the agent has collected but not yet consumed (through Metabolism). There is no upper limit on the amount of resources an agent can hold;

Age The number of turns that the agent has existed;

Maximum Age The age at which the agent dies. An agent may die before it reaches its maximum age if it runs out of resources or is killed during *Combat*;

Gender The gender of the agent (male/female);

Cultural Tags P-dimensional bit vector. Culture is determined simply by the ratio of 1's to 0's held in this vector. P is defined to be an odd number to ensure there can never be an equal amount of 1's and 0's. A specific value for P is undefined;

Immune System Q-dimensional bit vector. Diseases are represented by bit strings (vector). An agent is immune to a disease if that disease bit string is a substring of its immunity vector.

4.5 The Rules

4.5.1 Growback

Sugarscape Growback$_\alpha$ At each Lattice position, sugar grows back at a rate of α units per time interval up to the capacity at that position.

This is the simplest rule and is employed in every rule combination used in Sugarscape. We found no issues with its definition. Although this rule is trivial is is an essential component of any Sugarscape simulation.

4.5.2 Seasonal Growback

Seasonal Growback $S_{\alpha,\beta,\gamma}$ Initially it is *summer* in the top, half of the Sugarscape and *winter* in the bottom half. Then every γ time periods the seasons flip - in the region where it was summer it becomes winter and vice versa. For each site, if the season is summer then sugar grows back at a rate of α units per time interval; if the season is winter then the grow back rate is α units per β time intervals.

Seasonal growth is defined only for a single resource (sugar). The simplest way to extend this rule for multiple resources is to assume that the same constant defining the Winter grow back rate is applied to all resources in the same manner.

4.5.3 Pollution Formation

Pollution Formation $P_{\alpha,\beta}$ When sugar quantity s is gathered from the Sugarscape, an amount of production pollution is gathered in quantity αs. When sugar amount m is consumed (metabolised), consumption pollution is generated according to βm. The total pollution on a site at time t, p^t, is the sum of the pollution present at the previous time, plus the pollution resulting from production and consumption activities, that is, $p^t = p^{t-1} + \alpha s + \beta m$.

There is a separate Pollution Formation rule definition that handles arbitrary amounts of pollution types but we did not specify this more complex version of this rule. Multiple pollution types are not, to the best of our knowledge, employed by any Sugarscape implementation. Indeed it is not clear if there would be any benefit to doing so.

4.5.4 Pollution Diffusion

Pollution Diffusion D_α

- Each α time periods and at each site, compute the pollution flux the average pollution level over all its von Neumann neighbouring sites;
- Each site's flux becomes its new pollution level.

We note here that the introduction of the "flux" term is purely an implementation convenience (for sequential implementations). The fact that this rule is explicitly defined with concurrent semantics indicates that there is no ideological objection to implementing the rules synchronously. It is possible that it was only the lack of any practical algorithms for the more complex rules prevented this approach being used throughout the simulation.

4.5.5 Replacement

Replacement - $R_{[a,b]}$ When an agent dies it is replaced by an agent of age 0 having random genetic attributes, random position on the Sugarscape, random initial endowment, and a maximum age selected from the range $[a, b]$.

When an agent is replaced by a new agent, does this replacement immediately consume the resources at its starting location as part of the replacement process or does it wait until its first move to do so? Either option is valid. We have assumed that it does not consume the resources on creation solely on the basis that this is the simplest solution.

Although the idea of "death" is introduced early in Sugarscape no stand alone rule is defined for it. It is defined only as part of the *Inheritance* and *Replacement* rules even though it is used independently of either rule. We have introduced a separate *Death* rule that follows the description of death given in the book. This is a case of an entire missing rule. A single resource agent dies when it reaches maximum age or it runs out of *sugar*. For multiple resources we extended this by assuming that an agent dies if it reaches maximum age or if it runs out of any one resource.

4.5.6 Movement

Movement - M

- Look out as far as vision permits in each of the four lattice directions, north, south, east and west;
- Considering only unoccupied lattice positions, find the nearest position producing maximum welfare;
- Move to the new position
- Collect all resources at that location

Metabolism costs are deducted during each simulation step but this fact is not included in any of the rules so it is unclear what rule, if any, is responsible for this. Rather than randomly picking a rule and adding metabolism deduction to the rule we created a separate rule that handled both the incrementing of the step count and deduction of metabolism costs from each agent.

4.5.7 Agent Mating

Agent Mating S

- Select a neighboring agent at random;
- If the neighboring agent is of the opposite sex and if both agents are fertile and at least one of the agents has an empty neighboring site then a newborn is produced by crossing over the parents' genetic and cultural characteristics;
- Repeat for all neighbors.

We assume that new agents do not inherit diseases from either parent, that is, they start disease free.

The *Replacement* and *Mating* rules posit an initial endowment be given to newly created agents. This rule does not state where this endowment originates from. Either the endowment is somehow spontaneously produced "out of the ether" or it must come from its parent's resource stores.

Elsewhere in *Growing Artificial Societies* [Epstein and Axtell, 1996] it is stated that both parents donate to their offsprings initial endowment to the value of one half of their own initial endowment and that to have offspring an agent must have at least as much sugar as they were initially endowed with. This radically changes the dynamics of the Agent Mating rule as stated. We were left with a choice of using the rule as stated or including the extra information to our specification. Either choice seems defendable but we decided that including the extra information, although resulting in a more complex rule, is the most appropriate option.

In the case of *Replacement* and *Agent Mating* we have assumed that the constraints on newly created agents resources and attributes are the same as those for the original agents created during the simulation initialisation. The rule implies that those constraints may be different but we can see no useful purpose for this to be the case. Despite the implication that these starting constraints may differ we have specified that they are the same as those for the initial agents anyway as we believe this makes more sense.

4.5.8 Agent Inheritance

Agent Inheritance *I* When an agent dies its wealth is equally distributed among all its living children.

We have ignored rounding errors when dividing an agents resources amongst its offspring as resource levels are natural numbers. We have not specified any special way of handing this because do we think there is a need to and because it simplifies the specification.

We have also assumed that an agent who is due an inheritance but who is also marked for death (all deaths occurring simultaneously during a step) does not receive the inheritance, instead it is allocated only to children who are not marked for death. This seems the more correct interpretation.

Wealth is taken to mean all resources that an agent holds and also any outstanding loans that are due to them. Therefore these loans are also specified as being distributed amongst their children.

4.5.9 Agent Culture

Agent cultural transmission

- Select a neighboring agent at random;

- Select a tag randomly;
- If the neighbor agrees with the agent at that tag position, no change is made; if they disagree, the neighbor's tag is flipped to agree with the agent's tag;
- Repeat for all neighbors.

Group membership Agents are defined to be members of the Blue group when 0s outnumber 1s on their tag strings, and members of the Red group in the opposite case.

Agent Culture K Combination of the "agent cultural transmission" and "agent group membership" rules given immediately above.

Agent culture is a simple rule. It is stated in the book, but not in the rule definition, that agents always have an odd number of culture tags so avoiding any ties in the number of 1s and 0s.

Although the length of the culture string is undefined it is possible to deduce the most likely value used originally was 11 from examining a graph that appears within the text of *Growing Artificial Societies*.

4.5.10 Agent Combat

Agent Combat C_α

- Look out as far as vision permits in the four principal lattice directions;
- Throw out all sites occupied by members of the agent's own tribe;
- Throw out all sites occupied by members of different tribes who are wealthier then the agent;
- The reward of each remaining site is given by the resource level at the site plus, if it is occupied, the minimum of α and the occupant's wealth;
- Throw out all sites that are vulnerable to retaliation;
- Select the nearest position having maximum reward and go there;
- Gather the resources at the site plus the minimum of α and the occupants wealth if the site was occupied;
- If the site was occupied then the former occupant is considered "killed" - permanently removed from play.

The *Combat* rule states that we check each potential destination site to see if that site is subject to retaliation; that is if that location is within range of an agent belonging to a different tribe with greater wealth. It is not stated how we perform this test. An agent can only see other agents that are within its field of vision. If the search for agents capable of retaliation is based on our current position then the agent will only ever be able to see a small subset of the locations within striking distance of the intended destination. On the other hand, if we base the search on what is visible from the destination location, then we can see a more complete picture.

In either case, if we base what the agent can see on its own range of vision then we will not see agents, outside our range of vision, who may still be in striking range. Perhaps the rule allows us a "god's eye" view and we can detect all retaliation capable agents regardless of our vision. We have assumed that the attacking agent can see all agents around the destination location within the range of the attacking agents vision but other interpretations are possible and equally likely.

Retaliation is not the only ambiguity present in the $Combat$ rule. The rule is not clear about what action an agent takes when all available sites are subject to retaliation. There are two possible solutions: either we pick the least worst site and move there or we do not make any move. Arguments can be supplied to support either solution but we believe the intention is for the agent to remain *in situ* if all destinations are subject to retaliation. A more complex rule might entail moving to the site with the most wealth so as to make the agent less vulnerable to other agents.

Wealth is calculated using sugar levels but there is no guidance as to how it is calculated if the simulation has multiple resources. This same problem also occurs in the definition of reward. In both cases, we take the simplest solution defining reward and wealth using the simple sum of all available resources.

4.5.11 Credit

Credit $L_{d\,r}$

- An agent is a potential lender if it is too old to have children, in which case the maximum amount it may lend is one-half of its current wealth;
- An agent is a potential lender if it is of childbearing age and has wealth in excess of the amount necessary to have children, in which case the maximum amount it may lend is the excess wealth;

- An agent is a potential borrower if it is of childbearing age and has insufficient wealth to have a child and has income (resources gathered, minus metabolism, minus other loan obligations) in the present period making it credit-worthy for a loan written at terms specified by the lender;
- If a potential borrower and a potential lender are neighbors then a loan is originated with a duration of d years at the rate of r percent, and the face value of the loan is transferred from the lender to the borrower;
- At the time of the loan due date, if the borrower has sufficient wealth to repay the loan then a transfer from the borrower to the lender is made; else the borrower is required to pay back half of its wealth and a new loan is originated for the remaining sum;
- If the borrower on an active loan dies before the due date then the lender simply takes a loss;
- If the lender on an active loan dies before the due date then the borrower is not required to pay back the loan, unless inheritance rule I is active, in which case the lender's children now become the borrower's creditors.

The Credit rule states that after an agent's death all outstanding loans that this agent owes are nullified. This is reasonable as, on death, all the dying agents resources are destroyed. If, however, the inheritance rule is also in play then the children of the dying agent inherit their wealth. If an agent dies with outstanding loans due to them then these loans are passed on to their children and any money owed to them now become owed to their children.

It is not stated what happens to loans that the dying agent owes to other agents. While, logically, we might deduce that these debts are now debts of its children, in the absence of information to the contrary we have assumed that these debts are now nullified even though this arguably makes less sense in the light of the rest of the inheritance rule.

Similarly, it is unclear how loans belonging to agents killed in combat are handled. Some fraction of their resources are taken by the aggressor agent that killed them but it is not stated whether we redistribute any remaining resources (if any remain) or for that matter loans and debts. Does the combat rule override the inheritance rule? We have assumed it does.

In both these cases the ambiguities are caused by the interaction between different rules. In isolation each rule seems clear but when used together they are not. As ABM are used to study emergent properties it should be no surprise that the interactions between rules can cause ambiguities. This is especially true in Sugarscape as it contains a large number of rules that can be used in many different combinations. We also note that the process of formally specifying these rules brings these ambiguities to the fore where they can be properly addressed.

Creditworthiness depends on outstanding loans and ability to repay them. We have based this on all outstanding loans no matter when due. This is the simplest and was probably the intention as no information on how else it might be calculated is provided.

We also note that when we pay back loans before we run the death/replacement rule then there is the possibility of divide by zero. This would occur as the agent would have no resources. It could be argued that this problem is due to our having a separate *Death* rule. We have avoided this possible error by imposing restrictions on the order of the rule execution that precludes this possibility.

The *Credit* rule nowhere states how much an agent can or should loan to a potential borrower. Does an agent loan as much as it has available or only a fixed percentage of its resources? A borrower is defined as an agent that is fertile and that has insufficient wealth to have children. We have two issues here.

First, there is no mention of any minimum amount required by agents to have children in the *Mating* rule. This represents an important piece of missing information from the *Mating* rule. As a result, we do not know what this minimum amount should be.

Secondly, is this amount the minimum amount looked for by a borrower or a maximum amount a borrower is prepared to take? We have assumed that the difference between the resources required and held is the amount required by the borrowing agent. We also assume that an agent will accept a loan for a lesser amount if that is all that is offered and that the maximum amount offered by a lender is no more than that required by the borrower to have children.

As with other rules it is not stated how credit operates if there are multiple resources in play so we have assumed that separate loan agreements can be made for each resource.

4.5.12 Agent disease processes

Agent immune response

- If the disease is a substring of the immune system then end (the agent is immune), else (the agent is infected) go to the following step;
- The substring in the agent immune system having the smallest Hamming distance from the disease is selected and the first bit at which it is different from the disease string is changed to match the disease.

Disease transmission For each neighbor, a disease that currently afflicts the agent is selected at random and given to the neighbor.

Agent disease processes E Combination of "agent immune response" and "agent disease transmission" rules given immediately above

It is not clear when or where the effects of disease, such as increased metabolism rate, are calculated or even for that matter what exactly those effects would be. Without any guidance we assumed no effects.

4.5.13 Agent Trade

Agent Trade T

- Agent and neighbour compute their MRSs; if these are equal then end, else continue;
- The direction of exchange is as follows: spice flows from the agent with the higher MRS to the agent with the lower MRS while sugar goes in the opposite direction;
- The geometric mean of the two MRSs is calculated-this will serve as the bargaining price, p;
- The quantities to be exchanged are as follows: if p>1 the p units of spice for 1 unit of sugar; if p $<$ 1 the 1/p units of sugar for 1 unit of spice;
- If this trade will (a) make both agents better off (increases the welfare of both agents), and (b) not cause the agents' MRSs to cross over one another, then the trade is made and return to start, else end.

Trade is a complex rule to specify but it is stated sufficiently precisely so that we found no issues with its definition.

4.6 Conclusion

The rules of Sugarscape are defined in a simple manner, each in a paragraph of English text. However, these definitions really only serve as guidelines or summaries of the complete rule definition. As such they omit details that are either left ambiguous or referred to elsewhere in the text. A complete understanding of the rules requires a close reading of the entire book.

A good example of this is the *Agent Mating* rule. It omits important details that have a huge effect on how the rule plays out in a simulation. The rule as defined places few limits on how many offspring each agent can have during a step. Simulations employing this version of the rule rapidly fill the lattice to capacity with the explosive growth agents offspring. If, on the other hand, the omitted information is included in the rule a completely different population dynamic occurs. Agents will seldom have enough sugar to have more than one offspring during a step and only the fittest agents will have offspring.

By pulling all of the information together we have provided a a starting point for a standard definition of the rules that is consistent with the authors original intentions. The next step is to remove all remaining ambiguity by formally specifying these rules. This will allow us to tackle the *Replication Problem* in ABM. It is this that we turn to next.

Chapter 5

Formally Specifying Agent Based Models

5.1 Introduction

We start with a reminder of what the *Replication Problem* in ABM is. Following on from this we give a brief overview of formal specification in general and Z in particular.

We show our general approach of using formally methods for specifying an ABM by examining how we formally specified Sugarscape using Z. We then demonstrate the efficacy of using this approach by listing the underlying issues with the original definition of Sugarscape that the formal specification uncovered. We follow this section with a discussion of the merits of formally specifying ABMs and the possible weaknesses with this approach.

5.2 The Replication Problem

As we have previously stated, the Replication Problem in ABM is caused by the difficulty in reproducing results obtained by other researchers. Given the nature of ABM, where behaviour is defined in terms of simple agent interactions it might appear, at first glance, that reproducing a ABM defined by another researcher would be easy. However, in reality it has proven very difficult due to a number of factors as discussed in [Hinsen, 2011, Sansores and Pavn, 2005, Edmonds and Hales, 2003] and summarised here:

1. The simulation source code is not publicly available. Thus the simulation

cannot be rerun to confirm the results or the code checked for errors;

2. The rules of the model are not defined sufficiently precisely. Even a simple rule defined in a natural language (such as English) can have more than one valid (and possibly unseen) interpretation;

3. The updating method (AU or SU) is not declared or clearly explained. It is sometimes the case that some of the results of the simulation are caused by the updating strategy used and not due to any inherent property of the simulation.

There are a number of attempts to overcome these problems (as previously explained in section 2.5) but none have been completely successful.

We posit a new approach, that of using formal specification languages as used in computer science. Although developed for proving correctness in safety critical systems we will demonstrate that they can tackle the Replication Problem in ABM. In fact the use of Formal specification alongside multiple updating strategies and full access to code allows for reproduction of results not just replication.

5.3 Formal Specification

We chose the Z notation [Spivey, 1989] as our specification language. The Z notation is a formal specification language for describing and modelling computer systems. It was first proposed in 1977 by Jean-Raymond Abrial and gained ISO standard accreditation in 2002. It is based on Axiomatic Set Theory, Lambda Calculus and First Order Predicate Logic. All expressions in Z are typed.

Specification in Z is state-based, it is based on a system of states with operations defined in terms of before and after states and the relationship between these states. A specification is defined by first specifying the underlying state of the system (that is the totality of information contained within the system) and then specifying the complete set of allowable operations that are to be applied during each step. Each step is considered to be discrete in nature. In the case of ABM the state of the system is generally defined in terms of the sum of each individual agent's state. Each operation then occurs over the set of agents. This makes Z a good fit for ABM where simulations are defined in terms of atomic steps.

A high level Z specification will leave open (or undefined) implementation issues such as the precise algorithms employed to effect the changes defined in the specification. It is possible to formally refine a Z specification down into computer code but this is not necessary or even common. The high level specification only states the before and after states of each operation and any implementation that satisfies these constraints is allowable.

Because Z has been around for so long it has evolved into a mature and standardised system that is widely understood within the Formal Methods community. It has a range of software based tools [1] readily available for specifiers that aid with the production of specifications and proving properties of those specifications.

This availability of tools combined with the widespread recognition and maturity of Z alongside its state-based approach makes it a good choice for specifying ABMs. In particular, we have found that the specification of rules in a manner that is completely agnostic as to the implementation approach is a boon. Specifically it allowed the specifier to leave open what forms of conflict resolution rules are used. It only states the *before* and *after* state of the simulation on application of each rule.

5.4 Specifying Sugarscape

5.4.1 Overview

A Z specification will generally be broken down into the following steps. First we define any necessary *constants*. Alongside these constants we define new *types*, outside of the built-in types[2] provided by Z, that are required for the specification. Following this we specify the state of the system being specified. This state definition will include all the information held by the system alongside any *invariants*. Invariants are properties that remain true for the lifetime of the system. For example in Sugarscape each location cannot contain more sugar then its predefined maximum load carrying capacity.

Once the state has been defined we give the initial state of the system which assigns the appropriate initial values that the system will have on startup. With this in place all of the allowable operations are individually specified. Each operation is defined in terms of its effect on the state of the system. These effects are expressed by showing the relationship between the state as it was before the operation begins and the state as it is after the operation has completed. A good high level specification does not say how this transformation occurs, only what the end result will be. What algorithm we use to convert the before state to the after state is an implementation issue and there may be many different ways of effecting this transformation.

[1] `http://czt.sourceforge.net/`

[2] Such as Integers, Rationals, Reals, etc.

It is considered bad practice to enforce one particular implementation approach. For example, a specification of matrix multiplication will state what the resulting matrix looks like in relation to the two matrices being multiplied but will not enforce any one particular algorithm for achieving this result. Thus the specification would be open to being implemented using the standard implementation, a tiled implementation or even Strassen's algorithm [Huss-Lederman et al., 1996].

We made the decision to restrict the initial specification of Sugarscape to one pollution type and one resource type (known as *Sugar*) in an effort to guarantee clarity. While the original rules were designed so that they could be extended to arbitrary numbers of resources and pollutants, explicitly specifying an arbitrary number of resources and pollutants would make the specification more difficult to understand and thus more likely to either contain or cause mistakes.

Once we produced a specification for the single resource scenario we extended the specification to a two resource situation (these resources are known respectively as *Sugar* and *Spice*). This is necessary for specifying the final rule, $Trade$, as that rule requires two resources to function.

This two part approach allowed for:

1. A simpler and easier to understand specification of the rules that use only one resource (trading clarity against completeness);

2. A complete but separate specification for simulations that use two resources.

We do not provide specifications for multiple pollutants as multiple pollutants have never been used in Sugarscape. While specifying multiple pollutants makes for a more general specification there is a trade off in terms of the complexity this adds. If the specification becomes too complex it can make it more difficult to reason about. As multiple pollutants are never used we do not feel it is necessary to specify multiple pollutants.

Similarly we do not provide a specification for more than two resources as the benefits of doing so are counterbalanced by both the complexity of the resulting specification and the lack of any requirement to use such a complex simulation. Sugarscape has only ever been implemented with two standard resource types - *sugar* and *spice*. Anyone wishing to extend Sugarscape further can use the two resource specification for guidance.

We show how the specification of Sugarscape proceeded before discussing how effective this process was. The complete specification is not shown here due to its length but is presented in full in appendix D.

5.4.2 Types and Constants

Constants are defined by first naming the constant and then stating its type. After this we say what its specific value is and any invariant properties that it must satisfy. These invariants may relate its value to some other known constants. For example we could say that every agents vision is between the values of 1 and the maximum allowable agent vision. In some cases we may not wish to state what its value is and only state that it is some constant value. For example, the size of the lattice in Sugarscape is a constant size throughout any given simulation run but different values maybe used for any one particular run. The simulation size can be chosen based on the amount of available processing power we have available. In this case we state that it is a constant and leave its actual value unstated in the specification. Anyone using the specification must then decide themselves what size grid they are using and must also state what its value is. In this case the grid size is passed in as a parameter by the implementer of the simulation. These constants act as placeholders that must be given values before the simulation can proceed.

By identifying these constants that have not been assigned specific values we identify ambiguities in the definition of Sugarscape. These ambiguities may be deliberate but the fact that the specification pinpoints and identifies them means that anyone using the specification must explicitly state what values they must have thus enabling replication of any results obtained by other researchers.

The following extract of the specification shows how the important constants are specified. The specification is divided into two parts separated by a horizontal line, a structure followed by all Z specifications. Constants are declared above the line and then additional invariants attached to those constants are defined below the line.

$$
\begin{array}{|lr}
M : \mathbb{N}_1 & (1) \\
CULTURECOUNT : \mathbb{N}_1 & (2a) \\
MAXVISION : \mathbb{N}_1 & \\
MINMETABOLISM, MAXMETABOLISM : \mathbb{N} & (3a) \\
SUGARGROWTH : \mathbb{N}_1 & \\
MAXAGE, MINAGE : \mathbb{N}_1 & \\
MAXSUGAR : \mathbb{N}_1 & \\
DURATION : \mathbb{N}_1 & \\
RATE : \mathbb{A} & \\
INITIALSUGARMIN, INITIALSUGARMAX : \mathbb{N} & \\
WINTERRATE, SEASONLENGTH : \mathbb{N}_1 & \\
PRODUCTION, CONSUMPTION : \mathbb{N} & \\
COMBATLIMIT : \mathbb{N} & \\
IMMUNITYLENGTH : \mathbb{N} & \\
INITIALPOPULATIONSIZE : \mathbb{N} & \\
POLLUTIONRATE : \mathbb{N} & \\
CHILDAMT : \mathbb{N} & \\
\hline
CULTURECOUNT \bmod 2 = 1 & (2b) \\
MINMETABOLISM < MAXMETABOLISM & (3b) \\
MAXAGE < MINAGE & \\
MAXVISION < M & \\
INITIALSUGARMIN < INITIALSUGARMAX & \\
INITIALPOPULATIONSIZE \leq M * M &
\end{array}
$$

1. M is the dimension of the lattice used in the simulation ($M \times M$). 40 is the value most often alluded to in the original definition of Sugarscape;

2. $CULTURECOUNT$ defines the size of the culture string. Although we do not have a specific size for $CULTURECOUNT$ we do know that it must be an odd number as the culture string is defined to contain an odd number of bits and this is encoded in the invariant marked (2b);

3. Similarly although we know that $MINMETABOLISM$ and $MAXMETABOLISM$ (3a) values are not explicitly stated we are told that $MINMETABOLISM$ must be less than $MAXMETABOLISM$ as stated in the invariant marked (3b). Specific values are used for in different simulation runs depending on what simulation properties are being focused on.

When the simulation is run specific values *must* be assigned to all these constants and these values must satisfy all of the invariants. We leave these specific values to the implementer to assign but we have flagged what the constants that define any particular instance of Sugarscape are and have also forced the implementer to give them values within the given constraints. Values are chosen on a case by case basis depending on what the simulation is trying to demonstrate.

We often need to define new types in a specification. These new types are generally used to make the specification easier to read and understand as the examples below clearly demonstrate.

$$[AGENT] \quad (1)$$
$$POSITION == 0 \mathrel{..} M-1 \times 0 \mathrel{..} M-1 \quad (2)$$
$$SEX ::= male \mid female \quad (3)$$
$$BIT ::= 0 \mid 1 \quad (4)$$
$$affiliation ::= red \mid blue \quad (5)$$

1. $AGENT$ is used as a unique identifier for agents. We could just assign each agent a unique natural number but our approach makes our intentions easier to understand, the specification easier to read and is standard practice in Z;

2. $POSITION$ is also used to make specifying the 2D indices within the grid easier to read and more compact;

3. All agents have a sex attribute that can only take on of two possible values;

4. BITs are used to help encode both culture preferences and diseases of agents. Diseases are defined as specific strings of BITs, as is Culture;

5. Every agent has a cultural affiliation defined as either belonging to the blue tribe or red tribe. If an agent has more '1' BITs than 0 BITs in the culture bit string then it is said to be in the blue tribe, otherwise it is in the red tribe.

5.4.3 State

In Z, modularisation is achieved by dividing the specification into schemas. In this case we divided the state into two main separate parts or schemas. The first schema defines the information held by the lattice component of the simulation and the second schema defines the information held by the agents in the simulation. Although Lattice locations could also be viewed as a type of agent this division makes the state easier to comprehend and serves a useful purpose as some operations (known as *rules* or *behaviours* in Sugarscape terminology) act only on one of these two schemas (for example, the *Growback* rule/operation only affects the lattice and not the agents on the lattice).

The Lattice Schema

The Lattice is an $M \times M$ grid or matrix of locations where each location contains an amount of sugar[3] and pollution. Each location can hold up to a maximum amount of sugar where this maximum amount can vary from location to location.

We can see that a schema has two parts. The top part is where the variables defining the state are declared. The bottom part lists all the invariant properties that the schema enforces.

$$\begin{array}{|ll}
\multicolumn{2}{l}{\textit{Lattice}} \\
sugar : POSITION \rightarrow \mathbb{N} & (1) \\
maxSugar : POSITION \rightarrow \mathbb{N} & (2) \\
pollution : POSITION \rightarrow \mathbb{N} & (3) \\
\hline
\forall x : POSITION \bullet sugar(x) \leq maxSugar(x) \leq MAXSUGAR & (4) \\
\hline
\end{array}$$

Taking each part of the schema in turn:

1. $sugar$ is a mapping that stores the amount of sugar stored at each position in the lattice;
2. $maxSugar$ is a mapping that records the maximum amount of sugar that can be stored in (carried by) each position;
3. $pollution$ records the amount of pollution at each location;
4. This is the only invariant. It states that every position's sugar level is less than or equal to the maximum allowed amount for that position which is in turn less than or equal to the $MAXSUGAR$ constant;

[3]We ignore spice here for simplicity. The complete full specification also includes spice.

Each function in the *Lattice* schema is a total function. Different types of function such as Total, Partial, Injective, etc. are defined within Z and each has its own symbol. Because they are total functions it is implictly implied that every position has a sugar level, a maximum level and a pollution amount.

The different function types are defined as follows:

$X \pfun Y$	-	Partial function: some members of X are paired with a member of Y
$X \rightarrow Y$	-	Total function: every member of X is paired with a member of Y
$X \pinj Y$	-	Partial injection: some members of X are paired with different members of Y
$X \rightarrowtail Y$	-	Total injection: every member of X is paired with a different member of Y
$X \bij Y$	-	Bijection: every member of X is paired with a different member of Y, covering all Y's

The Agent Schema

The agent schema is more complex due to the amount of information held by each agent. The attributes that every agent has are:

Position Where the agent currently resides on the Lattice;

Vision How far in the four cardinal directions that an agent can see;

Age Number of turns of the simulation that an agent has been alive;

Maximum Age Age at which an agent dies (assuming it has not being killed previously by combat or starvation);

Sex Agents are either male or female;

Sugar Level The amount of sugar that an agent currently holds. There is no limit to how much sugar an agent can hold;

Metabolism The amount of energy, defined by sugar (or resource) consumption, used during every turn of the simulation;

Culture Tags A sequence of bits that represents the culture of an agent;

Children For each agent we track its children (if any). This is necessary as the *Inheritance* rule requires that we track each agent's children;

Loans Under the credit rule agents are allowed lend and/or borrow sugar for set durations and interest rates so we need to track these loans. For each loan we need to know the lender, the borrower, the loan principal and the due date (represented as the step number);

Diseases Diseases are sequences of bits that can be passed between agents. An agent may carry more than one disease;

Immunity Each agent has an associated bit sequence that confers immunity against certain diseases. If the bit sequence representing a disease is a subsequence of an agents immunity bit sequence then that agent is considered immune to that disease.

We have listed here the full range of agent attributes. In practice we may only need to use a subset of these depending on which subset of rules we are implementing in our simulation. For example, *Disease* and *Immunity* attributes are required only if we are implementing the *Disease* rule so any simulation not using this rule will be simplified by not modelling these attributes.

Agents

$$
\begin{array}{l}
population : \mathbb{P}\, AGENT \\
position : AGENT \rightarrowtail POSITION \\
sex : AGENT \nrightarrow SEX \\
vision : AGENT \nrightarrow \mathbb{N}_1 \\
age : AGENT \nrightarrow \mathbb{N} \\
maxAge : AGENT \nrightarrow \mathbb{N}_1 \\
metabolism : AGENT \nrightarrow \mathbb{N} \\
agentSugar : AGENT \nrightarrow \mathbb{N} \\
agentCulture : AGENT \nrightarrow \mathrm{seq}\, BIT \\
children : AGENT \nrightarrow \mathbb{P}\, AGENT \\
loanBook : AGENT \leftrightarrow (AGENT \times (\mathbb{N}, \mathbb{N})) \\
agentImmunity : AGENT \nrightarrow \mathrm{seq}\, BIT \\
diseases : AGENT \nrightarrow \mathbb{P}\, \mathrm{seq}\, BIT
\end{array}
$$

$$
\begin{array}{ll}
population = & \\
\quad \mathrm{dom}\, position = \mathrm{dom}\, sex = \mathrm{dom}\, vision & \\
\quad = \mathrm{dom}\, maxAge = \mathrm{dom}\, agentSugar = \mathrm{dom}\, children & \\
\quad = \mathrm{dom}\, agentCulture = \mathrm{dom}\, metabolism = \mathrm{dom}\, age & \\
\quad = \mathrm{dom}\, agentImmunity = \mathrm{dom}\, diseases & (1) \\
\mathrm{dom}\, loanBook \subseteq population & (2) \\
\mathrm{dom}(\mathrm{ran}\, loanBook) \subseteq population & (3) \\
\forall\, x : AGENT;\ d : \mathrm{seq}\, BIT \bullet & \\
x \in population \Rightarrow & (4) \\
\quad ((age(x) \leq maxAge(x) \wedge MINAGE \leq maxAge(x) \leq MAXAGE & \\
\quad \wedge\ \#\, agentCulture(x) = CULTURECOUNT & \\
\quad \wedge\ \#\, agentImmunity(x) = IMMUNITYLENGTH & \\
\quad \wedge\ vision(x) \leq MAXVISION & \\
\quad \wedge\ MINMETABOLISM \leq metabolism(x) \leq MAXMETABOLISM) & \\
d \in \mathrm{ran}\, diseases(x) \Rightarrow \#\, d < IMUNITYLENGTH & (5)
\end{array}
$$

1. Every existing agent has an associated age, sex, vision, etc. Note that the population holds only the currently existing agent IDs. Once an agent dies it is no longer modelled and no longer part of the population;

2. Only agents that are alive (current members of the population) can be lenders;

3. Only agents that are alive (current members of the population) can be borrowers
4. For every agent in the population:
   (a) It has a current age less than the maximum allowed age for that agent and this maximum age is less than or equal to the globally defined $MAXAGE$ constant;
   (b) Metabolism is always between the allowed limits and vision less than or equal to the maximum vision;
   (c) The sequence of bits representing its culture tags is $CULTURECOUNT$ in size while the string representing immunity is $IMMUNITYLENGTH$ in size.
5. All diseases are represented by sequences of bits that are shorter than the length of the immunity sequence.

We need to track the number of turns that have occurred in the simulation. Each turn consists of the application of all rules that form part of the simulation. This is specified in the simple $Step$ schema. In this schema the variable $step$ contains the current simulation step count.

$Step$

$step : \mathbb{N}$

The entire simulation consists of locations, agents and the counter holding the tick count. We combine them all in the schema $SugarScape$. This defines the entire state as consisting of both the agents state, the lattice state and a count indicating how many steps have been taken so far in the simulation. We note here that the inclusion of a schema name is taken to be shorthand for including all of that schema (both definitions and invariants). This schema contains every part of the three achemas: $Step$, $Lattice$ and $Agents$.

$SugarScape$

$Agents$
$Lattice$
$Step$

5.4.4 Initial State

At the start of the simulation certain variables must be initialised with values. The initial state schema states what these values are.

We note here the first appearance of the notion of *after-state*. As we already noted Z specifies operations in terms of how the state after the operation relates to the state as it was before the operation. For any particular variable X in a schema, X refers to the value held by that variable before the operation starts while X' (X *prime*) refers to the value it holds after the operation has finished. Thus to state that an operation increments X by one we would write $X' = X + 1$. We note that "$=$" is the equality predicate and not an assignment operator[4].

For the initial state there is no before state as the simulation does not exist before the initial state. Therefore only the after values of variables are referenced. We can see this by the inclusion of only $Sugarscape'$ (primed) in this schema. Sugarscape' is the Sugarscape schema with every variable *primed* (i.e. only the after states of variables are included).

$InitialSugarScape$

$Sugarscape'$

$$\begin{array}{ll}
step' = 0 & (1) \\
\#population' = INITIALPOPULATIONSIZE & (2) \\
loanBook' = \varnothing & (3) \\
\forall\, a : AGENT \bullet & (4) \\
a \in population' \Rightarrow & \\
\quad (age(a) = 0 \land diseases'(a) = \varnothing \land children'(a) = \varnothing & \\
\quad \land\ INITIALSUGARMIN \leq agentSugar'(a) & \\
\qquad \leq INITIALSUGARMAX) &
\end{array}$$

[4] So $X + 1 = X'$ or $X = X' - 1$ are also a valid way of stating that X in incremented by one.

1. $step$ is set to zero;
2. The population is set to some initial size (a constant that must have been explicitly given a value before the simulation can proceed);
3. There are no existing loans as yet (the $loanBook$ is an empty set);
4. Every agent in the starting population has an age of zero, has no diseases or children and has some initial sugar level within the agreed limits. In this case any random value is allowed for the sugar level as long as it satisfies the scheme invariants

The other attributes remain undefined. This means they can start with any value restricted only by the invariants of the $Agents$ and $Lattice$ schemas. Similarly the initial values assigned to the lattice can be any random values that satisfy the invariants stated in the Lattice schema. In consequence agents can be placed anywhere on the lattice as long as the schema invariants are satisfied. These constraints enforce conditions such as ensuring a maximum of one agent can be at any location at a time.

5.4.5 Specifying Behaviours

Each rule in Sugarscape is defined as a separate operation in Z. To demonstrate how this proceeds we will first show the specification for the *PollutionDiffusion* rule as it is one of the simpler rules. First we show the original rule definition as stated in [Epstein and Axtell, 1996]:

Pollution Diffusion D_α

- Each α time periods and at each site, compute the pollution flux the average pollution level over all its von Neumann neighbouring sites;
- Each site's flux becomes its new pollution level.

This rule determines how pollution levels diffuse over the locations in the lattice. Pollution diffusion is calculated every α turns and is computed as the average pollution level of all that location's von Neumann neighbours. We rename the constant α as $POLLUTIONRATE$ for clarity as the greek symbol α is used in many Sugarscape rule definitions with different meanings in each case.

The von Neumann neighbours of a location are those immediately above, below, left and right of the current locations (aka North, South, East and West). We define the four cardinal directions taking into account the fact that the grid wraps around at its edges (i.e. it is a torus). Once defined we can use these functions in our schemas.

$$
\begin{array}{|l}
north : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
south : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
east : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
west : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
\hline
\forall x, y : \mathbb{N} \bullet \\
\quad west((x, y)) = ((x - 1) \bmod M, y) \\
\quad east((x, y)) = ((x + 1) \bmod M, y) \\
\quad south((x, y)) = (x, (y - 1) \bmod M) \\
\quad north((x, y)) = (x, (y + 1) \bmod M)
\end{array}
$$

Each function is defined as a *bijection* ensuring that (taking north as an example):

1. Every location has one and only one location to its north;
2. Every location is north of one and only one other location.

$$
\begin{array}{|l}
\underline{\quad PollutionDiffusion \qquad\qquad\qquad\qquad\qquad\qquad} \\
\Delta Lattice \\
\Xi Step \\
\hline
maxSugar' = maxSugar \\
sugar' = sugar \\
(step \bmod POLLUTIONRATE \neq 0) \Rightarrow pollution' = pollution \\
(step \bmod POLLUTIONRATE = 0) \Rightarrow pollution' = \\
\quad \{\forall\, l : POSITION \bullet l \mapsto (pollution(north(l)) + pollution(south(l)) \\
\qquad + pollution(east(l)) + pollution(west(l))) \operatorname{div} 4\} \\
\hline
\end{array}
$$

Placing the symbol Δ before a schema name indicates the inclusion of the before and the after states but does not state the relationship between them. The following two schemas are equivalent:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PollutionDiffusion \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta Lattice \\
\hline
\\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad PollutionDiffusion \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
Lattice \\
Lattice' \\
\hline
\\
\hline
\end{array}$$

If we use Δ then we are required to state in the schema how the after state values relates to the before state values.

Placing the symbol Ξ before a schema name indicates the inclusion of the before and the after states with the added invariant that all the after states are equal to their before states, that is the schema does not (and cannot) change the state values. Thus in this schema we do not need to state the after state of the $step$ variable - it must be equal to the before state. In other words, $\Xi Step$ is shorthand for including $step$ and $step'$ alongside the invariant $step' = step$.

The $PollutionDiffusion$ schema tells us that both $maxSugar$ and $Sugar$ values remain unchanged. This is stated explicitly by saying that the before and after values are equal. Pollution diffusion occurs only once every $POLLUTIONRATE$ steps. Our schema enforces this by only calculating new pollution levels when $step$ (the variable telling us the number of this tick) is evenly divided by $POLLUTIONRATE$. When it is updated the new pollution level at a location is the average of its neighbour's pollution, specifically their pollution level before the operation occurs. We note in passing that this rule is defined to be explicitly synchronous (or concurrent). We can only infer from this that the authors of Sugarscape do not hold that asynchronous updating is, as some claim [Huberman and Glance, 1993], more reflective of reality but instead is used only as a convenience. Convenient because some operations are easier to implement sequentially than concurrently.

5.4.6 Synchronous Rule Specification

It is instructive, at this point, to show how rules or behaviours can be specified both synchronously and asynchronously. To do this we will specify the *Movement* rule as an example, first synchronously and then asynchronously.

Movement - M

- Look out as far as vision permits in each of the four lattice directions, north, south, east and west;
- Considering only unoccupied lattice positions, find the nearest position producing maximum welfare;
- Move to the new position
- Collect all resources at that location

The previous rules affected only the locations but the remaining rules affect agents as well as locations. The Movement rule determines how agents select their next location. There are a number of different versions of this rule defined in *Growing Artificial Societies*.

Not explicitly stated within the rule but added as a footnote to the original rule definition is the restriction that the order in which the lattice directions are searched should be random. This comes into play when two or more available sites exist with the same welfare score.

This rule does **not** guarantee that an agent will move to the best location. To see why this is the case consider what happens if two agents both try to move to the same location. Only one can succeed and the other will have to move to a less advantageous location. How we decide which agent succeeds is not defined. We assume that either a conflict resolution or conflict avoidance rule is available to make this decision but it is not stated what this rule should be. The original implementation assumes sequential updating is applied thus enforcing collision avoidance. However a SU approach does have to handle collision detection in some way as it assumes all agents move concurrently.

To help make the specification clear we define some simple helper functions. The distance between two positions is only defined for positions that are directly horizontal or vertical. This function takes into account the torus-like (wrap-around) structure of the simulation. This function takes as input two positions and returns the distance between them as a natural number.

$$distance : POSITION \times POSITION \\ \rightarrow \mathbb{N}$$

$$\forall x1, x2, y1, y2 : \mathbb{N} \bullet$$

$$distance((x1, y1), (x1, y2)) = \qquad (1)$$

$$\quad min(\{\mid y2 - y1 \mid, M - \mid y2 - y1 \mid\})$$

$$distance((x1, y1), (x2, y2)) = \qquad (2)$$

$$\quad min(\{\mid x1 - x2 \mid, M - \mid x1 - x2 \mid\})$$

$$distance((x1, y1), (x2, y2)) = \infty \Leftrightarrow$$

$$\quad x1 \neq x2 \wedge y1 \neq y2 \qquad (3)$$

1. If two agents are vertically aligned (they share the same x coordinate) we calculate distance based on the horizontal distance (the y coordinate);

2. If two agents are horizontally aligned (they share the same y coordinate) we calculate distance based on the vertical distance (the x coordinate);

3. Otherwise the distance is defined as infinity.

Given this distance function we can now define the $Movement$ schema.

$$
\begin{array}{l}
\textit{Movement} \\
\hline
\Delta \textit{SugarScape} \\
\hline
step' = step \\
population' = population \\
maxSugar' = maxSugar \\
pollution' = pollution \\
sex' = sex \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
loanBook' = loanBook \\
diseases' = diseases \\
agentImmunity' = agentImmunity \\
children' = children \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
\forall a : AGENT;\ l : POSITION \bullet \\
a \in population' \Rightarrow \hfill (1) \\
\quad distance(position'(a), position(a)) \leq vision(a) \\
\\
(distance(position(a), l) \leq vision(a) \land (l \notin \operatorname{ran} position')) \Rightarrow \hfill (2) \\
\quad sugar(l) \leq sugar(position'(a)) \hfill (2a) \\
\quad \land\ (distance(l, position(a)) < distance(position'(a), position(a))) \hfill (2b) \\
\qquad \Rightarrow sugar(l) < sugar(position'(a)) \\
\\
agentSugar' = \{\forall a : AGENT \mid a \in population' \bullet \\
\quad a \mapsto agentSugar(a) + sugar(position'(a))\} \hfill (3) \\
sugar' = sugar \oplus \{\forall l : POSITION \mid l \in \operatorname{ran} position' \bullet l \mapsto 0\} \hfill (4) \\
\hline
\end{array}
$$

After the rule is applied the following most of the state variables remain unchanged. If a variable retains its value then this must be explicitly stated in the schema (e.g $step' = step$). The invariants from the included schemas are also implicitly carried forward into the new schema and so also remain true. It is the case for every agent:

1. They will be located within one of the locations in their original neighbourhood (possibly the same position as before). That is they cannot move further than their neighbourhood in a single step;

2. After every agent has moved:

   a) There will exist no remaining unoccupied locations from the original neighbourhood of any agent that would give a better welfare score than the location that agent now inhabits (Each agent has picked the maximum rewarding destination that no other agent has moved to);

   b) If there was more than one location with maximum reward then the agent moved to the closest location.

3. Agent sugar levels increase because they consume all the sugar at their new location. This is true even if the new location is the same as their old location;

4. All Location sugar levels are set to zero everywhere there is an agent present (as agents consume all sugar on arrival at a location) and remain unchanged if no agent is at that location.

5.4.7 Specifying Conflict Resolution

The specification states what is true after the application of the rule but not how we achieve that state. In any implementation some conflict resolution strategy will be needed but in the $Movement$ schema above we remain agnostic as to what it should be. In effect this is a random choice conflict resolution strategy: if two or more agents want to move to the same location the specification gives no indication as to which agent succeeds. A possible conflict resolution rule could be that when two agents try to move to the same destination then the closest agent to the destination wins by virtue of arriving there first. We can call this *Closest Wins*.

We can incorporate this into our Move rule easily by inserting the following constraint into the body of the *Movement* specification:

$$
\begin{aligned}
&(distance(position(a), l) \leq vision(a) && (1)\\
&\wedge\ sugar(l) \geq sugar(position'(a)) && (2)\\
&\wedge\ (distance(l, position(a)) < distance(position'(a), position(a)))) \Rightarrow && (3)\\
&\quad l \in \operatorname{ran} position' && (4)\\
&\quad \wedge\ (distance(l, position(a)) \geq distance(l, position(position'^{\sim}(l))) && (5)
\end{aligned}
$$

This states in logic the only circumstances in which we do not move to the best destination is when another agent closer to that destination has moved there.

1. *If* there was a location[5] within range
2. *and* this location had at least as much sugar as our final destination
3. *and* this location was closer to us than our chosen destination
4. *then* it must be the case that another agent moved to this location
5. *and* that other agent was closer to it than we were.

Any collision resolution rule that can be defined can be added to the specification in the same manner. As another example of a conflict resolution rule would be *Strongest Agent Wins* where we define strongest as the agent with the most sugar reserves or wealth.

$$
\begin{aligned}
&(distance(position(a), l) \leq vision(a) && (1)\\
&\wedge\ sugar(l) \geq sugar(position'(a)) && (2)\\
&\wedge\ (distance(l, position(a)) < distance(position'(a), position(a)))) \Rightarrow && (3)\\
&\quad l \in \operatorname{ran} position' && (4)\\
&\quad \wedge\ (sugar(position'^{\sim}(l)) > sugar(a)) && (5)
\end{aligned}
$$

Lines 1 to 4 are exactly the same as the previous collision resolution rule. Line 5 differs in that it states that the agent that moved to this destination instead of us must have had more sugar than us.

[5]Variables a and l (agent and location respectively) are both defined within the *Movement* specification body

In this way it is possible to incorporate many different collision resolution strategies into the behaviours. It is not possible to do the same with asynchronous rule definitions due to the sequential nature of this updating strategy. We will now look at how the *Movement* rule can be defined asynchronously.

5.4.8 Specifying Asynchronous Updating Strategies

AU is the sequential application of rules to agents during a simulation step. If, for example, all agents move during a single step then a sequential ordering is imposed on all of the agents and they will move one at a time (that is, sequentially) based on that ordering. This is in contrast to SU where all agents will attempt to move simultaneously (concurrently). AU is easier to implement that SU as it maps directly onto the current standard sequential programming practice. AU requires no collision detection and resolution (as for example when two agents try to simultaneously move to the same location) because concurrency is excluded - only one agent can move at any one time. It is well know that the AU and SU approaches can deliver different simulation results.

There are a number of varieties of AU [Schönfisch and de Roos, 1999]. These variations differ in how they sequentially order agents for updating. The four best known variations are :

Fixed Direction Line-By-Line The locations in the lattice representing the simulation space are updated in the order they appear in the lattice (usually left to right, top-down);

Fixed Random Sweep The order that is used is determined randomly at the start of the simulation and this order is used for every step in the simulation;

Random New Sweep The order that the agents are updated in is determined randomly at the start of each step (each step uses a different order);

Uniform Choice Each agent has an equal probability of being chosen. If there are n agents, then n agents are chosen randomly during a step. During any single step an agent may not be picked at all or may be picked more than once (in contrast Random New Sweep guarantees every agent is picked exactly once per step);

We will provide a specification for each variation in turn. Each specification will order the agents in a simulation according to the rules of each particular AU variation.

Fixed Direction Line-by-Line takes in a set of agents and their positions on the lattice. It produces a sequence of agents where every agent appears once and only once in the sequence and the order of the sequence is determined by the agents position on the lattice.

$$\begin{array}{|l}
lineByLine : AGENT \rightarrowtail POSITION \\
\rightarrow seqAGENT \\
\hline
\forall thePositions : AGENT \rightarrowtail POSITION; \\
theSequence : \mathrm{seq}\, AGENT \bullet \\
lineByLine(theSet) = theSequence \\
\quad \Leftrightarrow \mathrm{ran}\, theSequence = \mathrm{dom}\, thePositions \wedge \\
\quad\quad \# theSequence = \# thePositions \qquad (1) \\
(n, a) \in theSequence \Leftrightarrow \\
\quad n = first(thePositions(a)) * DIM + second(thePositions(a)) \qquad (2)
\end{array}$$

1. Each agent in the population appears in the sequence once and only once;

2. If one agent appears before another in the sequence then it must also appear before that agent on the lattice where order is defined in terms of row and column indices of position.

Fixed Random Sweep returns a sequence of the agents in some fixed random ordering. This random ordering is chosen once at the start of the simulation and is fixed for the entire simulation run. First we define what a random ordering of lattice locations looks like:

$$\begin{array}{|l}
RANDOMORDER : \mathrm{seq}\, POSITION \\
\hline
\# RANDOMORDER = \# POSITION \qquad (1) \\
\forall n, m : \mathbb{N} \bullet RANDOMORDER(n) = RANDOMORDER(m) \qquad (2) \\
\quad \Leftrightarrow n = m
\end{array}$$

1. $RANDOMORDER$ is a globally defined sequence containing an ordering of positions on the lattice;

2. Each position on the lattice appears once and only once in this sequence.

Any ordering that satisfies these constrains is allowable according to our specification. This introduces the randomness into the sequence. Note that $RANDOMORDER$ represents one particular random sequencing.

$$
\begin{array}{|l}
fixedRandom : AGENT \rightarrowtail POSITION \\
\leftrightarrow seq AGENT \\
\hline
\forall thePositions : AGENT \rightarrowtail POSITION; \\
theSequence : \mathrm{seq}\, AGENT \bullet \\
fixedRandom(thePositions) = theSequence \\
\quad \Leftrightarrow \mathrm{ran}\, theSequence = \mathrm{dom}\, thePositions \land \\
\qquad \#\, theSequence = \#\, thePositions \qquad (1) \\
\forall i : 0 \mathrel{..} \#\, theSequence - 2;\ a_1, a_2 : AGENT \bullet \\
\quad (i, a_1) \in theSequence \land (i+1, a_1) \in theSequence \Rightarrow \\
\quad (\exists x_1, x_2 : \mathbb{N} \mid (x_1, a_1), (x_2, a_2) \in RANDOMORDER \\
\qquad \land x_1 < x_2 \qquad (2)
\end{array}
$$

1. Every agent in the population appears once and only once in the resulting sequence;

2. The ordering of agents in the sequence is based on the ordering defined in $RANDOMORDERING$.

Random New sweep is simpler to specify. We return a random ordering of agents after each call. We only need to ensure that every agent appears in this sequence exactly once.

$$
\begin{array}{|l}
rndNewSweep : AGENT \rightarrowtail POSITION \\
\leftrightarrow seq AGENT \\
\hline
\forall thePositions : AGENT \rightarrowtail POSITION; \\
theSequence : \mathrm{seq}\, AGENT \bullet \\
rndNewSweep(thePositions) = theSequence \\
\quad \Leftrightarrow \mathrm{ran}\, theSequence = \mathrm{dom}\, thePositions \land \\
\qquad \#\, theSequence = \#\, thePositions \ (1)
\end{array}
$$

1. Every agent in the population appears once and only once in the resulting

sequence;

Uniform Choice allows for an agent to be picked multiple times. The only constraints are that the sequence returned contains only agents in the population and that the size of the sequence equals the number of agents.

$$uniformChoice : AGENT \rightarrowtail POSITION \leftrightarrow \operatorname{seq} AGENT$$

$$\begin{aligned}
&\forall thePositions : AGENT \rightarrowtail POSITION;\ n : \mathbb{N};\\
&theSequence : \operatorname{seq} AGENT \mid 0 \leq n < \# theSequence \bullet\\
&uniformChoice(thePositions) = theSequence \Leftrightarrow\\
&\quad theSequence(n) \in \operatorname{dom} thePositions && (1)\\
&\quad \land \# theSequence = \# thePositions && (2)
\end{aligned}$$

1. Every agent in the sequence is an agent from the simulation population;

2. The size of the sequence equals the total number of individual agents in the population.

Each variation of AU can now be covered by the simple matter of swapping one of the above ordering functions within the specification.

The specification of rules under an AU regime follows a standard pattern. First we impose an ordering on all the agents subject to the rule and then we recursively apply the update to each agent in the defined order. Each individual agent update can affect the global state and these changes must be passed forward to the next sequence of agent updates. This is in contrast to SU where all updates occur simultaneously.

We always define the application of the rule to agents in a sequence recursively. While the rules themselves can be quite simple, the Z notation forces us to pass to each update all parts of the global state that can be changed. This can result, as can be seen, in large function signatures. These function signatures can be difficult to read but this is a flaw within the Z notation itself that needs to be addressed rather than AU.

$AsyncMovement$

$\Delta SugarScape$

$$
\begin{aligned}
&step' = step\\
&loc' = loc\\
&maxSugar' = maxSugar\\
&pollution' = pollution\\
&sex' = sex\\
&vision' = vision\\
&age' = age\\
&maxAge' = maxAge\\
&agentCulture' = agentCulture\\
&loanBook' = loanBook\\
&diseases' = diseases\\
&agentImmunity' = agentImmunity\\
&children' = children\\
&metabolism' = metabolism\\
&population' = population\\
&initialSugar' = initialSugar\\
&(sugar', agentSugar', position') =\\
&\qquad applyMove(rndNewSweep(position), vision,\\
&\qquad\qquad sugar, agentSugar, position)
\end{aligned}
$$

Movement is a typical example of this structure. The main specification $AsyncMovement_{basic}$ simply passes the relevant state information alongside the ordering of agents (according to whatever AU variant we are using - in this case it is $rndNewSweep$) to the recursive function $applyMove$. This recursive function applies the move rule to each agent in turn and returns the final updated agent position, agent sugar levels and lattice sugar levels. We will look at this function now.

$$
\begin{array}{l}
applyMove : \mathrm{seq}\, AGENT \\
\times AGENT \nrightarrow \mathbb{N} \\
\times (POSITION \rightarrowtail\!\!\!\!\nrightarrow \mathbb{N}) \\
\times AGENT \rightarrowtail\!\!\!\!\nrightarrow \mathbb{N} \\
\times AGENT \rightarrowtail\!\!\!\!\nrightarrow POSITION \\
\quad \leftrightarrow \\
((POSITION \rightarrowtail\!\!\!\!\nrightarrow \mathbb{N}) \\
\times AGENT \rightarrowtail\!\!\!\!\nrightarrow \mathbb{N} \\
\times AGENT \rightarrowtail\!\!\!\!\nrightarrow POSITION)
\end{array}
$$

$$
\begin{array}{ll}
\forall\, head : AGENT;\ tail : \mathrm{seq}\, AGENT;\ population : \mathbb{P}\, AGENT; & \\
positions : AGENT \rightarrowtail\!\!\!\!\nrightarrow POSITION;\ sugar : POSITION \rightarrowtail\!\!\!\!\nrightarrow \mathbb{N}; & \\
agentSugar : AGENT \rightarrowtail\!\!\!\!\nrightarrow \mathbb{N};\ vision : AGENT \nrightarrow \mathbb{N};\ \bullet & \\
applyMove(\langle\rangle, vision, sugar, agentSugar, positions) = & (1) \\
\quad (sugar, agentSugar, positions) & \\
 & \\
applyMove(\langle head \rangle \frown tail, vision, sugar, agentSugar, positions) = & (2) \\
\exists\, newLoc : POSITION \mid & \\
\quad newLoc \in neighbourhood(position(head), vision(head)) & (3) \\
\wedge\ \forall\, otherLoc : POSITION \mid & \\
\quad otherLoc \in neighbourhood(position(head), vision(head)) & \\
\quad \Rightarrow sugar(otherLoc) \leq sugar(newLoc)\ \bullet & \\
\quad applyMove(tail, vision, sugar \oplus \{newLoc \mapsto 0\}, & \\
\qquad agentSugar \oplus \{head \mapsto agentSugar(head) + sugar(newLoc)\}, & \\
\qquad positions \oplus \{head \mapsto newLoc\}) &
\end{array}
$$

1. **The base case:** If there are no agents left to update then we simply return the current state;

2. **The recursive case:** If we have agents left to process then we move the first agent in the list and apply the rule to the remaining agents;

   (a)

   (b) Find the best location for the agent to move to based on sugar levels at each location.

$AsyncMovement_{basic}$ uses Random New Sweep to determine the sequencing application of the move behaviour to individual agents. Any of the other updating strategies can be used by replacing the $rndNewSweep$ function call with the appropriate replacement.

5.4.9 Overview of Specification Process

The modularity within Z allowed us to specify each of the 13 rules independently. This is essential as Sugarscape is really a family of simulations where different rule combinations can be applied to demonstrate different aspects of the society that is simulated. For example if we are interested in how cultures spread across a society we would use only three of the rules: *Growback, Movement* and *Culture*.

We divided our specification into two parts. First we specified Sugarscape with only one defined resource (*Sugar*). This allowed us to specify 12 out of the 13 rules while simultaneously keeping the complexity of the specification as low as possible. The 13^{th} rule requires the simulation to have two resources (*Sugar* and *Spice*). We specified Sugarscape again, but this time included both resources. This allowed us to specify the final rule and demonstrate that Z can handle the most complex ABM rules.

The rules are specified with both AU and SU. The specification allows for multiple possible definitions of Collision Resolution under SU giving flexibility and generality to the specification. To be useful the specification must allow some degree of freedom for implementers of Sugarscape. Models are only really useful if they are robust to changes in the specification. Any results claimed for Sugarscape should also hold for a wide range of similar models, otherwise the results are not generalisable. The specification acts as a framework that modellers can work within to see how general the properties of the simulation are. It gives freedom to try well defined changes and see how or if they affect the simulation outcomes.

The specification can be used to compare:

1. Different implementations of Sugarscape to test for reproducibility of results;

2. Asynchronous and synchronous updating strategies to see if the properties of Sugarscape are robust under different updating strategies;

3. Different collision detection and resolution rules when synchronous updating is employed.

5.5 Results of the Specification

The issues with the original definition of Sugarscape as presented in *Growing Artificial Societies* that we have encountered, and dealt with, can broadly be grouped into three main types: *Lack of Clarity, Missing Information* and *Sequential biases*.

Lack of Clarity The rules, although simply stated, lack clarity in their definition. Only one version of each rule is presented even when many variations are referred to in the original definition. The variations presented cannot always be used together, for example the *Movement* rule defined in the appendix is not the variant required if the pollution rule is also used. This specification brings all the variants together in one place for ease of reference. Where there is more than one variation of a rule all the different variants were specified and the appropriate one can be chosen based on the context.

Missing Information Missing or incomplete information is the biggest cause for concern. In many cases we can work out the most likely answer based on context but in some cases there is not one definitive correct answer. If there was more than one arguably correct solution the simplest was chosen. All hidden assumptions that could serve to advantage one implementation over another are excised. How these blanks are filled in can have a big effect on how the simulation proceeds. These effects may be important when comparing different implementations of Sugarscape. By replacing each ambiguous interpretation with one simple and precise interpretation we allow different developers to replicate their results and benchmark them against each other.

Sequential Biases Sugarscape is based on the assumption that it will be implemented sequentially. While this may have been a good assumption at the time it was written it is not now necessarily the case. Improvements in processing speed have recently been attained mainly through the introduction of concurrency. Simulations are now almost always run on multicore or even multiprocessor machines. The Z specification is free from all sequential assumptions. This leaves developers the freedom to try out different approaches as suits their implementation platform.

Further work remains to be done in getting agreement from the ABM community on the decisions made in producing this interpretation of Sugarscape. Any incorrect assumptions made in the course of producing this specification need to be identified, agreed upon and corrected. These issues can now be teased out by the

ABM community. A more complete list of the issues identified during the specification process can be found in appendix C. This specification is also available online [Kehoe, 2015a].

Sugarscape can now be used as a benchmark (or rather set of benchmarks) for ABM implementers. This is useful for those proposing new approaches to simulation such as synchronous updating or even, for example, comparing the performance gains of different approaches to concurrency (for example, GPGPU based approaches). A precise definition of Sugarscape now allows researchers the opportunity to try replicate results based upon and agreed and precise definition of the simulation.

5.6 Issues

We have demonstrated that formal methods, as used in computer science, can be used to specify even a complex ABSS such as Sugarscape. The fact that it identified many ambiguities shows that it fulfils its purpose. Formal methods have many years of solid research and a strong theoretical grounding behind them. They are designed to make specification of large and complex systems as easy as possible, for example, by building in modularity.

Our Sugarscape specification shows that an ABM can be formally specified at a very high level. It is possible to incorporate both AU and SU approaches while still remaining very precise. This is a key issue as we want a way of specifying our simulations that allows reproducibility instead of just replicability. Replication can be achieved by just rerunning the existing implementation of an ABSS but this will fail to find many types of error. If, for example, the code contains errors then rerunning the code will just repeat those errors. Reproducibility gives the user enough information to allow them to produce their own implementation with enough confidence that it correctly and precisely interprets the original ABSS definition. Formal methods gives the level of precision required for this to occur.

There are, of course, downsides to the use of formal specification. The price we pay for their precision is a requirement for mathematical formalisms that some modellers may find too difficult or time consuming. To quote a well known paper in computer science "there is no silver bullet" [Brooks, 1987]. The quest for a simple and intuitive natural language-based specification technique is futile. If we want reproducibility then we need the rigour of mathematics. The size of the formal specification produced for Sugarscape may seem large but we must remember that:

1. It is an entire family of simulations with very complex interactions;
2. It is specified twice, once with a single resource (Sugar) and once with two resources (Sugar and Spice);
3. Most other ABSS are smaller and simpler in scope and would require less work. Sugarscape was chosen because its size and complexity shows that formal methods can work with the most complex ABSS.

While the Z notation allowed us to produce an acceptable specification of Sugarscape there are other more recently developed specification languages such as Alloy [Jackson, 2006] or Object-Z [Smith, 2012] that may prove even more effective and produce simpler specifications. Translating from Z to these other notations would allow for a fair comparison to be made between the different specification techniques. Formal specification deserves serious consideration as an approach to solving the replicability issue in ABM. While some may feel it is "overkill" for normal specification there is a strong case for its use in simulations used as benchmarks, tests for the effects of updating techniques (synchronous vs asynchronous) or even implementation methods (GPU vs multicore). The fact that replication is such a problem in ABM shows that there is a need for more the precise simulation specifications only available through formal methods.

5.7 Conclusion

While it is recognised that there is an issue with replication in ABM, it has been pointed out [Peng, 2011, Drummond, 2009] that replication is not the same as reproduction. The difference between replication and reproducibility can be stated succinctly:

> The crux of the matter is that reproducibility requires changes; replicibility avoids them. A critical point of reproducing an experimental result is that irrelevant things are intentionally not replicated. One might say, one should replicate the result not the experiment. [Drummond, 2009]

Simply forcing authors to publish their model's implementation will not give us reproducibility. If their models are incorrect then just replicating them will just replicate their errors. What is required is more along the lines of the model-driven approach [Edmonds and Hales, 2003]. The inclusion of extra data alongside the published results will obviously help but it is not enough. It should be possible to reproduce previous results based on the model definition using a different implementation.

What is missing is the ability to specify the model in a clear and precise manner such that independent reproduction of a simulation becomes possible and ambiguities and errors in the model can be revealed. One approach in computer science that fits this criteria is formal specification where the model is specified with mathematical rigour and can be mathematically verified.

Although [Edmonds and Hales, 2003] states that experimentation is the only way to verify a model this does not preclude, but rather requires, formal specification of the model so we can prove our implementation is a correct interpretation of this model. Experimentation only works if we have a precise description of the model we are experimenting with.

We have produced a formal specification of the Sugarscape family of simulations. It is, to the best of our knowledge, the first formal specification of the entire Sugarscape simulation family. The purpose of the specification is to provide a clear, unambiguous and precise definition of Sugarscape. The specification has identified many ambiguities and missing information in the original rule definitions. Where there is an obvious way of removing these ambiguities we have done so. If there was more than one possible solution we have identified them and chosen the most likely one.

Both the AU and SU versions of Sugarscape are derived so either or both approaches can be used and compared. Comparing the results enables the simulation to be checked for *artifacts*. Artifacts are results that are only due to the updating strategy employed and not intrinsic to the model itself. The ability to check for artifacts is essential to allow us to have confidence in any results produced.

Because our specification is high level and only defines the before and after state of each rule it makes no assumptions as to how any rule will be implemented. Implementers have complete freedom as to what programming model they employ (Object-Oriented, Imperative, Functional, or any concurrent approach). Any simulations that adhere to the specification can be properly compared in terms of performance or patterns of behaviour. This will put on a firmer foundation any claims made by researchers about their implementations.

This specification has been made available [Kehoe, 2015a] for anyone who wishes to use it and provides a standard reference that researchers can use when producing their own implementation.

It will prove invaluable, for example, to researchers who advocate the use of a GPU [Deissenberg et al., 2008, Lysenko and D'Souza, 2008, Richmond et al., 2009], containing hundreds to thousands of individual processors. The more complex rules in Sugarscape (such as *Combat*, *Inheritance* and *Mating*) are not easily parallelized. By providing a precise and full set of these rules it is now possible for researchers to properly compare how different models cope with the more complex and realistic ABMs. Up until now it has been unclear as to which version of each rule has been employed thus making comparisons impossible.

In the next chapter we derive algorithms that allow synchronous updating to be applied to complex ABM and ABM. Synchronous updating, commonly used in CA, can then also be used in ABSS. This will allow modellers to compare and contrast the effects of these updating methods on their simulations. Repeating results with multiple updating strategies gives more confidence in any obtained results.

Chapter 6

Extending Synchronous Updating to Agent Based Modelling

6.1 Introduction

A simulation imitates the operation of a system over time. Development of a simulation is dependent on having a well defined model of the system being simulated, where this model encapsulates the key characteristics or behaviours of the system. An ABM is a simulation where individuals (also known as *agents*) within the system and their interactions are explicitly represented. Global properties are not explicitly modelled but emerge from the local interactions between populations of agents.

The asynchronous approach to simulating the real world is espoused by Huberman and Glance [Huberman and Glance, 1993] where they distinguish between SU and AU and defines the differences between them.

The synchronous approach is defined therein as having a global clock that synchronises the updating of all agent states so that all updates occur in unison or, in other words, simultaneously. Asynchronous approaches, on the other hand, have no global clock and updates of agent state do not occur simultaneously. In effect, AU applies individual agent interactions sequentially in some random order. There are various approaches that determine how this random ordering is decided [Cornforth et al., 2005]. It has been demonstrated that synchronous and asynchronous implementations of the same simulation can result in widely differing behaviours [Huberman and Glance, 1993].

it is sometimes assumed that AU is more realistic in that it correlates more closely with the reality being simulated but not everyone agrees that this is the case [Fatès and Chevrier, 2010]. In any event the lack of a SU algorithm that encapsulates the range of behaviours occuring in an ABM makes this point moot. As SU algorithms have only been available for CA based simulations it has not been possible to compare SU and AU implementations of ABM based simulations.

We define accuracy in terms of how closely a simulation replicates the real-world system being modelled. For a simulation to be accurate anything that is not allowed within the system being modelled must also be excluded from the simulation.

It is known that AU and SU can give different results within CA but it has not been shown to what extent they differ in ABM based simulations. Here some important differences between AU and SU are discussed. Then we produce novel SU algorithms and show using asymptotic complexity that the algorithms have comparable, if not equivalent, space-time complexity to the equivalent AU algorithms. These new SU algorithms can handle the full range of interactions that occur in ABM thus opening up ABM simulations to SU and allowing comparisons between the different updating strategies to be made.

6.2 Distinguishing Sequential and Concurrent Updating

The state of an ABM, say W, is defined as the set of agents (locations are types of agent) that make up the system being modelled

$$W = \mathbb{P}\, Agent \tag{6.1}$$

Agents are composed of a set of attribute-value pairs, sometimes known as properties. Each attribute is uniquely identifiable and its associated value can be either mutable or immutable. A mutable attribute-value pair may have constraints placed on the values it can take. The value of an attribute x belonging to an agent A is denoted "$A.x$".

In any ABM there must exist a function, τ, that can tell how far apart any two agents are. It calculates this by comparing various attributes of the two agents in some predefined manner.

τ takes as input any two agents and returns a natural number ($\mathcal{N}$) representing the "distance" between them (equation 6.2). There is no requirement that τ be commutative but non-commutative τ functions are very unusual (that is, it is generally the case that distance from a to b is always the same as distance from b to a). This function defines the topology of a model[1].

$$\tau(Agent \times Agent) \rightarrow \mathcal{N} \tag{6.2}$$

All agents have an attribute called $range$ which holds an integer value ≥ 0. For every agent a we define a set containing all agents within its locality (also known as its neighbourhood) where locality is defined using an agent's $range$ attribute:

$$\mathcal{L}_{a,W} == \{x \in W \mid \tau(a,x) \leq A.range\} \tag{6.3}$$

In ABM it is always the case that each agent's locality is a small subset of the entire world (Equation **??**).

$$\forall\, a : Agent, W;\ \mathbb{P}\, Agent \bullet a \in W \Rightarrow \mathcal{L}_{a,W} \subset W \tag{6.4}$$

In fact each agents neighbourhood tends to compose a tiny fraction of the world being modelled: $|\mathcal{L}_{a,W}| \ll |W|$ in all ABMs. It is an important principle in ABM that agents can only access local information when making decisions. Not allowing agents access to global information enforces a property known as *bounded rationality*. Bounded Rationality is enforced by only allowing agents directly interact with, and have direct knowledge of, agents within its locality. Overall simulation properties, known as *emergent properties* are caused by individual local interactions. An agent is allowed to get indirect knowledge of things outside its local neighbourhood through information transfer across localities e.g. gossip, newspaper, television, etc. But this knowledge takes time to arrive, that is, it is time delayed. Information takes time to move through an ABM. I can see my neighbours state as they are now (instantaneously or within this time step) but I cannot see the state of non neighbours instantaneously. My knowledge of non local agents is always out of date due to the time it takes information to get to me.

One of the defining properties of an ABM is that all state changes occur through individual agent interactions. An agent interaction rule δ (see Equation 6.5) defines how the behaviour of each agent affects the world. It takes in the agent that is undertaking the behaviour and the current state of the world and then updates the state of the world by:

1. Updating the state of the agent the rule is applied to;

[1]Here we use standard mathematical notation for functions, not Z notation.

2. Possibly updating one or more other agents within the neighbourhood of the agent applying the rule.

$$\delta : Agent \times \mathbb{P}\, Agent \rightarrow \mathbb{P}\, Agent \tag{6.5}$$

When a behaviour δ is applied to an agent a in some world W the following must apply:

1. It cannot update any agents not in its neighbourhood (Equation 6.6). Equation 6.6 states that if an agent b is not in my neighbourhood then any action I take now cannot directly affect b. My actions may have future consequences for b but these are indirect results of the current action. This is the principle of locality - all interactions are local;

2. The outcome of an agent's behaviour cannot be influenced by any other agent not in its neighbourhood (Equation 6.7). Equation 6.7 states that anything occurring outside my neighbourhood **now** is unknowable by me **now** so cannot affect how I act **now**. An agent cannot be *directly* aware of any agents outside its neighbourhood.

$$\forall\, a, b : Agent;\ \forall\, W_i, W_k : \mathbb{P}\, Agent \bullet (b \in W_i \wedge b \notin \mathcal{L}_{a,W_i} \wedge \delta(a, W_i) = W_k) \Rightarrow b \in W_k \tag{6.6}$$

$$\forall\, a : Agent;\ \forall\, W_i, W_k : \mathbb{P}\, Agent \bullet \mathcal{L}_{a,W_i} = \mathcal{L}_{a,W_k} \Rightarrow \delta(a, W_i) = \delta(a, W_k) \tag{6.7}$$

These restrictions are essential properties of any ABSS. Without these restrictions agents have access to non local information and that is not allowed.

$$\Delta : \mathbb{P}\, Agent \rightarrow \mathbb{P}\, Agent \tag{6.8}$$

The state transition function Δ (Equation 6.8) represents a single time step in the model. During each step all agents simultaneously take some action (that is, apply δ). This is implemented by applying the agent interaction rule, δ, to every agent (equation 6.9). This is also true if we are using a DES implementation. In that case the behaviour will revert to the identity mapping id (do nothing) for any agent whose timestamp is not the minimum timestamp in the World W.

$$applyRule : \mathbb{P}\,Agent \times \delta \to \mathbb{P}\,Agent \tag{6.9}$$

A simulation run of a model is a sequence of model states where the first state matches the model initial state and every other state is computed from the previous state using the state transition function Δ (equation 6.8).

If $M = (W_0, \Delta)$ then $[W_0, W_1, W_2, ..., W_N]$ is a valid simulation run iff $\forall\, i : 0 \leq i < N \bullet W_{i+1} = \Delta(W_i)$. That is, there is a valid transition between each adjoining pair of states. If the model is stochastic there may be many different valid simulation runs starting from the same initial state. A stochastic transition function is one where there is more than one valid outcome from applying the function to the starting state For example, agents might be allowed to move at random and can have a number of equally plausible destinations during any step.

There are a number of different ways of implementing an agent behaviour in a simulation, for example, different AU variants or SU. It is up to the modeller to choose a particular approach that correctly implements the transition function. $applyRule$ is a correct implementation if its output matches that of our transition function Δ as per equation 6.10.

$$\forall\, W : \mathbb{P}\,Agent \bullet applyRule(W, \delta) = \Delta(W) \tag{6.10}$$

An AU version of $applyRule$ will update each agent in some order (Equation 6.11). Here we see the first parameter identifies the sequence in which the agents are to be updated. This sequence is generated using a well defined method that is defined by the particular AU variant employed.

$AUApplyRule$ is defined recursively. The base case occurs when there are no agents to apply the rule to so the state remains unchanged (Equation 6.12). Otherwise we apply δ to the first agent in our sequence producing an updated simulation state and then we recursively call $applyRule$ with the remaining agents and the updated state (Equation 6.13).

$$AUApplyRule : \text{seq}(Agent) \times \mathbb{P}\,Agent \times \delta \to \mathbb{P}\,Agent \tag{6.11}$$

$$applyRule([], w) = w; \tag{6.12}$$

$$applyRule([a_1, a_2, \ldots, a_n], w,) = applyRule([a_2, \ldots, a_n], \delta(a_1, w)) \quad (6.13)$$

For $AUApplyRule$ to be a correct implementation of a set of concurrent events it must be the case that the order in which the rule is applied to the agents in the model does not affect the outcome of the rule (as defined in Equation 6.14). If application of the implementation has results that depend on the order in which the rule is applied to individual agents then it does not reflect the concurrent nature of a step. This is what leads to artefacts in a simulation. The sequence of behaviour applications does not exactly match the concurrent nature of the system being modelled.

$$\begin{aligned}
&\forall\, a, b : \mathrm{seq}(Agent);\ \forall\, W : \mathbb{P}\, Agent \bullet \\
&\qquad (\#\, a = \#\, b = \#\, W \wedge \mathrm{ran}(a) = \mathrm{ran}(b) = W) \Rightarrow \\
&\qquad\qquad applyRule(a, W, \delta) = applyRule(b, W, \delta) \quad (6.14)
\end{aligned}$$

6.3 DES versus ABM

Although we are concerned here with ABM, the issues concerning synchronous and asynchronous implementation are also applicable to DES. There is debate about the relationship between ABM and DES. In [Brailsford, 2014] it is stated that DES is a subset of ABM while [Onggo, 2010] claims that any ABM can be translated into an equivalent DES. In any case the two approaches have much in common [Siebers et al., 2010] and the issue of synchronous or asynchronous implementation of events is relevant to both approaches.

The use of Sugarscape as a testbed for our approach has the added advantage that as well as being one of the best known ABSS, it has the distinction that it has been implemented using both approaches: time-stepped (aka *clockwork*) and more recently DES [Zaft and Zeigler, 2002]. Thus any findings from investigating Sugarscape can also be more easily checked to see how they apply to DES.

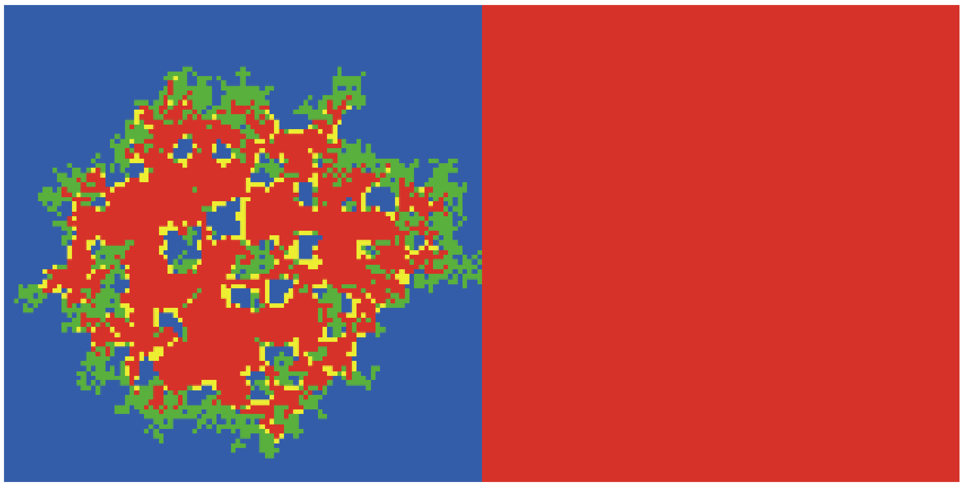


Figure 6.1: Spatial Iterated Prisoners' Dilemma: Synchronous on left, Asynchronous on right

6.4 Asynchronous versus Synchronous

Synchronous simulations are defined as those in which all the agents in that simulation are updated simultaneously and instantaneously at each time step. Each step is a discrete quantum of time and the simulation progresses as a sequence of discrete states, one per time step. The state at step n is dependent solely on the state at step $n - 1$. The best known simulation of this type is Conway's Game of Life [Gardner, 1970]. This is implemented by employing two copies of each agents state. One copy represents the state of the agent as it is now (the *current* state). The other state represents what its state will be at the start of the following step in the simulation (the *next* state). When agents are applying a rule during a step they see only the current state of their neighbours and any updates made are stored in the *next* state until every agent has completed the rule. Once everyone has completed the rule application based on the *current* state and stored the results in the *next* state then the *current* state is made equal to the value contained in the *next* state.

Using the synchronous approach, Nowak developed a simulation of Spatial Iterated Prisoners' Dilemma [Nowak and May, 1992] and showed that the simulation generates chaotically changing spatial patterns, in which *cooperators* and *defectors* both persist indefinitely. Huberman and Glance [Huberman and Glance, 1993] used AU on the same simulation to instead show that the simulation always evolves, within 100 generations into a steady state where all agents become *defectors* (see

Figure **??**). Thus we have two clearly contradictory results deriving from the application of the same rule in the same simulation that differ only in the updating technique.

[Huberman and Glance, 1993] drew the conclusion that to mimic continuous real world systems we need a procedure that ensures the updating of interacting agents is continuous and asynchronous. This asynchronous model is implemented by:

> choosing an interval of time small enough so that at each step at most one individual agent is chosen at random to interact with its neighbours. During this update, the state of the rest of the system is held constant. The procedure is then repeated throughout the array for one player at a time, in contrast to a synchronous simulation in which all the agents are updated at once. [Huberman and Glance, 1993]

In line with this, [Caron-Lormier et al., 2008] claimed that AU is more realistic (i.e. more closely resembles the real world). Other researchers have claimed that synchronous behaviour is rare in the real world [Cornforth et al., 2005] and that AU is more realistic [Newth and Cornforth, 2009]. This is disputed by researchers [Fatès and Chevrier, 2010] who state that AU is suitable only for instantaneous events which do not occur in, for example, biological systems. In any event the debate about the appropriateness of any one approach is ongoing and has yet to be settled.

The precise ordering used by asynchronous updating (dependent on which AU scheme is employed) affects the outcomes as well [Ruxton and Saravia, 1998] but this is often ignored in ABM [Radax and Rengs, 2010].

It is standard practice to use the asynchronous approach for ABMs and in particular Agent Based Social Simulations (ABSS). AU has been adopted as standard by the major agent toolkits such as NetLogo, Repast, Mason and Swarm [North et al., 2013b, Luke et al., 2005, Berryman, 2008]. Standard ABSSs, such as Sugarscape [Epstein and Axtell, 1996], assume an AU implementation. The ability to execute a sequence of agent actions in a random order, an essential part of asynchronous simulations, forms part of *StupidModel* [Railsback et al., 2005], a suite of models designed to test the suitability of any toolkit for ABM development.

The prevalence of AU is due to the lack of any SU algorithm that can handle the complex interactions that occur in ABMs. This contrasts with CA based simulations where, because the interactions are simpler, synchronous algorithms exist and have proven popular. CA-based Simulations of traffic flow, a real world system, employ SU [Burstedde et al., 2001] as standard. It is also interesting to note that it is common to see CA based simulations employ both updating methods and compare the results [Bezbradica et al., 2014, Bach* et al., 2003]. Even in CA, where these comparisons are more common, the effects of the two updating techniques are not well understood [Grilo and Correia, 2011].

The lack of SU algorithms suitable for ABM has meant that AU is used exclusively in this area. This has made comparisons between AU and SU results in ABM impossible to make. The new SU algorithms developed in this thesis provide modellers with the ability to run AU and SU based ABMs side by side for comparison (something that is commonly done with CA based simulations). The difference between AU and SU has only properly been investigated in CA and it cannot be assumed that these results also hold for ABM without further investigation. Comparing AU and SU in a simulation gives confidence in any results obtained. It can demonstrate that results are not artefacts of the updating strategy and will help demonstrate repeatability of simulation results.

6.5 Instantaneous Information Leakage between Neighbourhoods

Bounded rationality is the idea that in decision-making rationality of individuals is limited by the information they have available to them, the cognitive limitations of their minds, and the finite amount of time they have to make a decision. Many consider this an essential property within ABSS [Epstein, 1999]. Within ABM this principle is generally taken to mean that agents do not have access to global information. This principle of bounded rationality is enforced in ABM by ensuring that *an agent can only be aware of the state of other agents within its neighbourhood (locality).* This places limits on the information agents have available at any one time. They can see their locality as it is now and gain information about the world outside their locality through time delayed information transfer (e.g. gossip).

Locality is, of course, defined in a simulation specific way. In some cases locality resembles physical proximity in the real world but in others (for example, simulating Facebook or Twitter connections to study how memes traverse social networks) it does not. Here locality is defined information theoretically. Anyone who I receive information from in real time (instantaneously) is in my "locality" or "neighbourhood". If I am undertaking a video conference call with colleagues then the people physically present in the room with me as well as the people on screen (even though they are physically removed from me) are in my locality but everyone else outside of the room is not.

A local neighbourhood may change from step to step but within each time step it is fixed. The results of actions taken during the current step can change the neighbourhood of an agent for the following step. This principle guarantees that an agent cannot be omniscient, that is, an agent cannot be aware of the global state as it is *now* or have complete knowledge of the universe. If an agent is not in my immediate neighbourhood then I cannot know what it is doing now. I may find out at a later time what it was doing now through time delayed information diffusion (e.g. gossip) but that is time delayed information. SU guarantees bounded rationality by imposing a consistent speed for the transmission of information across a simulation space. If this property is important in a particular ABM then more consideration should be given to using SU.

Take as a *gedankenexperiment* or thought experiment the case of a simulation where agents perform some action as soon as they become aware that some particular agent A is dead. An agent B can become aware of this fact under two conditions:

1. If B is a neighbour of A and witnesses A's death;
2. If B is not a neighbour of A but one of B's neighbours is aware of A's death and informs B.

Now once A dies the amount of time steps it takes for any other agent to find out should vary proportionally based on how far away that agent is from A. Agents in the immediate locality of A should be first to know (as they witness A's death) and this information should then percolate through the system from local neighbourhood to local neighbourhood over one or more subsequent steps. Under SU this is exactly what happens. Now consider what happens under AU if we are an immediate neighbour of A. Once A dies in step i when do we find out? It depends on the order in which the actions occur *during* the step. If A's behaviour is scheduled to occur before ours in this step then we will be aware of A's death within this step. However, if A's behaviour is scheduled after ours then we will not find out until the following step. It is clear that on average half of A's neighbours will be aware of its demise immediately and half will not!

This random sequencing of actions means that agents **not** within A's neighbourhood can also be aware of A's change of state instantly (within the same time step). We can construct a specific sequence of an update ordering $[a_1, a_2, ..., a_n]$ where each agent in the sequence a_i runs before a_{i+1} within this step and every a_i is a neighbour of a_{i+1}. If a_1 dies then every agent in the chain will pass this information on to their neighbour within the same step. This gives a_n instant access to information from outside of its neighbourhood. This is more likely to happen in simulations with high population densities where continuous chains of neighbouring agents can form which may help explain the conclusions in [Caron-Lormier et al., 2008] that high population densities enhance the differences in outcomes between asynchronous and synchronous versions of the same ABM.

There are two related issues here:

1. Agents who are not within the locality of A can be immediately aware of the change of state of A thus violating bounded rationality;
2. Agents further away from A can be aware of A's change of state before agents closer to A are (inconsistent speed of information transfer through the system).

Any simulation where bounded rationality and/or consistency in the speed at which information spreads through the simulation space are necessary properties should therefore consider using SU to ensure that any results are not due to artefacts of the AU strategy. Of note here are two findings from other researchers.

[May, 1973] found that delay is a potential contributor to periodicity in systems. AU can interfere with, or even remove, the delay that we would expect in any system with bounded rationality (remember bounded rationality is enforced by the fact that agents can only view their immediate neighbours). Any natural systems where this delay is present would need to be cautious about employing AU or employ SU alongside AU for comparison. Our analysis explains why SU is the preferred updating strategy in traffic flow and pedestrian dynamics simulations as it picks up the important periodic changes in overall state (e.g. traffic jams caused by cars slowing as they pass an accident) by properly modelling the flow of information through the system. It also helps explain the stark differences produced by the two versions of the Spatial Iterated Prisoner's Dilemma as the simulation had a maximal population density of one agent per location;

High population densities enhance the differences in outcomes between asynchronous and synchronous versions of the same ABM [Caron-Lormier et al., 2008]. It may be that there is a connection between the fact that in high density populations violations of bounded rationality are more likely to occur thus giving rise to larger differences in outcomes.

To show how this can affect an ABSS let us look at another simple illustrative example, that of *White Tailed Deer* herd behaviour [Hirth and McCullough, 1977]. White Tailed Deer use their tail to warn members of the herd of impending danger. When sensing danger, the deer raises its tail this is called *flagging*. Showing this large white patch on the underside of the tail signals an alarm to other deer and helps a fawn follow its mother to safety. We can simulate herd behaviour using the simple rule:

- If a deer sees one or more of its neighbours raise their tail(s) or it senses danger it immediately raises its tail and runs in either the opposite direction of the danger (if it sensed danger) or in the same direction as the deer with their tails raised (if it saw tails raised); otherwise it grazes.

We expect that once a deer senses danger the alarm will spread throughout the herd in enlarging concentric circles (for simplicity we assume all deer are paying attention to their surroundings). This is what will happen in a synchronous approach but AU will allow the information to spread inconsistently through the herd. Deer furthest away from the initial locus of alarm could be alerted immediately while deer in the immediate vicinity of the alarm might not be alerted during this step. It is even possible to come up with sequences where that half of the herd nearest the alarm are unaware of an danger while the half of the herd furthest away are aware.

When considering which updating strategy to use careful consideration should be given to how important bounded rationality (as enforced through locality) is in the system being simulated and whether inconsistent propagation speed of information through the simulation can adversely affect the outcomes.

6.5.1 Probability of Instantaneous Information Transfer Accross Local Boundaries

It is clear that instantaneous information transfer accross local boundaries (we term such events *leakages*) can be caused by AU but to determine how this affects simulations we need to know how often such leakages occur.

For a leakage to occur a chain of three (or possibly more) agents must be updated in an order that allows information to move across neighbourhood boundaries in a single timestep. The amount of leakage will be directly proportional to the number of such chains occuring during each timestep.

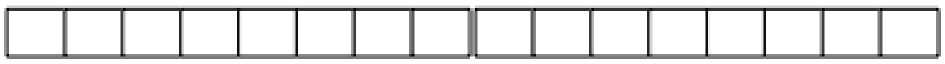


Figure 6.2: One Dimensional Lattice

To help calculate the liklihood of leakages occuring we make some basic assumptions:

- The lattice in which the ABM runs is a torus structure. That is, it wraps around the edges. This is true of Sugarscape and not uncommon in general. This asumption makes the following calculations simpler but the use of a non torus structure does not have a major effect on the outcome of the calculations;
- The locality of an agent is determined by a range R. That is, all locations within R steps of the current location are within its locality. We also assume that $R < N/2$ where N is the dimension of the simulation lattice. This is true in all but the most trivial ABM;
- Agents use a *von Neumann* neighbourhood when calculating locality. In a one dimensional lattice the neighbourhood extends to the left and the right. In a two dimensional lattice the neighbourhood extends in four directions (left, right, up and down);
- Agents are distributed evenly throughout the lattice;

We will return to reexamine the implications of the last two assumptions after we make our calculations.

One Dimensional Lattices

To begin with we will limit ourselves to one dimensional lattices (see Figure 6.2). Each location has neighbours only to its left and right. Leakage in this case can only occur between a minimuim of three locations where these locations have the following properties:

- The first location, A, is a neighbour of the second location B;
- The second location B is a neighbour of the third location C;

- The third location C is **not** a neighbour of A.

We will label any three locations that satisfies these properties a *Leakable Triple*. Such a chain of locations will be updated under AU in one of the following orders

1. A B C;
2. A C B;
3. B A C;
4. B C A;
5. C A B;
6. C B A.

Two of these six possibilities result in leakage (sequences 1 and 6 only) while the other four of the six possibilites (sequences from 2 to 5) do not. In a one dimensional lattice of N locations we can determine the number of such chains as follows:

We first need to enumerate the total number of possible *leakable triples* in the lattice. Then we need to calculate how many of these possible leakable triples have all three locations occupied by agents. Finally, we need to calculate how likely each occupied leakable triple is to be updated in an order that causes leakage. This, as we have seen above, is $\frac{2}{6}$ or 33%.

Every leakable triple has a rightmost location defined by the lattice dimensionality. Therefore if we can calculate the number of leakable triples that have a location L_i as a rightmost location then we can use this to calculate the total number of leakable triples on the lattice (by multiplying the number of leakable triples with a fixed rightmost location by the total number of locations in the lattice).

For any starting location A there will be R locations within its locality (or neighbourhood) to the left.

- The first such location (call it B) is one step away and will only have one other location (call it C) within its range but outside of A's neighbourood;
- The second location to the left of A is two steps away and this location will have two other locations within its range and outside of A's neighbourhood;

- The third location to the left of A is three steps away and this location will have three other locations within its range and outside of A's neighbourhood;
- The i^{th} location ($i < R$) is i steps away and this location will have i other locations within its range and outside of A's neighbourhood;
- The R^{th} location is R steps away and this location has R other locations within its range and outside of A's neighbourhood.

In total we can see that this gives a total of $1 + 2 + ... + i + ... + R$ (or $\Sigma_{i=1}^{i=R} i$) chains for this one rightmost location. In general terms we can see that this gives a total of $\frac{R(R+1)}{2}$ leakable triples with the same rightmost location. Since there are N locations in the lattice the total number of chains must be N times this number:

$$N \times \frac{R(R+1)}{2} \tag{6.15}$$

Each chain is relevent only if all three locations contain agents. If there are A agents on a lattice of size N then the probability of a location being occupied is $\frac{A}{N}$. We let P represent this probability of a location containing an agent, $P = \frac{A}{N}$. For CA, it is always the case that $P = 1$ as each location is also an agent ($A = N$) but for an ABM this figure will be lower as the number of agents will be less than the number of locations. The probability of all three locations within a chain containing agents is P^3. Therefore the number of occupied chains is this times the total number of possible leakable triples:

$$P^3 \times N \times \frac{R(R+1)}{2} \tag{6.16}$$

We know that the possibility of AU updating such a chain in a way that preserves locality is $\frac{2}{3}$ so the probability of locality being preserved throughout the entire lattice is:

$$(\frac{2}{3})^{P^3 \times N \times \frac{R(R+1)}{2}} \tag{6.17}$$

It immediately follows that the possibility of leakage is then given by equation 6.18:

$$1 - (\frac{2}{3})^{P^3 \times N \times \frac{R(R+1)}{2}} \tag{6.18}$$

As R and P increase in value to the probability of leakage reaches 1 *during each step*.

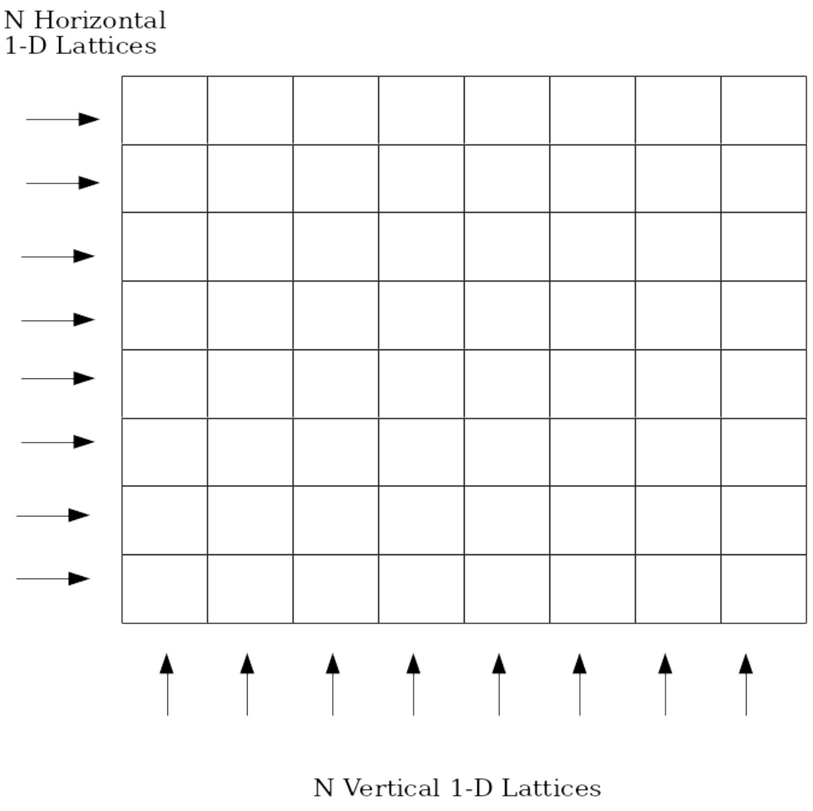


Figure 6.3: Two Dimensional Lattice

Two Dimensional Lattices

Extending this to a two dimensional $N \times N$ lattice we can see that this lattice consists of N horizontal one dimensional lattices of size N as well as N vertical one dimensional lattices of size N (see Figure 6.3). Combining these we get a total number of possible chains equal to:

$$2 \times N \times N \times \frac{R(R+1)}{2} \tag{6.19}$$

Therefore the probability of no leakage occuring in an $N \times N$ lattice is:

$$(\frac{2}{3})^{P^3 \times N^2 \times R(R+1)} \tag{6.20}$$

From this we can again deduce that the probability of leakage during each step is (equation 6.21):

$$1-(\frac{2}{3})^{P^3\times N^2\times R(R+1)} \tag{6.21}$$

In any CA it is always the case that $P=1$ and generally the case that $R=1$ making the probability of no leakage $(\frac{2}{3})^{2N^2}$. For any reasonable value of N therefore the probability of leakage occuring approaches 1. In an ABM on the other hand $P<1$. Remembering that $P=\frac{A}{N}$ we can see that in general by inserting $\frac{A}{N}$ for P and simplifying we get:

$$(\frac{2}{3})^{\frac{A^3R(R+1)}{N}} \tag{6.22}$$

Sugarscape, as can be seen from the work on carrying capacity in Chapter 7, can have a value for P as low as $\frac{2}{25}$. Based on this we can see that leakage is much more likely to occur in a CA than an ABM.

Of course this is based on the assumptions listed at the start of this section. These assumptions are reasonable but there are some final points that we should note:

- The assumption that agents are distributed evenly throughout the lattice is open to question. In an ABM this is not necessarily the case. Often the simulation will have clustering of agents into groups resulting in a higher likelyhood of leakage within these groups and less leakage between groups;

- ABM require inter agent communication for overall behaviours to emerge. The liklihood that we will design an ABM that has little communications between agents is low. Most simulations will have a large amount of interagent communication thus leading to increased probability of leakage;

- If a *Moore* neighbourhood is used instead of a *von Neumann* neighbourhood then the probability of leakage will increase.

This analysis leads to the conclusion that leakage is a bigger issue in CA than ABM and this is in fact bourne out in Chapter 7, as we will see later.

6.6 Conflict Resolution

If two agents are competing for the same resource during a single step then only one can be successful in winning that resource. For example, if two agents want to move to the same location only one can succeed. If the simulation employs AU then the winner will be the agent that was updated first during the step. Since the order that agents are updated in during a step is determined randomly the winner will always be decided randomly or by whatever scheduling method is employed by the asynchronous approach. In any case it is determined not by the rule definition itself but by the choice of updating algorithm.

SU, on the other hand, will detect all conflicts and a conflict resolution mechanism can be employed to pick a winner. If the conflict resolution mechanism is left undefined then the winner may be chosen at random, as in the case of AU. But the possibility exists within SU of a more logical mechanism for dealing with the conflict. For example, in the case of two agents trying to occupy the same location then the conflict resolution rule can choose the agent nearest to the location to be the winner, a *closest wins* strategy. This is a rational choice, arguably a more accurately model of the real world that also preserves determinacy.

The collision resolution mechanism employed can be an arbitrary algorithm (either simple or complex). The collision resolution algorithm chosen will effect the outcome of agent interactions and so must be an important part of a rule's definition. It is possible that the same rule can result in different emergent properties due to the collision resolution algorithm used. For example the *closest wins* strategy defined above is arguably a more realistic simulation of movement as it occurs in the real world. If two agents try to move to the same destination then, all things being equal, the closed to that destination will arrive there first and win.

When collision resolution occurs between any two agents, one agent will be the winner and succeed while the other will fail. One question we may ask is what happens to the loser? Has it forfeited its move for this time step? This is what we call the *No-Op* option. In many cases this may be the most realistic option but we need to be aware of its consequences. A *No-op* approach can result in agents being edged out of existence due to one bad move. Take for example a move rule that states that in case of collision the agent with the most sugar reserves wins (*Fittest wins*). An agent who loses out during one turn has now lost a move and not had the opportunity to replenish its sugar reserves. This will have a knock on effect in that the agent will have even less sugar in the next time step and be more likely to lose out in each following time step until it starves. In certain situations this makes sense but there is an alternative. Instead of a *No-Op* rule we can allow losing agents to *Redo* their move thus ensuring that every agent gets to make a move during each time step. This is an example of collision resolution introducing the notion of fairness into a simulation. Every agent gets to move in each step but collisions can be handled in any manner that we wish. This is in opposition to AU where collisions can *only* be resolved randomly.

Collision Resolution, on the other hand, can be used to ensure that:

1. Rules more closely imitate real life. For example, *Closest Wins* strategy in Movement;
2. Every agent runs as often as every other agent. For example, *Fittest wins* strategy in Movement with *Redo*;
3. Certain agents are prioritised over others by virtue of some property. For example, *Fittest wins* strategy in Movement with *No-Op*;
4. No one agent is privileged over any other agent;

This facility to define collision resolution as part of the behaviour definition is not available with AU. AU avoids collisions throughout the sequential ordering of agent moves within each time step. These orderings are usually randomised in some fashion and as a result fairness in AU is restricted to the much weaker notion that statistically over a large number of time steps these random orderings will ensure each agent will win as often as it loses.

SU allows finer detail to be added to any rules within an ABM and much more scope for fairness.

6.7 Rule Classification

Flow of information is used to classify interaction types. This property identifies the direction of information flow between the agent executing the action[2] and its neighbouring agents.

Flow

The *information flow* property can have one of four different values based on how that rule accesses the attributes of agents. We explain the four types of flow below:

No Flow This is the simplest type of behaviour. Here an agent a updates its own attributes without reference to any of its neighbours. Each agent acts com-

[2]Action and Behaviour are used interchangeably.

pletely independently of any other agent. We call these behaviours *Independent*.

The *Growback*[3] rule in Sugarscape causes each location to increase its *Sugar* resource by a fixed amount up to some predefined maximum during each step.

Unidirectional Unidirectional behaviours can be subdivided into two types based on the direction of the flow of information:

Pull A behaviour $\mathcal{B}$ that reads the attributes of the agents in the neighbourhood of agent a but only writes to the attributes of agent a is defined as having "pull" flow. Information flow is one way from a's neighbours into a. That is, the states of the neighbouring agents are read-only and only the agent performing the action can update its state.

In Sugarscape the diffusion rule allows a location to read the pollution values from its neighbours and use them to update its own pollution attribute so that it equals the average pollution level of its four neighbours.

Push These behaviours are the opposite of pull behaviours. Here an agent a reads its own state and writes the state of one or more of the agents in its neighbourhood. It pushes information from the owner to its neighbours. If we can avoid collisions, that is, two or more agents trying to write to the same agent simultaneously, then it is possible to restate a push rule as an equivalent pull rule [McCool et al., 2012].

The *Disease transmission* rule in Sugarscape makes an agent transmit to each of its neighbours one of its current diseases. We can convert this into a *pull* rule by restating it as: an agent receives one disease from each neighbour. Any behaviour that is, or can be, converted into a *pull* behaviour we will refer to as *Read-Dependent*.

Bidirectional Bidirectional behaviours combine the characteristics of both push and pull rules. Information is exchanged, or flows, in both directions, from the agent a performing the behaviour to its neighbours and from the neighbours back to a. A mutual agreement, in the form of a handshaking protocol, between the groups, most typically pairs, of agents involved in the exchange must be arrived at. These are the most complex rules.

[3]All examples are taken from Sugarscape as described in chapter 4.

Agent Trade is an exchange rule that requires both agents to agree terms, such as what items are to be traded and what is the agreed price, before exchanging resources.

These behaviours we refer to as *Write-Dependent*. These behaviours are the most difficult to implement synchronously. All behaviours that are not *Independent* or *Read-Dependent* are *Write-Dependent*.

Any proposed SU algorithms must be able to handle all these different categories of interaction.

6.8 Synchronous Algorithms

In this section we will provide synchronous updating algorithms for all possible agent interaction types. Each algorithm will be presented alongside an asymptotic analysis of the algorithm. For each algorithm a shared memory concurrent version of the algorithm is also presented and analysed. In the case of the most complex behaviour type (*Write-Dependent*) a tiling approach suitable for multiprocessors is briefly discussed.

We have partitioned all behaviours into three different types: *Independent*, *Read-Dependent* and *Write-Dependent*.

Independent Independent behaviours are behaviours where an agent does not interact with any outside agent. That is, it is entirely self contained and there is no shared state. These are the simplest type of agent behaviour. An example from Sugarscape is $Growback$ where each location increases the amount of sugar it contains by a fixed amount during each step. In this case each agent applies this rule independently of any other agent in the simulation.

Read-Dependent Read-Dependent behaviours are those where an agent reads the state of one or more other agents but does not cause any updates to their states. The $Culture$ rule is a good example where each agent reads the state of its neighbours culture in order to update its own.

Write-Dependent Write-Dependent behaviours are the most complex. Agents with these behaviours can read and also write to the state of other agents. The $Combat$ rule in Sugarscape allows an agent to attack another agent thus changing its own state (e.g. health and position) and the state of the agent it is attacking.

We now provide synchronous algorithms for each category.

6.8.1 Independent Actions

In some cases actions can be expressed naturally in a synchronous manner. Rules that involve no interactions between agents[4] are trivially implementable synchronously. *Growback*, for example, states that each location increases its sugar resource by a set amount during each time step. Because each agent updates itself independently of any other agent the order in which these updates occur has no effect on the outcome. These actions are deterministic by nature. It is clear that there can be no differences in outcomes between a synchronous and an asynchronous interpretation of an independent rule. As each agent performs its action in isolation, the order of execution of the agents will have no effect on the outcome. This is the only category of action that this holds true for, all other categories give different results when asynchronous updating is employed.

Algorithm

Each agent is assumed, for the sake of exposition, to hold its state in the variable $state$. The same algorithm can be used for the synchronous and asynchronous approaches (see Algorithm 1). $applyRule$ performs the action based on the rule definition and the agent's current state returning the new state (line 2). We assume a random ordering but any update order will do.

Algorithm 1 Independent Action

```
Input: $Agents : list\langle agent \rangle$
1: for all $a \in Agents$ do
2:     $a.state \leftarrow applyRule(a)$
```

Asymptotic Analysis

It should be clear that independent actions are $\Theta(n)$ where n is the number of agents in the simulation. That is, the time required to apply the rule to a set of agents is bounded above and below by the number of agents n. We can deduce this

[4] *Growback, Seasonal Growback* and *Pollution Formation*

by observing that the loop iterates n times with each iteration consisting of a constant number of statements. The only assumption we make is that the $applyRule$ function takes constant time (i.e. it is $\Theta(1)$).

Concurrency

The **for**-loop can be implemented as a parallel map and belongs to the class of *embarrassingly parallel* problems. There is no communication required between agents as each agent acts alone. Given this it is clear that for P processors the parallel algorithm is $\Theta(\frac{n}{P})$ where $P \in \Theta(n)$. This is equivalent to $Theta(1)$ as P and n cancel out. In other words it scales linearly.

Deadlock and Livelock

As we have already stated in section 3.2, four conditions (Mutual Exclusion, Hold and Wait or Resource Holding, No Preemption and, Circular Wait) are required to hold before deadlock is possible. If any of these conditions do not hold then deadlock is impossible. A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing.

Independent actions by definition have the property that agents do not share any resources thus rendering both deadlock and livelock impossible.

Determinacy

Independent rules are deterministic, they always produce the same outcomes whether implemented synchronously or asynchronously.

6.8.2 Read-Dependent Actions

Rules where agents are required only to read the state of their neighbours[5] are all directly and simply expressible synchronously. For example, $PollutionDiffusion$, is an explicitly synchronous rule that determines how pollution levels diffuse over time. In effect each location absorbs a fraction of the pollution of its neighbours. Of the other such rules $Inheritance$ is implicitly synchronous and the remaining two rules are stated in an asynchronous manner but can also be given an equally plausible synchronous interpretation.

[5] $Culture$, $PollutionDiffusion$, $Inheritance$ and $DiseaseTransmission$

Applying a synchronous interpretation to these rules is a simple matter ensuring that state updates all occur simultaneously. This is enabled by separating the calculation of an agent's new state from the updating of the agent's old state.

Algorithm

The synchronous algorithm (Algorithm 2) contains two loops. The first loop computes what the value of the new state will be, while the second loop (lines 4-5) updates the agents state to the new state. This update occurs only after every agent has applied the action rule based on the *current* state of its neighbours. This ensures that the updates are applied simultaneously.

Agents do not have access to global state. They are only allowed access to the state of their *neighbours*. Each simulation will have some rule that defines the concept of neighbours. It is usually defined for an agent a as all agents within some set radius of a. We assume that the function $computeNeighbourhood$ (line 2 of Algorithm 2) takes in an agent a and returns the complete neighbourhood of a. The $applyRule$ function (line 3 of Algorithm 2) must take as input the neighbourhood of agent a to help it compute the new state of agent a.

Algorithm 2 Synchronous Read-Dependent Action

```
Input: $Agents : list\langle agent\rangle$
1: for all $a \in Agents$ do
2:     $neighbours \leftarrow computeNeighbourhood(a)$
3:     $a.newState \leftarrow applyRule(a, neighbours)$
4: for all $a \in Agents$ do
5:     $a.state \leftarrow a.newState$
```

Asymptotic Analysis

There are two loops in the algorithm. The second loop (lines 4–5) is obviously $\Theta(n)$, with exactly one iteration per agent. The first loop (lines 1–3) is $\Theta(n)$ only if we assume that the functions $computeNeighbour$ and $applyRule$ take constant time ($\Theta(1)$). Once we accept those assumptions then the sequence of two loops both of $\Theta(n)$ still gives an overall (upper and lower) bound of $\Theta(n)$. These assumptions are reasonable as each agent has a predefined neighbourhood size that is constant, completely independent of the size of the simulation space.

In Sugarscape both assumptions hold for all rules. In any case the running time bounds is the same for the synchronous and asynchronous algorithms. We can see this is the case by comparing Algorithm 2 and Algorithm 3. The second loop is the only extra work in the synchronous algorithm and this is always $\Theta(n)$. Therefore if our assumptions about $computeNeighbour$ and $applyRule$ being $\Theta(1)$ are correct we get the same result for both approaches and if our assumptions are incorrect then the work required by the synchronous algorithm will be dominated by the extra work of the first loop; in other words the work done by the second loop would be insignificant when compared to the work undertaken by the first loop.

The synchronous approach requires two copies of state for each agent. Each agent's state requires C bits of space, for some value C, so n agents require $2 \times Cn$ bits and, although this is twice the space requirements of the asynchronous approach, the space complexity in both cases reduces to $\Theta(n)$ as we ignore constants when dealing with space-time complexity. This space complexity limit also holds for both *Read-Dependent* and *Write-Dependent* actions.

Concurrency

In the case of the synchronous algorithm the separation of state update into its two components means that each of the two loops belongs to the class of *embarrassingly parallel* problems. Therefore the algorithm can be implemented as a sequence of two parallel maps with each concurrent loop having a bound of $\Theta(\frac{n}{P})$ where $P \in \Theta(n)$ or $\Theta(1)$

Algorithm 3 Asynchronous Read-Dependent Action

```
Input: $Agents : list\langle agent\rangle$
1: for all $a \in Agents$ do
2:     $neighbours \leftarrow computeNeighbourhood(a)$
3:     $a.state \leftarrow applyRule(neighbours)$
```

Because the asynchronous approach (Algorithm 3) does not separate out the update of the states there is more work required to parallelize that approach. Specifically we need to ensure that if one agent in a neighbourhood is updating then no other agent in that neighbourhood can update at the same time.

As a result, the asynchronous algorithm is not as easily parallelised as the synchronous algorithm and requires either locking, with all that entails, or a restructuring of the algorithm to employ e.g. geometric tiling.

Deadlock and Livelock

This algorithm ensures that no agent performs updates until all agents have finished accessing its state. The separation of the behaviour into two separate components, one reading agent state and the other writing to the new state, means that the reading of agent state can proceed without locking as can the updating of state. Without the need to perform any locking there is no Deadlock or Livelock.

The issues of Deadlock and Livelock in the asynchronous algorithm, however, remains contentious and some form of locking is required.

Determinacy

The synchronous algorithm guarantees determinacy but the asynchronous approach does not. This will make the asynchronous algorithm more difficult to test as each execution run can have a unique outcome dependent on the order that the agents were updated.

6.8.3 Write-Dependent Actions

The rules for which an asynchronous interpretation might appear, at first, to make more sense are those where agents form exclusive subgroups (usually but not necessarily exclusive pairings of agents) where the members of a group cause mutual updates of each others state[6]. In these cases the rules can still be defined synchronously. The solution is not to create more time intervals (as attempted in asynchronous updating) but to break these rules into smaller (atomic) components. These rules all have two components: firstly the formation of exclusive groupings in preparation for updating of agent state and, secondly, the execution of these updates within each subgroup. For example, the trade rule in Sugarscape requires that agents first decide on who they wish to form exclusive trading pairs with before they start trading. These two components must occur in sequence but each component can, in itself, be performed synchronously.

It is instructive to look at the movement rule from Sugarscape to compare its asynchronous and synchronous interpretations [Epstein and Axtell, 1996]:

Movement - M

- Look out as far as vision permits in each of the four lattice directions, north, south, east and west;

[6] $Credit$, $Combat$, $Trade$, $Mating$, $Replacement$ and $Movement$

- Considering only unoccupied lattice positions, find the nearest position producing maximum welfare;
- Move to the new position
- Collect all resources at that location

When an agent moves it changes location and consumes the resources at its destination. With multiple agents all moving simultaneously we must ensure that no two agents move to the same location, as each location can only host one agent at a time. The asynchronous approach handles this difficulty implicitly by ensuring that only one agent can make a move at a time. The random ordering of this sequence of move events acts as an implicit collision avoidance (or resolution) mechanism.

Our synchronous approach divides this rule into its two component parts: exclusive group formations and agent updates. In this case each exclusive group consists of a moving agent and its chosen destination location. Each agent can produce a proposed group on parallel. For example each agent can independently propose a destination that it wants to move to. However, some explicit collision resolution (avoidance) rule is required to determine the outcome when two or more agents attempt to chose the same destination. If two or more agents propose moving to the same destination then some way of deciding who succeeds must be applied. Under the asynchronous approach this is not an issue as we force each agent to move in sequence. We can mimic the asynchronous approach by flipping a coin to randomly determine the outcome of such collisions but we also have the opportunity to put in place a more logical resolution strategy such as "closest agent to the destination wins" and save the coin-flipping for tie breakers (such as when both agents are equidistant from the destination). This gives us three advantages over the asynchronous approach

1. We are not forced to introduce randomness into the simulation;
2. We make our resolution strategy explicit;
3. We can pick from a number of different conflict resolution strategies.

The second component of the synchronous $Movement$ behaviour is the state update phase. Because we have already removed all possible collisions these updates can all occur in parallel without any issues.

The asynchronous approach serialises agent actions and this acts as its implicit collision resolution mechanism. The outcome is dependent on the order of agent movement. Since the asynchronous approach uses random orderings of agent actions this is similar to employing an implicit *random choice* conflict resolution strategy.

The synchronous approach requires an explicitly stated resolution strategy and allows different strategies to be used. Unless we explicitly choose to use a random resolution strategy the synchronous outcome will be deterministic and is independent of the order of agent action execution.

Algorithm

The synchronous algorithm (Algorithm 4) divides *Write-Dependent* actions into three components:

1. Formation of exclusive groupings of agents (lines 1–17);

2. Application of action update within each group (lines 18–20);

3. Updating of Agent State (lines 21–23).

The formation of the exclusive groups is the more substantial part of the algorithm. Once the exclusive groups are formed the updates can proceed much the same as the previous action categories. When forming the exclusive groups we must do so in a manner that detects conflicts and resolves them. In certain cases it may be necessary to give agents more than one chance to form groups amongst themselves. We have assumed the most complex case where we continue to try form groups until every agent is either a part of a group or has no possible groups left (hence the loop condition at line 2). The simpler case where each agent gets only one shot at forming a group can be achieved by omitting the while loop (line 2) from the algorithm.

The function $computeNeighbours$ returns the complete neighbourhood of an agent, that is, all surrounding locations (empty or not) and all residing agents in those locations. If the action requires interaction between agents then it is only the other agents, if indeed there are any, within this neighbourhood that can take part in the following action. For the Movement behaviour the interaction is between an agent and an empty location so in this case it is only the empty locations within the neighbourhood that are relevant.

The function $formGroup$ takes in an agent a and the complete neighbourhood of a and returns a *proposed* group containing a and some of its neighbours. $formGroup$ is defined for each behaviour. For the $Movement$ behaviour $formGroup$

searches the neighbourhood for empty locations and picks the one with the highest Sugar resource as the proposed destination. For the $Trade$ behaviour it would look for the location containing an agent that it can trade most profitably with. If there is no available grouping then it returns a group containing only a itself (that is the agent has no new destination to move to, it stays put).

Each group has a weighting attached to it by $formGroup$ that signifies the *ranking* of that group. This ranking score is used to determine the outcome of conflict detection and resolution. If two or more proposed groups overlap (e.g. two or more agents have chosen the same destination) then only one of these groups can be accepted. The ranking of the groups is used to determine which group will be accepted. Ranking then implements collision resolution.

For example, a move to a closer destination may have a higher rank than a move to a destination further away so if two or more agents try to move to the same location then the agent closest to the destination wins. It is this ranking that is used to sort the groups. A conflict occurs whenever two or more groupings share agents in common.

Conflict resolution is handled by testing groups for exclusivity in their sorted (rank) order. A group G will only be deemed exclusive if there are no other selected groups with a higher ranking that contain agents within G. Because groups are checked in order of their rank a group can only be blocked if there was a group that overlapped with this group and was accepted before this group. Because groups are checked in rank order we can guarantee that any groups already checked have a rank at least as high as the current one. How the rank score is determined has to be explicitly stated by the rule that defines the action.

There are two key data structures in the algorithm: $AcceptedGroups$ and $ProposedGroups$. These are (respectively) the list of accepted exclusive groups and the list of proposed groups. We first generate the list of proposed groups, one per agent and then sort these in order. We then iterate through this sorted list picking out groups that do not overlap with any of our accepted groups and adding them to the list of accepted groups. Any groups that do overlap with accepted groups are thrown out. We can then, if necessary, allow the rejected agents to propose new groups until every agent has found an acceptable group or is unable to form any group.

The algorithm proceeds as follows. Every available agent proposes an exclusive group based on the set of neighbours still available (lines 4–7). These proposed groups (one for every available agent) are then sorted in order based on their associated rank (line 8). Then we iterate through the list of proposed groups in order of ranking (lines 9–17). If all agents in a proposed group are still available (lines 11-14) we add that group to the list of accepted groups (line 15) and remove the agents contained in that group from the list of available agents (lines 16–17). If any one agent in a proposed group is already allocated to another higher ranking group then this proposed group is rejected. This process continues until there are no agents remaining who are not in a group (**while**-loop on line 2).

The second part of the algorithm just iterates through the final list of accepted groups (lines 18–20) and calls $applyRule$ for each group. This function applies the rule to the group of agents and stores their new states in preparation for updating.

The third and final part of the algorithm goes through each agent and updates their state to the new state (lines 21–23).

Algorithm 4 Synchronous Write-Dependent Action

```
Input: $AvailableAgents : list\langle agent\rangle$
 1: $AcceptedGroups \leftarrow \varnothing$                         ▷ Part I: Form Groups
 2: while $AvailableAgents \neq \varnothing$ do
 3:     $ProposedGroups \leftarrow \varnothing$
 4:     for all $a \in AvailableAgents$ do
 5:         $neighbours \leftarrow computeNeighbourhood(a)$
 6:         $g \leftarrow formGroup(a, neighbours)$
 7:         $ProposedGroups.add(g)$
 8:     $ProposedGroups.sort()$            ▷ put proposed groups in ranked order
 9:     for all $g \in ProposedGroups$ do   ▷ pick acceptable groups based on rank
10:         $isExclusive \leftarrow true$
11:         for all $a \in g$ do
12:             if $a \notin AvailableAgents$ then
13:                 $isExclusive \leftarrow false$
14:         if $isExclusive = true$ then
15:             $AcceptedGroups.add(g)$                ▷ Accept proposed group
16:             for all $a \in g$ do
17:                 $AvailableAgents.remove(a)$
18: for all $group \in AcceptedGroups$ do      ▷ Part II: Update Agents in Groups
19:     for all $a \in group$ do
20:         $a.newState \leftarrow applyRule(a, group)$
21: for all $group \in AcceptedGroups$ do     ▷ Part III: Apply Updates to Agents
22:     for all $a \in group$ do
23:         $a.state \leftarrow a.newState$
```

The asynchronous algorithm (as shown in Algorithm 5) is simpler because it does not have an explicit conflict resolution rule. As a result this algorithm is nondeterministic. It is also possible, in the asynchronous algorithm, for an agent to get updated more than once because it can appear in more than one group (lines 5–6). This is not possible in the synchronous algorithm.

Algorithm 5 Asynchronous Write-Dependent Action

```
Input: $AvailableAgents : list\langle agent \rangle$
1: $AcceptedGroups \leftarrow \varnothing$
2: for all $a \in AvailableAgents$ do                    ▷ a is chosen randomly
3:     $neighbours \leftarrow computeNeighbourhood(a)$
4:     $group \leftarrow formGroup(a, neighbours)$
5:     for all $b \in group$ do
6:         $b.state \leftarrow applyRule(b, group)$
```

Asymptotic Analysis

The loop applying the group rule (lines 18–20 in Algorithm 4) is $\Theta(n)$ as long as the function $applyRule$ takes constant time. Given that there is a constant upper bound on group size (usually groups contain only two agents), determined by the maximum number of agents that can be in a neighbourhood, this is not an unreasonable assumption. Similarly the final loop applying the updates (lines 21–23) is, as always, $\Theta(n)$ as each iteration removes one agent and we iterate over all agents.

The first part of the algorithm is the most complex and will dominate the overall time taken. In this loop, the work occurs in three stages: the first inner loop (lines 4–7) populating the $ProposedGroups$ data structure; the sorting of the groups by rank (line 8) and; the second inner loop (lines 9-17) populating the $acceptedGroups$ data structure.

The first loop iterates through all available agents and each iteration performs a constant amount of work, under our assumption that $computeNeighbours$ and $formGroup$ are $\Theta(1)$, so this loop is $\Theta(n)$. The sort algorithm is $\Theta(n \log n)$[†]. The second inner loop iterates n times, as there is one proposed group for each agent. The loop within this loop (lines 11–13) will have a constant amount of work to do as each group will have C or less agents in it for some constant C where $1 \leq C \leq neighbourhoodsize$. This means that the second loop (lines 9–17) will be $\Theta(n)$ as long as each line of the loop runs in constant time. From this we can

[†]We assume *Quicksort* or *MergeSort* or something similar.

determine that the loop is dominated by the sort routine and *each iteration* of the **while**-loop is $\Theta(n \log n)$.

It is clear from this that the outer **while**-loop (lines 2–17) will dominate the running time of the overall algorithm. The upper and lower bounds will depend on the number of iterations of this loop with each iteration requiring $n \log n$ steps.

In the best case, every agent gets assigned to a group in a single iteration and we get a lower bound of $\Omega(n \log n)$. Computing an upper bound requires a little more work. Each iteration of the while loop is guaranteed to remove at least one agent - the highest ranking proposed group will always be chosen. If we assume that each iteration of the loop (lines 2–17) removes only *one* agent from the set of available agents then n iterations would be required giving an upper bound of $O(n^2 \log n)$.

This upper bound assumes that every proposed group is blocked by the first (highest ranking) proposed group but this assumption would violate the principle of locality in ABSS. In an ABSS each agent only has access to local information where *local* is defined by the neighbourhood size. As an agent can only interact with other agents within its neighbourhood it can only block agents within its neighbourhood from forming groups. Neighbourhood size is a constant, independent of the total number of agents n in the simulation. The number of neighbouring agents that an agent may have is also bounded by some constant value determined by the neighbourhood size. A single group preference can therefore only block a constant number of other groups, i.e. those groups containing agents in its neighbourhood. The maximum number of loop iterations is then equal to the maximum number of agents that can be contained within an agents sphere of influence which is a constant independent of n. Given this constraint, the upper bound now becomes $O(n \log n)$ matching the lower bound and giving us $\Theta(n \log n)$ for the algorithm.

The asynchronous algorithm (Algorithm 5) has two loops. The outer loop will iterate n times and the inner loop is bounded by the maximum group size (a constant independent of n). Therefore it will be $\Theta(n)$. This is a better result than $\Theta(n \log n)$ but $n \log n$ is considered an acceptable time complexity.

Concurrency

A naive concurrent implementation of this algorithm will still be dominated by the sort routine. The loops in Part II (lines 18–20) and III (lines21–23) and the choosing of preferred groups (lines 4–7) can all be implemented using simple parallel maps giving constant running time (for P processors we have a running time of

$\Theta(\frac{n}{P})$ where $P \in \Theta(n)$). The selection of groups from the preferred list (lines 9–17) must be run sequentially and so remains $\Theta(n)$. This is independent of the number of available processors. The sort routine (Line 8) will still remain close to $\Theta(n \log n)$ even with a concurrent implementation of the sort algorithm. Combining all these (which occur in sequence) we get:

$$\Theta(n + n \log n + \frac{n}{P} + \frac{n}{P}) \tag{6.23}$$

Since the $n \log n$ term outweighs all the other terms this is simplified to $\Theta(n \log n)$. This naive concurrent implementation will therefore give us a runtime equal to that of the sequential implementation independent of the number of processors. This is because the sorting of groups remains sequential and the concurrency is restricted to the updating of state and choosing of proposed groups. In other words the part of the algorithm that chooses acceptable groups from the list of proposed groups is still sequential. One solution would be to use a tiling algorithm. This is what we examine next.

Tiling for Multiprocessor Architectures

Tiling algorithms are useful if it is possible to divide a domain up into independent tiles or subsets that can then be processes concurrently. They are used on multiprocessor systems where tiles can be parcelled out to individual processors. Here we will first look at the possible speed ups when the tiles are independent. Then we will show that although adjacent tiles in ABM tend not to be independent it is possible to partition the tiles into sets of mutually exclusive subsets where each partition can be updated independently.

We note that in the analysis that follows N refers to lattice size while n refers to number of agents.

We can see if it is possible to use geometric tiling to attain weak scalability and a $\Theta(\frac{n}{P})$ concurrent algorithm (see Algorithm 6). For this to work we require that it is possible for each tile to be updated independently of any other tile. If the lattice can be divided into independent tiles then each independent tile can be processed concurrently. Assuming each tile contains $TileSize$ locations, a lattice with N locations will contain $\frac{N}{TileSize}$ tiles. Since each tile is a constant size processing a tile will take some constant amount of time, say $C_{TileSize}$. The total time complexity for processing the entire lattice is the number of tile times the amount of time taken by each tile:

$$C_{TileSize} \times \frac{N}{TileSize} \tag{6.24}$$

Because we have assumed that each tile can be processed independently they can also be processed concurrently without needing any locking. For P processors our time complexity is:

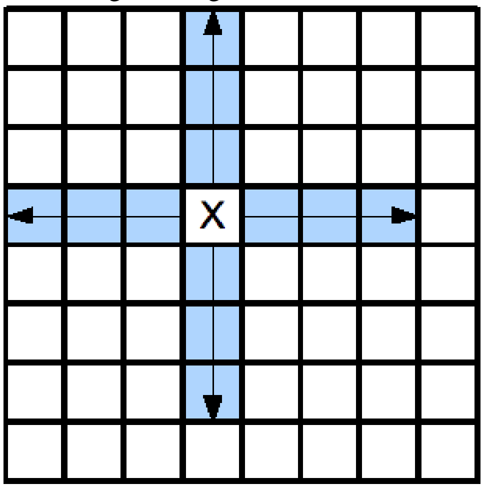
$$C_{TileSize} \times \frac{N}{TileSize} \times \frac{1}{P} where P \in O(N) \tag{6.25}$$

Ignoring constants gives an asymptotic complexity of:

$$\Theta(\frac{N}{P})\, where\, P \in O(N) \tag{6.26}$$

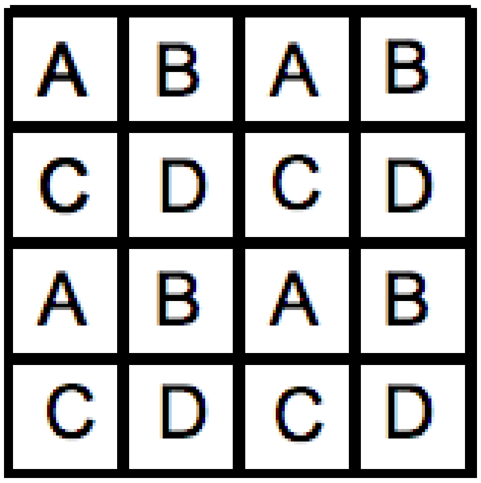
This result only holds under the assumption that the tiles are independent. This assumption is not true in our case because agents on the edge of one tile can interact with those agents on the adjoining tile who are in their locality. The amount of overlap that occurs between tiles is dependent on the size of an agents neighbourhood. In Sugarscape most behaviours only involve agents who are directly beside each other but some behaviours (e.g. $Move$ and $Combat$) use a neighbourhood size determined by the agents vision (see Figure 6.4).

Figure 6.4: Agent Neighbourhood with vision of 3



To make use of the idea of independent tiles we partition the lattice into sets of tiles that are not adjacent to each other (see Figure 6.5). Two tiles are nonadjacent if no agent in one tile has any neighbours in another. This can be achieved by ensuring that Tile size is greater than the maximum range of all agents. If this is the case then a tile interacts only with adjacent tiles to its North, South, East and West (its von Neumann neighbourhood). Given this we can see that this allows us to partition the tiles into four sets. In Figure 6.5 all tiles marked **A** are non overlapping and form one partition. similarly for tiles marked **B**, **C** and **D**. If agents use a Moore neighbourhood (agents can interact with its eight surrounding agents) then nine partitions are required.

Figure 6.5: Partition of tiles into four partitions (von Neumann Neighbourhood)



All the tiles in a partition, being non adjacent, can be processed in parallel but each partition must be processed in sequence. For example, in Figure 6.5 all tiles marked A can be processed concurrently as we are guaranteed they do not interact with each other but tiles in partition A cannot be processed concurrently with any tile from partition B because in that case there will be interactions between agents in the B tile and agents in its neighbouring A tiles. Each partition then takes:

$$C_{TileSize} \times \frac{N}{TileSize} \times \frac{1}{NumPartitions} \times \frac{1}{P}\, where\, P \in O(N) \tag{6.27}$$

The total amount of time to process all the partitions is the number of partitions times cost for one partition:

$$numPartitions \times C_{TileSize} \times \frac{N}{TileSize} \times \frac{1}{NumPartitions} \times \frac{1}{P}\, where\, P \in O(N) \tag{6.28}$$

which has an asymptotic time complexity of:

$$\Theta(\frac{N}{P})\, where\, P \in O(N) \tag{6.29}$$

This approach will work for the AU algorithm (Algorithm 7) and gives linear scalability. An identical approach to SU (Algorithm 6) does not work as well. The increased efficiency brought to bear by this tiled algorithm comes at a price. Each tile is updated synchronously but agents on the edge of a tile can interact with agents in their adjoining tile (that is, a tile in a different partition). Therefore in this case when that adjacent tile is processed these agents will have already been updated. In other words those specific agents will not be synchronously updated with reference to other agents in their tile. To retain a complete synchronous approach we must either use the original $\Theta(n \log n)$ algorithm or accept some asynchrony as a trade-off for increased efficiency.

Algorithm 6 Tiled Synchronous Write-Dependent Action

```
Input: $TileSet : List\langle Tile \rangle$
1: for all $tile \in TileSet$ do                              ▷ Outer for loop is parallel
2:     for $i \leftarrow 1 \ldots 2$ do                        ▷ Sequential for loop
3:         for $k \leftarrow 1 \ldots 2$ do                    ▷ Sequential for loop
4:             $AvailableAgents \leftarrow getAgentsInBlock(tile, i, k)$
5:             $ComputeSynchronousUpdate(AvailableAgents)$
6:             $Sync()$
```

Algorithm 7 Tiled Asynchronous Write-Dependent Action

```
Input: $TileSet : List\langle Tile \rangle$
1: for all $tile \in TileSet$ do                              ▷ Outer for loop is parallel
2:     for $i \leftarrow 1 \ldots 2$ do                        ▷ Sequential for loop
3:         for $k \leftarrow 1 \ldots 2$ do                    ▷ Sequential for loop
4:             $AvailableAgents \leftarrow getAgentsInBlock(tile, i, k)$
5:             $ComputeAsynchronousUpdate(AvailableAgents)$
6:             $Sync()$
```

Deadlock and Livelock

The tiling algorithm disallows more than one process from reading or writing the state of the same agent at the same time. As a result we require no locking and so both Deadlock and Livelock are impossible.

Determinacy

The algorithm is deterministic as long as a deterministic (or stable) sorting algorithm is used.

6.8.4 Synchronous DES

To demonstrate that the synchronous approach can be applied to DES we have provided a basic DES algorithm. The basic algorithm outlined (algorithm 8) assumes that the $EventQueue$ is never empty in order to simplify the code presented. It is very similar to the ABM based code but differs in that (i) rules are replaced by events and (ii) we cannot assume that an event is applied to every agent at each time step. Therefore we must first find all events that are scheduled to take place at a definite time, where this is defined as the time of the first event in the sorted $EventQueue$ (assuming the $EventQueue$ is sorted by event timestamp).

If an event cannot occur because it cannot find an acceptable group of neighbours that are available then we assume that it adds that event along with the empty set to the $ProposedEventGroups$ data structure. Depending on the simulation rules, either the event then does not occur or the event is reinserted back into the $EventQueue$ for some future time (now+'a time increment').

[Zaft and Zeigler, 2002] identified that an advantage of implementing an ABM as a DES is increased efficiency. When not every agent is required to apply a rule during every step then the amount of work required during each step is reduced.

6.8.5 Issues with Asymptotic Complexity

Asymptotic Complexity is a well known and understood method of measuring the time complexity of algorithms [Knuth, 1976] using complexity classes. While it is standard practice to use this technique when presenting an algorithm's time complexity there are some issues that need to be addressed.

1. Complexity classes ignore all constants when comparing algorithm complexity;
2. Asymptotic Complexity is architecture independent. It makes no assumptions about the type of processor architecture the algorithm will execute on.

Constants are ignored by asymptotic complexity on the basis that any constant speed difference between two times will eventually be overcome by the increasing

Algorithm 8 Discrete Event Simulation Algorithm

```
Input: $EventQueue : SortedQueue\langle event\rangle$
Input: $AcceptedEventGroups, proposedEventGroups$ :
    $List\langle(event, group)\rangle$
 1: $AcceptedEventGroups \leftarrow \varnothing$
 2: $currentEvents \leftarrow \varnothing$
 3: $nextEvent \leftarrow EventQueue.peek()$
 4: $currentTime = nextEvent.time$                    ▷ Get all events that occur now
 5: while $currentTime = nextEvent.time$ do
 6:     $currentEvents.add(nextEvent)$
 7:     $EventQueue.pop()$
 8:     $nextEvent = EventQueue.peek()$
                                                  ▷ Process all events that occur now
 9: while $currentEvents \neq \varnothing$ do
10:     $ProposedEventGroups \leftarrow \varnothing$
11:     for all $e \in currentEvents$ do
12:         $neighbours \leftarrow computeEventNeighbours(e)$
13:         $g \leftarrow formGroup(e, neighbours)$
14:         $ProposedEventGroups.add((e, g))$
15:     $ProposedEventGroups.sort()$
16:     for all $(e, g) \in ProposedEventGroups$ do                ▷ in ranked order
17:         $isExclusive \leftarrow true$
18:         for all $a \in g$ do
19:             if $a.available = true$ then
20:                 $isExclusive \leftarrow false$
21:         if $isExclusive = true$ then
22:             $AcceptedEventGroups.add((e, g))$
23:             for all $a \in g$ do
24:                 $a.available \leftarrow false$
25: for all $group \in AcceptedEventGroups$ do          ▷ Part II: Apply Events to
    Groups
26:     for all $(e, g) \in group$ do
27:         $applyEventRule(e, group)$
```

speed of processors under Moore's Law. This has proven to be the case up until now as processor speeds have followed an exponential increase in speed.

It is possible that a constant can cause an algorithm in one complexity class to operate more slowly than one in a worse complexity class for a range of input sizes. Eventually as the problem size increases the effect of the constant factor will become irrelevant in the overall analysis but in some cases the typical problem size is such that the constant factor is important. Table 6.1 shows one such case where the linear algorithm is beaten by the N^2 algorithm for data sizes up to 10,000 due to the presence of a large constant. If we are dealing with an algorithm that deals with data sizes within some specified range then constants can make a difference.

An area where this could affect our algorithm is the tiled version of the algorithm where there is an overhead associated with communication between adjacent tiles. Given that each tile is a constant size we can deduce that this overhead can be defined by some constant factor. As a constant it does not appear in the final asymptotic complexity but it may be the case that this overhead is large enough that it will place some lower bound on the size of simulation that it is profitable to use the tiling algorithm on.

Table 6.1: **Comparing Complexity Classes**

Data Size	Complexity:$10,000N$	Complexity: N^2
1	10,000	1
10	100,000	100
100	1,000,000	10,000
1,000	10,000,000	1,000,000
10,000	100,000,000	100,000,000
100,000	1,000,000,000	10,000,000,000

The second issue is processor architecture independence. This makes sense as a time complexity based on one particular architecture would be quickly out of date and the analysis would need to be repeated for every other available architecture. It does mean that certain architectural constraints are ignored that can have a large effect on running time. For example, no account is taken of cache memory yet coding within cache memory contraints is an important source of program speed.

We must also recognise that processor architectures are currently in a state of flux [Blake et al., 2009] as we move from a sequential processor model to a parallel one. Many different architectures are under development ranging from MIMD based multicore to many-core CPUs [Sato et al., 2014, J. Held and Koehl, 2006, Borkar et al., 2006] alongside GPU based SIMD [Nvidia, 2011] architectures. Even within each architectural approach updates in memory bus technologies and cache size is ongoing. AMD's APC architecture allows GPU and CPU to share the same memory thereby removing the expensive overhead of transferring data from CPU to GPU, thereby changing the dynamic of GPU programming. Until such a time as the switch is made to standard architectures there is little benefit in basing an algorithm on just one possible architecture. That being said once an algorithm is developed then different versions of the algorithm can be derived that target specific architectures if required.

In order to address some of these issues we have produced some benchmarks in appendix A that show these algorithms performance on multicore architectures. The tailoring of these algorithms to other specific processor architectures such as GPU or distributed processors has been left as further work to be explored in the future as it is tangential to the main thrust of this thesis, that of using formal methods alongside SU and AU to get reproducibility in ABM/ABSS.

6.9 Conclusion

While AU and SU implementations of ABMs can produce different outcomes it is still an open question as to which is more realistic. In the CA literature it is not unusual to see the same CA implemented using AU and SU, and comparisons of the results made. The lack of SU algorithms for ABM has made this impractical until now. More work is required on understanding how the choice of updating algorithm can affect simulation outcomes. By allowing side by side comparisons to be made between the two approaches we can help identify and explain the circumstances under which they differ.

An analysis of the differences between AU and SU showed that SU provides a consistent flow of information in a simulation. SU also allows for the introduction of conflict resolution rules within a simulation thereby giving more precise control over agent interactions. Analysis suggests that:

1. AU, by interfering with the speed of transmission of information across the simulation space, will produce less periodicity in the system than SU;

2. High density populations within systems are more likely to enhance the differences in outcomes between synchronous and asynchronous implementa-

tions of the system.

Both of these are borne out in the literature [Caron-Lormier et al., 2008, May, 1973]. SU increases determinism and clarity while allowing different conflict resolution rules to be applied.

Novel SU algorithms were presented that allow SU to be extentded to ABM based simulations. Agent interactions were categorised into three different types based on their complexity: *Independent*, *Read-Dependent* and *Write-Dependent*. We showed that all types of action can be expressed synchronously by splitting these actions, where necessary, into their component parts. It was demonstrated, using asymptotic analysis, that the theoretical running time of the synchronous algorithm is roughly comparable to the asynchronous algorithm in the sequential case and acceptable in the concurrent case. Both SU and AU algorithms offer good *weak scalability*. The SU algorithms are *Deadlock* and *Livelock* free and offer deterministic execution. This rules out entire classes of error when developing code (for example, *Data Races* and *Deadlocks*) and ensures, due to the deterministic nature of the code, repeatability thus allowing easier debugging. These algorithms can handle more complex types of interaction than any of the alternative SU algorithms.

These new SU algorithms mean that it is now possible for comparisons to be made between SU and AU implementations of simulations. These comparisons make it possible to distinguish between outcomes that are inherent to the system being modelled and those that are just artefacts of the updating strategy. We have implemented a large ABSS (Sugarscape) using both AU and SU algorithms and compared the results. This is the only SU implementation of Sugarscape that we are aware of and the most complex ABM implemented with SU.

Chapter 7

Sugarscape: A Case Study in Replication

7.1 Introduction

We have derived synchronous algorithms that implement each category of agent interaction. In this chapter we demonstrate our framework that implements these algorithms alongside AU algorithms thus allowing side by side comparisons of different updating strategies. Based on the formal specification of Sugarscape (Chapter 5) and the novel SU algorithms from Chapter 6 this framework reproduced the original results of Sugarscape under a variety of different updating strategies. This shows in general terms:

1. Formally Specifying a simulation is a viable approach to replicating results;
2. The novel synchronous updating algorithms derived in this thesis are capable of handling the full range of complex interactions found in ABM and ABSS;
3. Application of different updating strategies can demonstrate the robustness (or otherwise) of ABM results;

More specifically with regard to the original Sugarscape results it shows:

1. The results claimed by Sugarscape are reproducible and robust with regards to the updating strategy employed. In other words the simulation properties claimed are not just artefact's of the AU strategy employed;

2. Where the obtained results differed from those in *Growing Artificial Societies* this approach allowed for the identification of the reasons for this. This is due to the precise nature of the formal specification of Sugarscape and the confidence that comes from side by side comparison of AU and SU implementation strategies.

First we give a brief overview of the framework. Then we use the framework to compare and contrast the outcomes of using SU and AU on Sugarscape. Specifically we examine its effect on the emergent properties of Sugarscape originally identified in [Epstein and Axtell, 1996]. We show that it is possible to reproduce many of the results originally obtained and check to see how the replacement of AU with SU affects these results. Where the results are not reproducible the issues causing this are clearly identified.

7.2 The Framework

The framework is a proof of concept implementation of the SU algorithms that also implements AU algorithms. This framework is designed in a manner that allows AU and SU strategies to be attached to a single implementation of any rule thereby guaranteeing that the only difference between the implementations is the updating strategy employed. It was developed in C++11 on OS X and Linux. As the framework primarily serves as a reference implementation the code has been kept as simple as possible. Whenever a choice existed between efficiency and clarity, clarity was chosen. The framework is documented using the *DOxygen* documentation tool [van Heesch, 2015]. Full source code and documentation is licensed under the GPLv3 and is available online [Kehoe, 2015b].

In general when using the framework to implement any particular behaviour or agent interaction all that is required is that we do two things. First we must determine what category (Independent, Read-dependent or Write-dependent) the interaction or behaviour belongs to. Then we create a class for the behaviour that inherits from the appropriate class. These three classes, each representing a different type of agent interaction, are defined in chapter 6.

Independent Any action that involves no interaction between agents. *Growback* is one such action where each location simply increases its sugar reserve without reference to any other location.

ReadDependent An action where the agent involved updates only its own state but reads the state of other agents to determine how to perform this update. For example, $PollutionDiffusion$ causes a location to updates its own state based on the state of surrounding locations.

WriteDependent An action that causes an agent to update its own state and the state of other agents as well. For example, $Movement$ updates the agent making the move as well as its starting location (changes from containing an agent to being empty as the agent moves from here to its new location) and the destination location (changes from being empty to containing this agent as the agent moves to this location).

Once the class representing the behaviour is created two functions must be implemented that carry out this behaviour:

formGroup This function returns the set of agents that the current agent wants to interact with. The location containing the agent performing the behaviour is passed in as a parameter to this function. It returns the group of agents that this agent requires access to (for read and write purposes). For example, to implement the *Movement* behaviour an agent will form a group consisting of their current location and their preferred destination. If necessary a group can also be assigned a ranking that will be used for determining priority. Ranking is used to implement collision resolution. We will rank a move according to how close the destination location is to the moving agent's current location. The framework will use this ranking to determine what happens if two or more agents try to move to the same location. In the case of $Movement$ we could rank the agent based on how close it is to the destination. An $IndependentAction$ contains no agent interaction so in this case $formGroup$ is not required. The formGroup method, in the case of *Movement*, contains the code that identifies the chosen destination for this agent to move to.

executeAction This function applies the rule to everyone in the group returned by the *formGroup* function. If we again take the *Movement* behaviour then the rule application simply changes the location of the agent in the group from its current location to the destination location.

The code in these two methods completely implements the agent behaviour independently of updating strategy. Once these functions are implemented a specific updating strategy for a behaviour can be chosen (AU or SU). All other issues, such as contention or concurrency, are handled transparently by the framework. Where a rule contains some random element (such as randomly choosing between different locations when deciding where to move to) the *Mersenne Twister* [Matsumoto and Nishimura, 1998] Pseudo Random Number Generator is employed.

The framework implements a number of different AU strategies alongside the new SU algorithms. Specifically three different AU strategies are implemented:

Fixed Direction Line-By-Line The locations in the lattice representing the simulation space are updated in the order they appear in the lattice (left to right, top-down);

Fixed Random Sweep The order that is used is determined randomly at the start of the simulation and this order is used for every step in the simulation;

Random New Sweep The order that the agents are updated is determined randomly at the start of each step (each step uses a different random order).

This allows the framework to be used to produce side by side comparisons between the contrasting synchronous and asynchronous updating strategies. Something that is not currently undertaken within ABSS.

7.2.1 Using the Framework

We will now look at how to implement the $Movement$ behaviour. This is a $WriteDependent$ behaviour so we create a class that inherits from WriteAction. The class definition is shown in Listing 7.1.

```
class AgentBasicMove :public WriteAction{
public:
    AgentBasicMove(World *s, Strategy *theStrategy);
    virtual bool executeAction(Location *, group *);
    virtual group* formGroup(Location *);

};
```

Listing 7.1: Class Definition for Movement (AgentBasicMove.h)

Taking each function in turn:

The constructor does not need to do anything as the base class (WriteAction) constructor does all necessary initialisation. The constructor must be passed a pointer to the *world* object and a pointer to a *strategy* object that implements the chosen updating strategy;

The *formGroup* function creates a group consisting of the agent making the move and the chosen destination location. In itself this is a simple function to write. The location of the agent undertaking this behaviour is passed in as a function parameter. All behaviours contain a pointer to the world object inherited from their base class. This contains an extensive API to allow us to collect any information we require about the simulation and in particular the agent neighbourhood. The code documentation contains full details on this API. Here we just ask for all empty locations within our vision. Pick one at random and add it to the group. Rank this group by distance from agent to destination (nearer is better) and return this group. If there is no suitable destination currently available then we return a group containing our current location as destination.

In the *executeAction* function the chosen group (as produced by the *formGroup* function) is passed in as a parameter. As before we do a simple check to ensure there is an agent present at this location first. All we have to do if there is an agent present is move the agent to this new location and update sugar levels.

```
AgentBasicMove::AgentBasicMove(World *s, Strategy *theStrategy)
    :WriteAction(s,theStrategy)
{
    //our work is done
}

group* AgentBasicMove::formGroup(Location *loc)
{
    group *ourChoice = nullptr;
    if (loc->hasAgent()) {/*!< Agent at this location */
        ourChoice = new group();
        /*!< Get agent performing action */
        Agent* theAgent=loc->getAgent();
        /*!< find all empty locations in locality*/
        std::vector<Location*> possibleDestinations=sim->
    getEmptyNeighbourhood(theAgent->getPosition(), theAgent->
    getVision());
        if (possibleDestinations.size()!=0) {
            /*!< There are possible destinations */
            int index=pickIndex(possibleDestinations);
            ourChoice->push_back(possibleDestinations[index]);
            ourChoice->setRank(sim->getRnd(0,10));
            ourChoice->setPrimeMover(loc);
```

```
                ourChoice->setActiveParticipants(1);
                //one active participant per group - the agent moving
            }
            else{/*!< nowhere to move so stay here */
                ourChoice->push_back(loc);
                ourChoice->setRank(0);
                ourChoice->setPrimeMover(loc);
                ourChoice->setActiveParticipants(1);
            }

    }
    return ourChoice; /*!< is NOT nullPtr only if we assigned it a
     value earlier */
}

bool AgentBasicMove::executeAction(Location *loc, group * grp)
{
    if (loc->hasAgent()) {
        Agent* theAgent=loc->getAgent();
        auto currPosition=theAgent->getPosition();
        auto newPosition=grp->getMembers()[0]->getPosition();
        /*!< remove old location ptr to agent */
        sim->setAgent(currPosition, nullptr);
        /*!< set new position to new location */
        theAgent->setPosition(newPosition);
        /*!< add ptr to agent at new location */
        sim->setAgent(newPosition,theAgent);
        /*!< eat sugar at new location */
        theAgent->incSugar(grp->getMembers()[0]->getSugar());
        /*!< sugar at new location now consumed */
        grp->getMembers()[0]->setSugar(0);
        return true;
    }else{/*!<We should never enter here */
        std::cerr << "executed Move with no agent! " << std::endl;
        return false;/*!< no agent present so do nothing */
    }
}
```

Listing 7.2: AgentBasicMove.cpp

Once we have created our behaviours applying them to a simulation is easy. We first create a world of some size (say a 50×50 lattice). We create one action object for each behaviour we want to run in the simulation and add these rules to the world. All behaviours use synchronous updating by default but this can be changed to any other updating strategy.

We do not have to worry about scheduling, concurrency or clashes with other agents. Our algorithm handles all of that transparently.

```
        //Initialise World 50 by 50 in size
        World theWorld(50);
        theWorld.init();
        theWorld.sync();

        /*!< Declare possible asynchronous updating strategies
    here   */
        NewSweepStrategy newSweep(theWorld);
        LineByLineStrategy lineByLine(theWorld);
        RndAsyncStrategy rndAsync(theWorld);

        /*!< create Movement behaviour object */
        AgentBasicMove move(&theWorld,&writeDependent);

        /*!< Add the Movement rule  to the world        */
        theWorld.addRule(&move);

        /*!< Change the updating strategy if required  - SU is
    default    */
        move.setStrategy(&lineByLine);
        /*!< run the simulation */
        for (int k=0; k<100; ++k) {//do 100 steps
            theWorld.applyRules();
        }
```

Listing 7.3: main.cpp

7.2.2 Framework Overview

There are six main uses for the framework:

1. It can be used by other researchers who wish to reproduce any of the results presented here;

2. It can be used by researchers who wish to run their own benchmarks;

3. It can be used by researchers to compare the effects of AU and SU in ABM and ABSS;

4. It can be used by modellers who wish to implement complex agent behaviours using the SU approach - a facility not currently available in other ABM toolkits;

5. It can serve as a starting point for anyone wishing to develop a more complete SU framework. For example to incorporate SU into an existing simulation toolkit;

6. The code can be browsed to see how our algorithms are translated into working code and how practical issues have impacted on the resulting implementation.

At present we have developed the single resource version of the Sugarscape simulation. The single resource version employs 12 of the 13 rules. The final rule *Trade* requires a second resource type, known as *Spice*, so that *Sugar* can be traded against *Spice*. Although this means that the *Trade* rule is not implemented, the 12 implemented rules cover the full range of interaction types including 8 rules with a complexity equivalent to *Trade* (e.g. *Combat* and *Culture*).

The framework consists of two applications. The first contains a simple Viewport GUI developed with the freely available and open source SFML (*Simple and Fast Multimedia Library*) [Gomila, 2015] . The viewport GUI is used for viewing agent behaviour in real time on the screen. The *viewport* provided is very simple and was used when implementing the framework to ensure that the various Sugarscape behaviours made sense, that is, to help judge their correctness. One example of its use would be to look for characteristic waves of agent migration between sugar peaks on the lattice as described in *Growing Artificial Societies* [Epstein and Axtell, 1996]. The second application contains no GUI. This was used when we required data logging of specific simulation attributes over time as in, for example, measuring the population count of the simulation over time.

The original Sugarscape implementation assumed an asynchronous updating strategy, specifically it employed the *Random New Sweep* AU strategy. The framework presented here covers SU as well as three different AU strategies.

It has already been demonstrated that synchronous and asynchronous updating give different outcomes [Radax and Rengs, 2010], [Caron-Lormier et al., 2008] in CA-based simulations. This framework allows allow this work to be extended to see how different updating strategies affect more complex ABSS.

There is no general agreement on what the original parameters of Sugarscape were and, as we have shown with our formal specification of Sugarscape (Chapter

5 and Appendix C), there are many ambiguities. This makes it impossible to show that any particular interpretation of the specification matches the original and thus makes comparisons between different implementations difficult. That being said, now that we have a formal specification of Sugarscape, this can be used as a reference for checking the effects of different updating strategies on the simulation outcomes. Where any particular result is not reproducible it is possible, based on the formal specification, to pinpoint the ambiguities that make reproduction of this result impossible.

7.3 Results from reproducing Sugarscape

Due to the aforementioned ambiguities in the original Sugarscape definition discovered in the formal specification process it is not possible to precisely replicate all the simulation parameters. We have made our definitions match the originals as closely as possible. Where the detail is precise it is replicated precisely and where there is ambiguity a best guess was made and documented of the original authors intentions. However, even given these ambiguities we can still see if we can replicate the general emergent properties and trends in the simulation[1].

Because the same framework is employed to produce both the AU and SU outcomes each simulation differs only in the updating method used - all other aspects are identical. In all cases the initial setup of the lattice was fixed so as to match the original Axtell and Epstein work as closely as possible.

Given the previous work on the differences between updating strategies in CA differences in the simulation outcomes were expected. What was not known, since this analysis had not been performed before on ABSS is how big or important these differences will be. The comparisons shown here provide new insights into this.

7.3.1 Carrying Capacity

One of the first emergent properties identified and measured was the *carrying capacity* of the lattice. That is, the number of agents that a fixed size lattice can support. The original graph produced by Epstein and Axtell shows the carrying capacity graph [Epstein and Axtell, 1996] has a distinctive shape. Population size initially steeply declines but very quickly levels off to a constant population size that represents the carrying capacity.

[1]That is, trends in population growth, spread of culture over time, etc.

The simulation used to attain the carrying capacity employs three rules: *Metabolism*, *Growback* and *Movement*. The first two of these rules *Metabolism* and *Growback* belong to the class of *Independent* actions and so will not be affected by the updating strategy (i.e. they produce the same outcomes under all updating strategies). The final rule, *Movement*, belongs to the *Write-Dependent* category and so will be affected by the strategy employed. Therefore any differences between the SU and AU versions can only be due to the *Movement* rule.

To reproduce the results of the original simulation we measured the carrying capacity of Sugarscape using SU over time. The specific *carrying capacity* of Sugarscape depends on many factors including:

Lattice Size Larger lattice sizes (all other factors being equal) will support more agents. A lattice size of 50×50 is used throughout the original tests so we replicate this in all our tests;

Maximum Sugar Capacity and Distribution Each location has a maximum amount of sugar that it can hold. The distribution of sugar capacities will decide whether the sugar is evenly distributed throughout the lattice or concentrated into smaller parts of the overall lattice. If the locations with the ability to hold higher levels of sugar are concentrated into smaller areas this will force agents to crowd together in those areas thereby increasing agent interaction and competition. The original simulation setup used an uneven distribution of sugar carrying capacities with higher capacity locations centred on the Northeast and Southwest quadrants of the lattice. Unless stated otherwise we used a similar distribution that matches the original as closely as possible given the information available. The exact configuration of this distribution is shown in Figuer 7.2 has been made openly available alongside our code;

Growback Rate The rate at which sugar grows back at each location (up to the maximum amount a location can hold) has a large affect on the carrying capacity. A standard grow back rate of one sugar unit per time step has been applied as per the original setup;

Agent Metabolism This determines how much sugar an agent must consume during each turn in order to survive. The higher an agents metabolism the more sugar it needs to consume in order to survive;

Agent Vision Vision determines how far in each direction an agent can see and also move. An agent with a vision of seven, for example, can move up

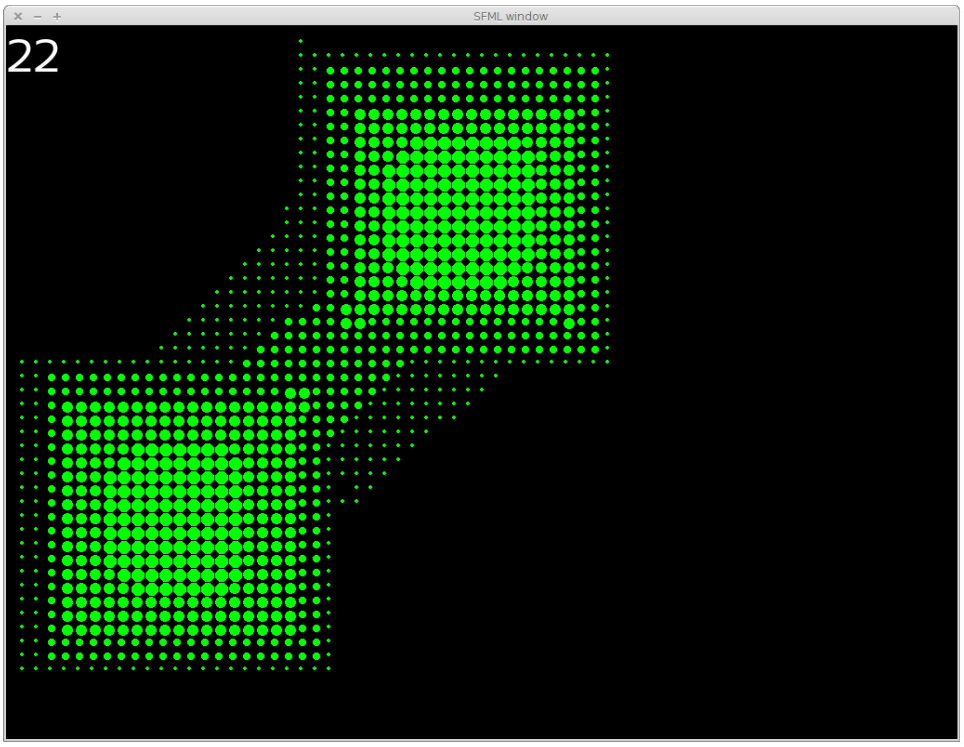


Figure 7.1: Food Distribution (Green indicates amount of Sugar at a Location)

to seven locations in a single move. The greater the vision the greater the chance of an agent reaching an adequate food supply and surviving longer.

In all cases the resulting graph (see Figure 7.2) of lattice carrying capacity replicated the characteristic shape obtained by Epstein and Axtell. Figure 7.2 shows the carrying capacity over time for a 50×50 lattice. In this case individual location maximum sugar carrying capacities ranging from 0 to 3 and were distributed evenly throughout the lattice. The initial agent population size was 600 and all agents were randomly assigned metabolisms ranging from 1 to 3 sugar units per step and a vision of 1.

This indicates that the original outcome, as reported by Epstein and Axtell, is not just an artefact of the updating strategy employed in the simulation. We have in fact shown a stronger result than just replication of previous results: the simulation is shown to reach a stable population size in the same general manner independent of updating strategy employed. This is a successful example of reproducing results as defined by [Drummond, 2009].

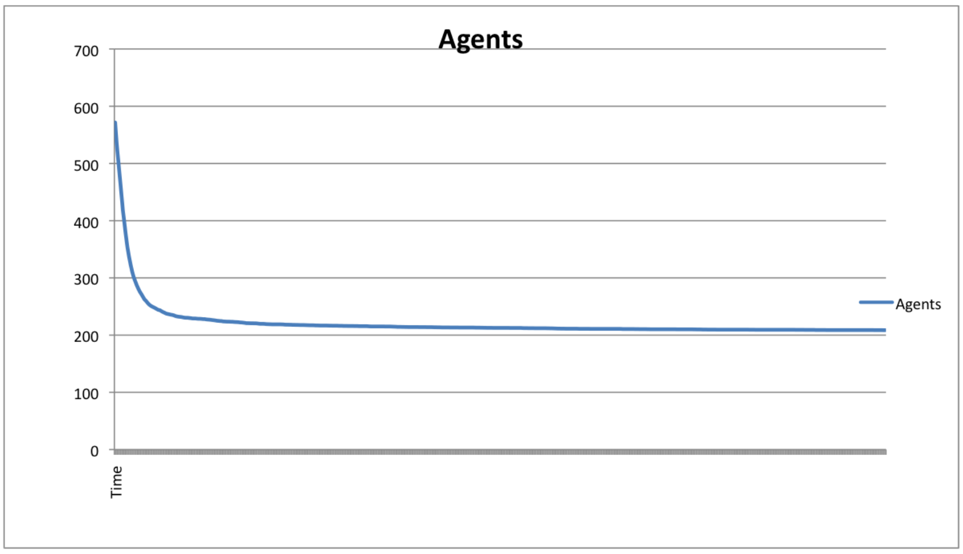


Figure 7.2: Lattice Carrying Capacity with Synchronous Updating (Agent Population versus Time

Table 7.1: **Carrying Capacity**

Strategy	Minimum	Maximum	Average
Synchronous	127	145	137
Line By Line	150	169	161
Random New Sweep	167	199	181
Fixed Random	172	198	190

To see if there is any difference in the carrying capacity when different updating strategies are used we compared the carrying capacities under each updating strategy while keeping all other aspects of the simulation identical. The simulation setup again closely matched the original setup used by Epstein and Axtell. The initial setup was:

- 50×50 lattice size;
- Max Sugar levels ranging from 0 to 3 sugar units per location with higher maximum levels clustered in the northeast and southwest quadrants;
- Agent metabolisms for each agent were individually randomly assigned values of between 1 and 3 units of sugar;
- Agent vision was fixed at 1;
- Growback of sugar resources at eah location was fixed at one unit per time step.

When we compare the carrying capacity of the simulation under the different updating strategies (Figure 7.3) we can see that although they all have the same distinctive shape there are some differences between them. The original sugarscape updating strategy was *Random New Sweep*, the most commonly employed updating strategy in ABM and ABSS.

There are two points of interest:

1. The carrying capacity of the simulation under SU is noticeably lower than it is under any AU strategy. It is not clear why this is the case but it can only be due to some manifest differences between movement under SU;
2. As we can see from Table 7.1, there is less variation in carrying capacity under SU. This is perhaps less surprising as synchronous updating is more deterministic that asynchronous updating. Given the same initial state the outcome of applying AU will depend on the random order applied to the agents during each step. Under SU this randomness is not present so we would expect less variation during each step.

A question that arises is whether the collision detection and resolution mechanism employed by the synchronous algorithm is responsible for these differences. The synchronous *Move* rule resolves collisions that occur between agents trying to

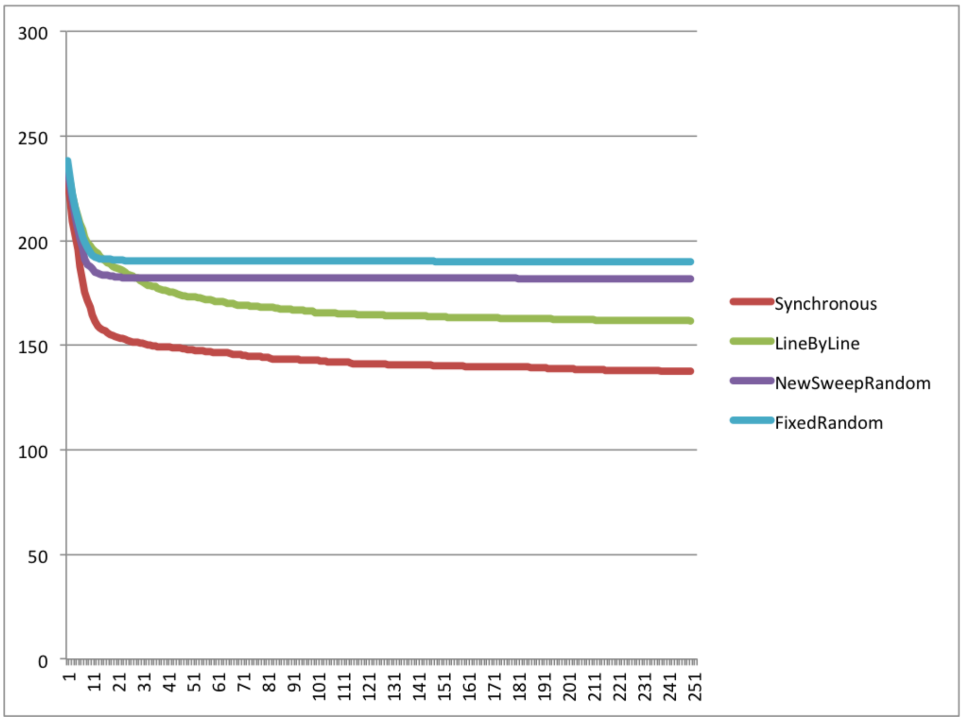


Figure 7.3: Lattice Carrying Capacity : Updating Strategies Compared

move to the same destination by favouring the agent closest to the destination. This is more deterministic than the asynchronous approach where the winner is chosen randomly.

To test this hypothesis that the collision resolution mechanism is responsible for the differences in outcomes, and since we have formally specified two collision resolution mechanisms for $Movement$, we compared two versions of the synchronous version of *Move*. The first version applies the standard resolution mechanism as already discussed while the second determines the winners of conflicts at random. Each version was run with identical starting parameters so that the only differences between them is the presence or absence of the collision resolution mechanism. For these simulations vision was set to seven for all agents so as to ensure overlap between agents occurs when choosing destinations to move to. Increasing the range of vision ensures that there is more possibility of agents attempting to move to the same destination and the collision resolution strategy will come into play more often. Each version was run 300 times and the resulting carrying capacities checked for any differences.

Table 7.2: **Carrying Capacity with different synchronous Movement Rules**

Strategy	Minimum	Maximum	Average
Closest Wins	210	267	210
Random Choice	211	272	211

As we can see from Table 7.2 there is no discernible differences between either version of the rule. Our hypothesis that the collision resolution aspect of the rule causes these differences turns out to be false. In this case, the differences between asynchronous and synchronous updating appears to be intrinsic to the nature of the two approaches and not affected by collision detection.

7.3.2 Effect of Metabolism and Vision on Carrying Capacity

It was originally demonstrated in *Growing Artificial Societies* that the lattice carrying capacity is affected by both agent metabolism and agent vision. The higher an agent's metabolism, the more sugar it needs to collect from the lattice in order to stay alive. If the available sugar in an agent's surrounding locations is less than it needs then it will die. Similarly, the better an agent's vision is, the more locations it can reach in a single move and the more likely it is to be able to find a location with

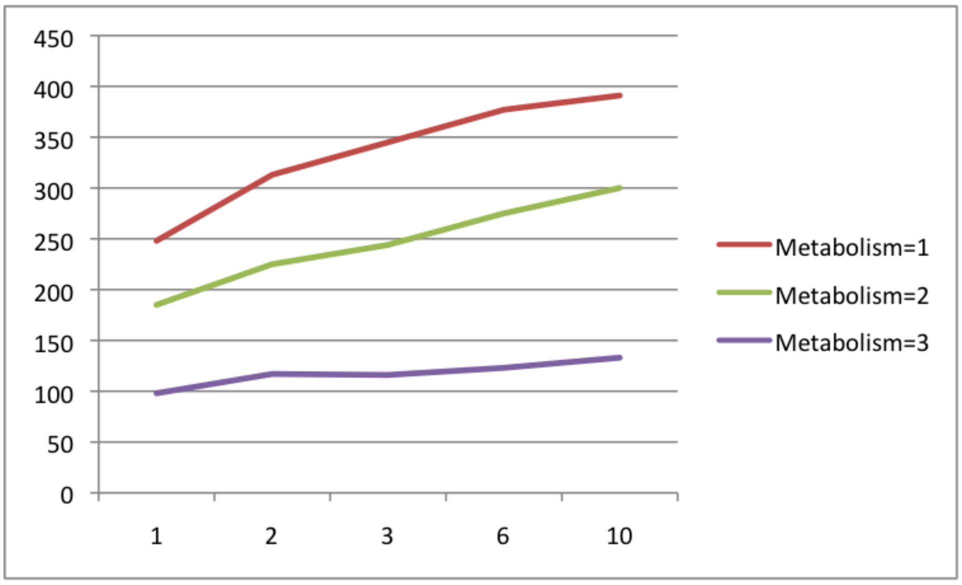


Figure 7.4: Carrying Capacity with varying Metabolism and Vision

enough sugar to cover its needs. Agents with good vision and low metabolism are more likely to survive on the lattice. Carrying capacity, by this reasoning, should be directly proportional to the average vision in the agent population and inversely proportional to the average metabolism of the agent population. This is what was reported in the original simulation study.

We reproduced these results using SU. The initial setup is as before but with vision and metabolism fixed at a constant value throughout the population, that is each agent is assigned the same vision and metabolism. This allowed us to see how carrying capacity was affected by different values for vision and metabolism. Figure 7.4 shows the effects of metabolism and vision on carrying capacity. This matches the original findings. To produce this graph, the carrying capacity of the simulation was calculated when all agents were assigned fixed metabolism and vision values. The simulation was tested with agent metabolisms fixed between one and three and vision fixed with values from one to eleven. It shows that the carrying capacity is inversely proportional to metabolism. It also demonstrates that for a population with a specific metabolism increasing the vision of all the agents in the population increases the carrying capacity of the lattice. The larger differences in carrying capacity when the agent population has a metabolism of three is due to the fact that the maximum sugar carrying capacity of any location is never more than three (these high capacity locations are centred on the Northeast and Southwest of the lattice). Therefore only agents reaching these high capacity locations can survive. Any agents straying away from these sugar "peaks" will use more sugar than they can harvest during each step and so will die.

The findings again show the robust nature of the general trends and properties in Sugarscape in that these are not artefacts of the original AU strategy employed but hold under different AU and SU strategies.

7.3.3 Comparing effects of SU and AU on Carrying Capacity

The differences in the specific values we have obtained for carrying capacity and the original values can be accounted for in differences between the original lattice set-up and the set-up employed here. Differences that are due to ambiguities in the original simulation definition. To see what specific differences are exist due to updating strategies we repeated the simulation with different updating strategies.

Figures 7.5 and 7.6 show that there is little discernible difference between AU and SU when metabolism is fixed at one or two sugar units per move. That represents the situation where there is a "plentiful" supply of sugar on the lattice. Figure 7.7, on the other hand, shows larger differences between AU and SU when the metabolism for agents is fixed at three units per move.

This marked difference in carrying capacity when metabolism is set to three can be explained. The lattice upon which the agents roam is set up so that each location can contain a maximum of three sugar units while every agent requires three sugar units to metabolise during each step. This is a world of scarcity. An agent moving in the world can, at best, consume just enough sugar units to cover the cost of its last move. In the best case, by moving to a location with three sugar units, an agent can only cover its costs. That is, it harvests three sugar units from the lattice but consumes exactly the same amount in making the move giving each agent no gain in sugar reserves. In fact no agent can never build up any sugar reserves. Every time it moves to a location with less than three sugar units any sugar reserves it already has will be depleted. The only locations that have the maximum capacity of three sugar units are based in two regions, the Northeast and the Southwest. Any agent not reaching these regions before their sugar reserves run out will die. Any agent forced to move away from these peaks (due to, say, overcrowding) will start losing sugar and in all likelihood die. Therefore any small differences between SU and AU will result in large effects on the population dynamics. When agents have a lower metabolism, as in the other two scenarios, agents can build up reserves of

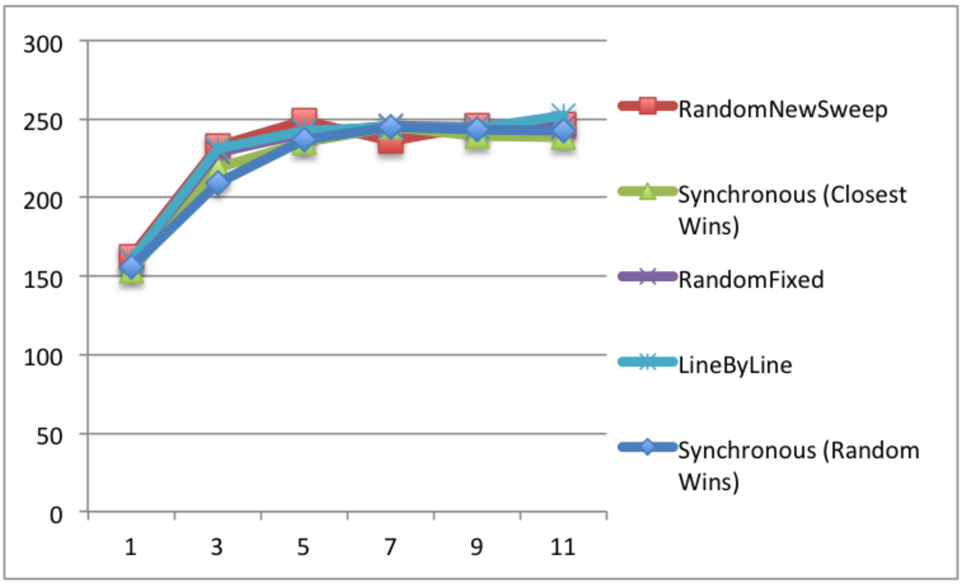


Figure 7.5: Carrying Capacity with varying Vision and Metabolism=1

sugar that allow them to compensate for not finding an optimal destination after each move.

It is also the case that if an agents vision has a greater range there will be more overlap between agents when they move and more frequent collisions (that is, two or more agents attempting to move to the same destination at the same time). As collisions are handled differently by SU and AU (in AU sequential movement prevents collisions) we would expect the differences between the two approaches to be emphasised.

It is clear that the differences betweenSU and AU are subtle and affected by many different factors. Our previous graph of carrying capacity (Figure 7.3) showed differences between the two updating strategies with a different simulation setup. In that case agent metabolism and vision varied from agent to agent and sugar was distributed more evenly throughout the lattice.

We would expect, based on previous work carried out [Huberman and Glance, 1993] in CA-based simulations, that differences will be present in the simulation, the question is how big those differences will be. Given the complex nature of ABM and ABSS simulations it has proven difficult to predict how these differences will manifest themselves.

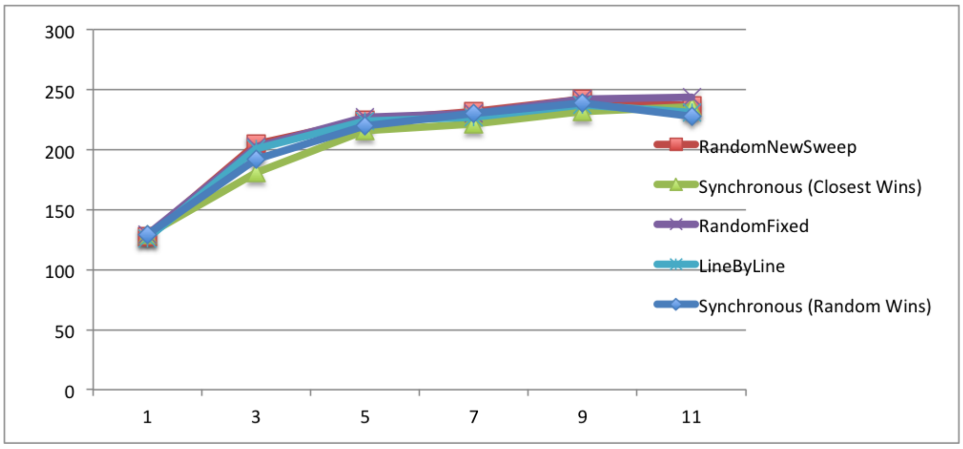


Figure 7.6: Carrying Capacity with varying Vision and Metabolism=2

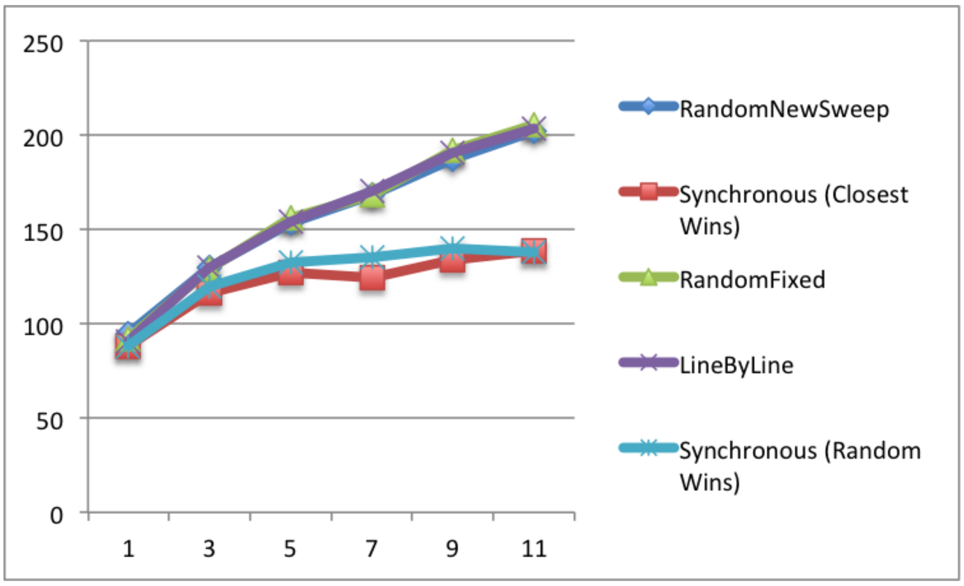


Figure 7.7: Carrying Capacity with varying Vision and Metabolism=3

7.3.4 Convergence of Culture

An agent's culture in Sugarscape is determined by a bit string of uneven length. If an agent's bit string contains more 1's than 0's then they are members of the red group (or culture) and if they have more 0's than 1's they belong to the blue group. Because the bit string contains an odd number of bits there will never be an equal number of 1's and 0's. To simulate agents influencing each others culture or group membership, agents that are neighbours swap a randomly selected bit in their culture string with each other during each step. Agents move across the lattice as before and seek out the best sugar resources to consume. This movement causes the agents to come into contact with different agents as the simulation progresses. The presence of two sugar peaks in the lattice ensures that the agents converge into two separate groups in the lattice.

After each move to a new location each agent will be influenced by its new neighbours. The simulation employs the rules: *Metabolism*, *Growback*, *Movement* and *Culture*. We replicated the configuration of the original experiments as closely as was possible. Simulations were run until the agent culture converged to one or other colour (see Figure 7.8). The number of steps taken before convergence occurred was recorded. We can see in Figure 7.8 a starting population of red and blue agents while Figure 7.9 shows convergence occurring, in this case converging to red.

In the original experiment after approximately 2,700 steps the agents converge to one or other colour across the entire lattice. However, the complete set of initial values used in this simulation is not available to us so we cannot replicate the experiment exactly. For example, the population size can affect convergence times and we have no precise figures on the original agent count. Similarly the distribution of agent's vision attribute values across the population affects population size. To account for these possible variances we have used the same configuration with both AU and SU. Even taking these ambiguities into account the obtained results closely align with the original results.

We found little difference in the average times for convergence to occur regardless of updating strategy (approximately 2,000 steps using AU and 2,200 steps with SU). In each simulation run the convergence time was highly dependent on the initial agent distribution. Occasionally convergence would occur quickly (approximately 1,200 steps) but it could also take much longer to converge (approximately 4,000 steps) and even oscillate between dominating red and blue cultures one or more times before final convergence. The large variance in convergence

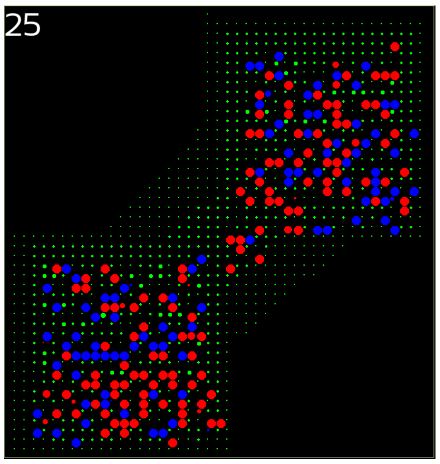


Figure 7.8: Initial Culture Distribution under G_1, M, K

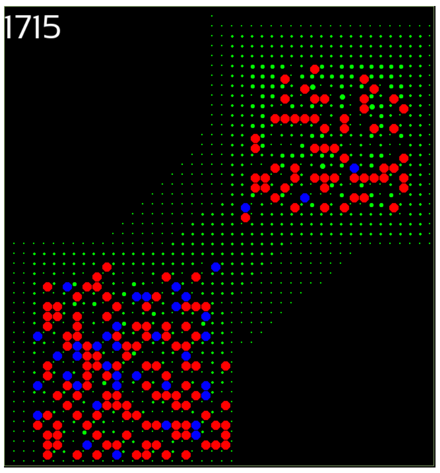


Figure 7.9: Cultural Convergence begins

times makes it difficult to draw any conclusions about differences between the two approaches

We did note that there appears to be less movement of agents between the two separate groupings (on the Northeast and Southwest sugar peaks) on the lattice when SU is employed. It is possible that this may cause the longer convergence times.

7.3.5 Mating and Evolution

Reproduction is used to demonstrate how populations and indeed cultures evolve over time. The original simulation uses the three rule combination of:

Growback Sugar is replenished at each location at a rate of one sugar unit per time step up to the location maximum;

Movement The standard movement rule is employed to allow agents to move around the lattice and interact. Specifically they compete with each other for resources;

Reproduction The sexual reproduction rule that allows male and female agent pairs to produce offspring. To be capable of producing offspring both agents must be fertile and have at least as much sugar in reserve as their initial sugar allocation on creation. Fertility is determined by age. For a female agent fertility begins between the ages of 12 and 15 and ends between the ages of 40 and 50. For a male agent fertility begins at between 12 and 15 and ends between the ages of 50 and 60. The offspring of two agents will inherit its attributes from its parents. That is, metabolism (and vision, etc.) will be randomly chosen to be equal to either its father or mother.

The initial simulation setup is defined as follows:

- For both men and women, childbearing begins between the ages of 12 and 15;
- For women, childbearing terminates between the ages of 40 and 50;
- For men, childbearing age terminates between the ages of 50 and 60;
- For both men and women, the age of natural death is between 60 and 100;
- Members of the initial population have initial endowments in the range 50 to 100 units;
- Metabolism ranging between 1 and 4 sugar units per step;
- Vision is in the range 1 to 6.

Unfortunately the initial setup of the lattice, in terms of carrying load, is not given. This determines the distribution of the high sugar carrying locations and the individual maximum carrying load of each location.

When these parameters are used the simulation showed the population stabilising over time with frequent oscillations of approximately 10% in population size (with peaks to trough lengths of approximately 150 steps). The average metabolism of the agents in the population reduced over time until almost all agents had a metabolism of one. Similarly the average vision of agents in the population increased over time until all agents had a vision of six. This showed that agents with a low metabolism and high vision were more likely to have offspring.

We were able to successfully reproduce these results. The results for metabolism are shown in Figure 7.10. These trends are independent of updating strategy - AU or SU with very little variance between them.

The results for vision correlate with the original simulation when AU is used. When SU is employed average vision does not change noticeably (see Figure 7.11). Under SU vision does not confer as much of a survival advantage on agents. To

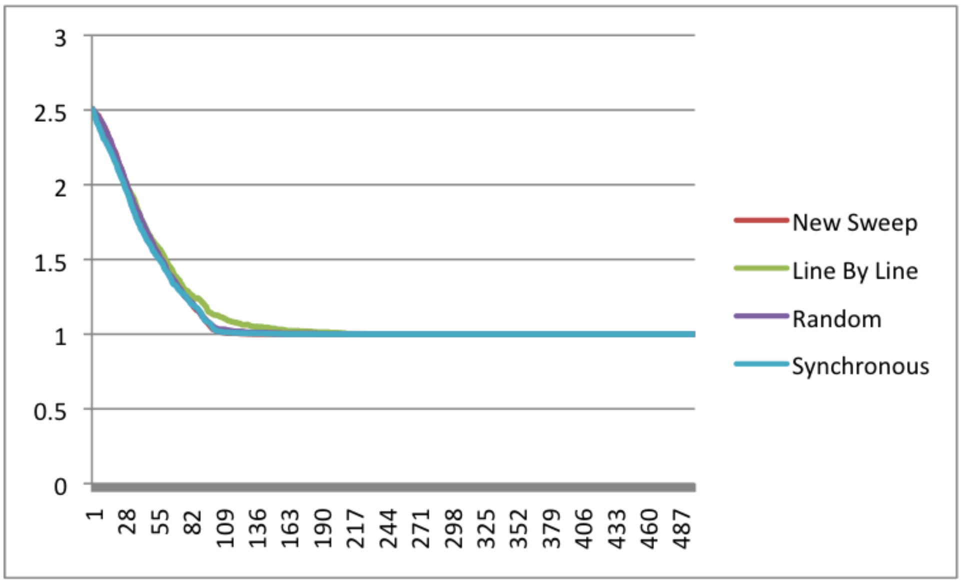


Figure 7.10: Average Metabolism under G_1, M, S

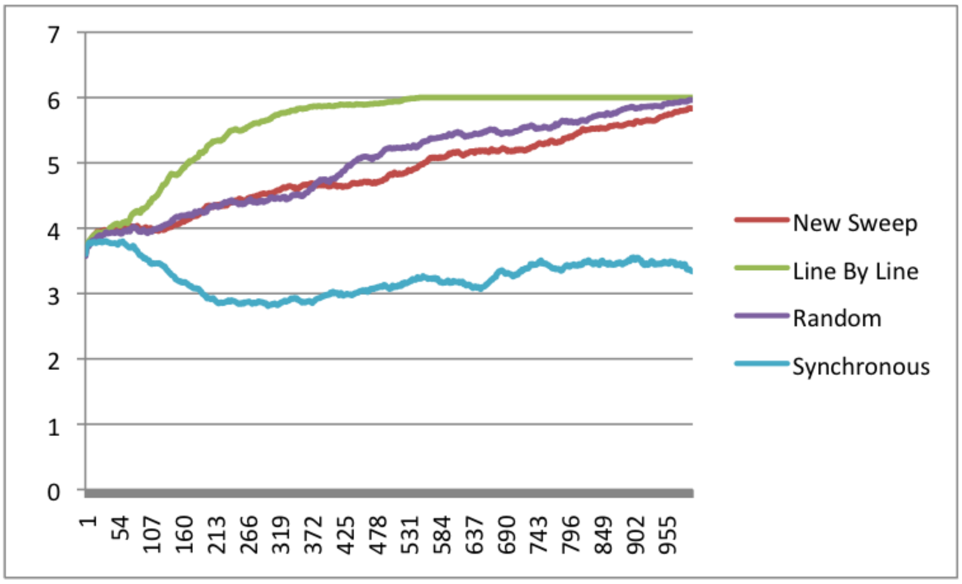


Figure 7.11: Average Vision under G_1, M, S

see why this is the case we must consider what happens under AU and SU when moving. Under AU an agent will pick the best available destination within range and then move to that site. In this case the larger the range the more likely the agent is to find a better destination thereby conferring an advantage on that agent. When SU is employed all agents pick their destination. An agent with a larger range will have more choice as before but they are now competing with all other agents who are also choosing their destinations at the same time. Therefore it is likely that not only will an agent with better vision find and choose a more resource heavy destination but also that other agents will also choose that destination as well. Once more than one agent picks the same destination one is randomly chosen and the others lose out. The losers now have to pick another destination from the remaining unpicked (unpicked because they are lower value) locations. In a crowded lattice this will mean that higher vision will lead, as often as not, to agents having to pick from the unwanted locations once their first choice fails.

In order to determine the effect of employing a collision resolution strategy on these properties we added collision resolution to the *Movement* behaviour. The collision resolution rule simply states that when two or more agents attempt to move to the same destination then the agent closest to the destination wins (gets there first). The losing agent must then choose a different destination not already chosen. Without this rule the winner is chosen at random. We can see from Figure 7.12 that this rule has no effect on the metabolism as it still decreases to one. It does affect the change in vision in that average vision now decreases as population increases and only starts to improve when the population size starts to drop. The collision resolution mechanism makes it much more likely that, in a crowded environment, an agent with a large vision range will be left with the remaining unwanted locations as possible destinations because their first choice was lost to another agent.

The collision resolution rule nullifies much of the survival benefit of having an increased vision in a lattice with concentrated sugar peaks where competition for the best location is intense. Previously (without collision resolution) an agent with a longer vision could see a destination with high sugar rewards from further away and be as likely to move to that location as any other agents. With the collision resolution rule it can still spot the resource but now it will be beaten to the resource by any other agent closer to it. This shows the complexity inherent in ABM where a change in one rule can affect the outcome of another rule.

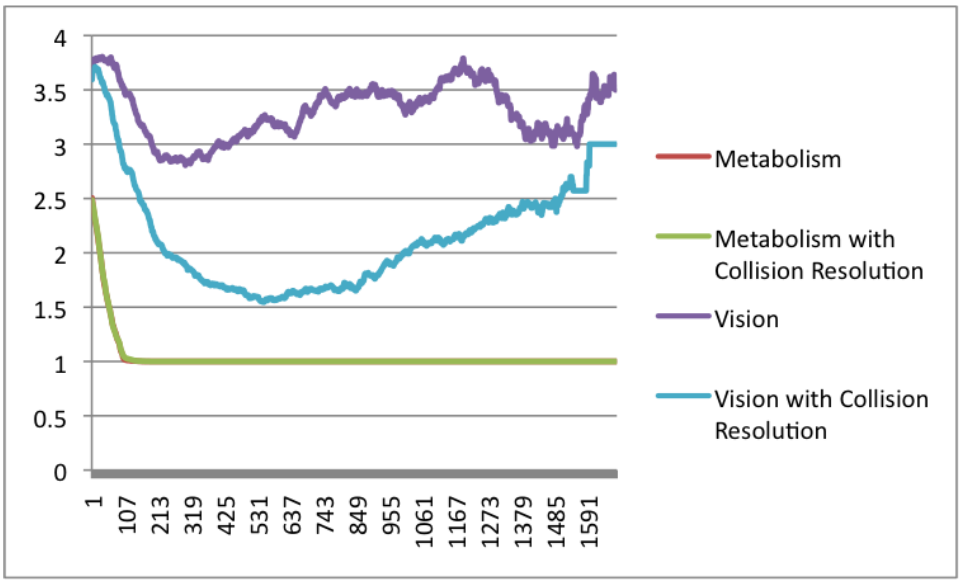


Figure 7.12: Average Vision and Metabolism with Synchronous Updating

The next result proved more difficult to repeat. It was shown that the population is stable under sexual reproduction with only small periodic oscillations in population size. Despite replicating the original setup as closely as possible we were unable to repeat this result. In all cases the population (under all updating strategies) collapsed before 2,500 steps had passed (see Figure 7.13).

The only unknown in the simulation setup is the initial lattice setup. Although we have replicated this as closely as possible, given the information available, without the original details we cannot be sure how close we are to the original. This missing information was enough to derail replication of the original result.

A closer examination of the simulation parameters shows why it is difficult to get a stable population. Female agents are fertile for at most 35 time steps. A stable population requires females to have an average of just over two offspring in their lifetime. Since having a child halves an agents resources the agent must then regain these resources at a rate that allows them a chance of having one or possibly two more children within those 35 steps. If the balance is wrong and they have fewer than two children then the population will collapse. Similarly if they have two many children then the population explosion will deplete the lattice of sugar and result in a population collapse.

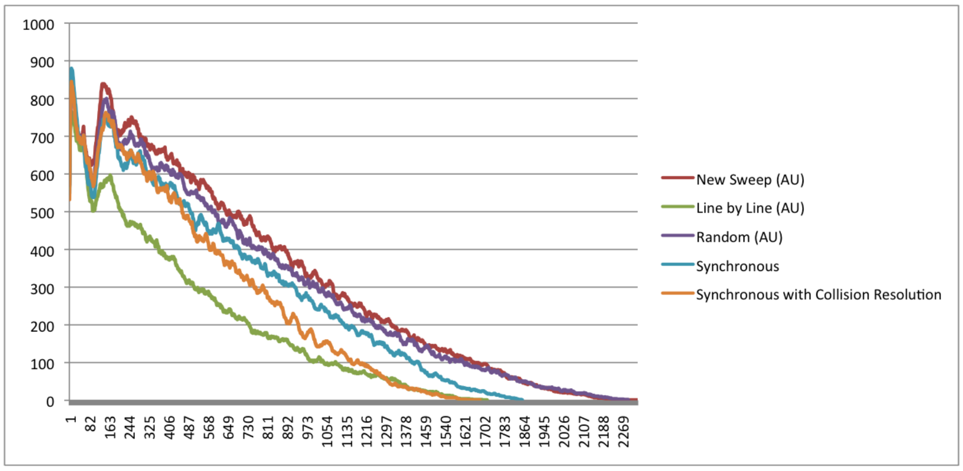


Figure 7.13: Population Collapse with G_1, M, and S

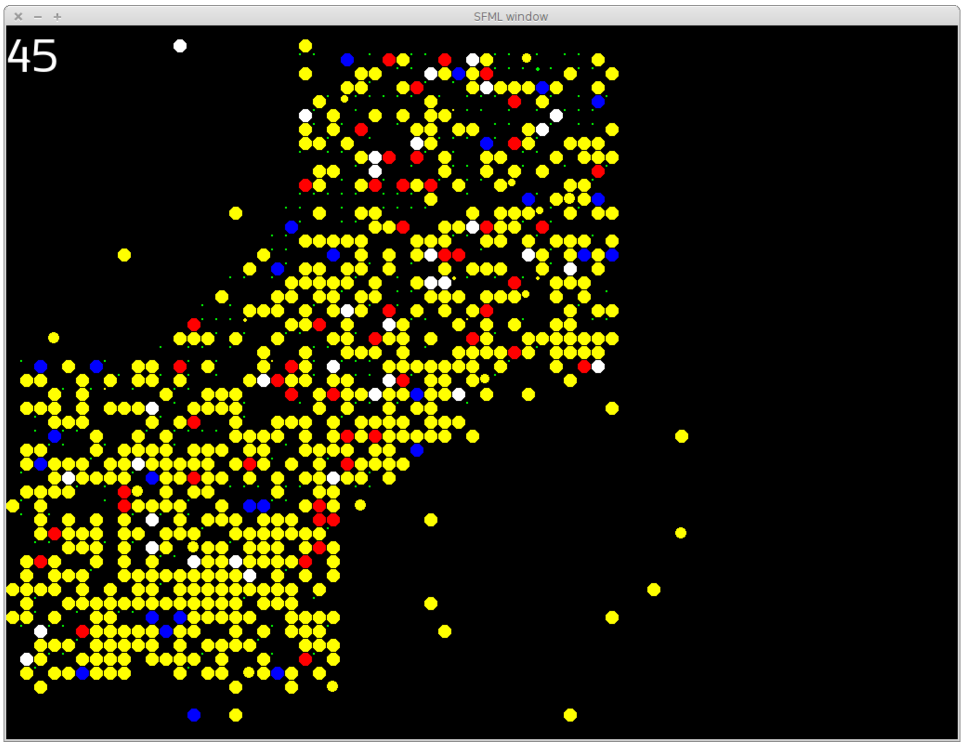


Figure 7.14: Snapshot of Evolution Simulation

We can see this from Figure 7.14. Here white represents a newly born agent, blue and red represent fertile male and female agents respectively, magenta an agent too old to produce offspring and yellow agents who do not have enough sugar to reproduce. As is clear the vast majority of agents do not have enough sugar resources to have children and so cannot produce offspring. Given the short period of time that agents remain fertile and the time it takes for an agent to amass enough surplus sugar resources for children it is unlikely that many agents will ever be in a position to have offspring and even if they do manage to get the resourses required for a short period they then need to be adjacent to another fertile agent of the opposite sex during that time interval.

The operational parameters we gleaned from our reading of Sugarscape does not appear to have the required balance. This part of the simulation needs to be revisited and clarified by the original developers of Sugarscape before reproduction of their results is possible. The ability to run different updating strategies side by side and compare the outcomes was a boon in that it give confidence in any results obtained.

7.4 Conclusion

Sugarscape is a well known complex ABSS that researchers have had difficulty replicating [Bigbee et al., 2007, Gilbert, 2014]. We have shown that by using Formal methods and employing SU alongside AU that:

1. In many cases, where sufficient information is available in the original simulation definition, it is possible to reproduce previous results;

2. When results are not reproducible the formal specification precisely identifies the parameters used and the ambiguities or missing information from the original simulation definition;

3. The use of different updating strategies adds confidence to any results obtained by ensuring results are not due to “artefacts” in the updating process. It also gives confidence in any results obtained even when those results do not agree with the original results that are being replicated. This extra assurance is invaluable.

The approach proposed in this thesis, that of formally specifying the simulation and employing multiple updating strategies for comparison, gives stronger results than just replication of the original results. Based on this approach we have shown that Sugarscape is robust to the updating strategy employed. That is, the results are not "artefacts" due solely to the updating strategy employed. This is information that was not known until now. All results have been verified using multiple strategies and any differences produced by the updating strategies accounted for, even in the case of *Mating and Evolution* where the findings differ from the original findings. The agreement in outcome obtained by all updating strategies gives confidence in the findings and the formal specification allows others to reproduce these results with confidence. Formal specification precisely defines the simulation rules and makes any differences in interpretations visible.

SU has been somewhat neglected in ABM. There are many reasons why this is the case, the lack of SU algorithms for ABM and, as a result, the fact that only AU strategies are built into ABM toolkits. The SU algorithms developed here will prove useful for modellers who want to ensure that their results are not dependent on the updating strategy employed thereby increasing confidence in their simulations.

The available literature notes two main differences between AU and SU in CA. The first difference is that higher population densities within a CA based simulation will result in bigger differences in outcomes between AU and SU. Higher density populations result in greater agent interactions and these interactions are handled differently by SU and AU.

The second difference noted in the current literature is that the use of SU causes more periodicity in simulation dynamics. It is contended here that this periodicity is caused by the consistent speed of information transmission through the simulation space that is imposed by SU. This consistency causes information to spread in waves through the simulation space. Asynchronous updating allows information to spread at random speeds determined by the random order of agent updates during each time step. This disrupts any structure that may cause periodic systemwide behaviour to occur. Here SU has been extended to more complex ABM based simulations. This allows us to see if the differences between AU and AU become more or less pronounced within ABM.

The use of SU within Sugarscape demonstrates that the synchronous algorithms developed in this thesis can cope with the full range and complexity of interactions commonly found in ABM-based simulations. We found that even in the simplest case where the carrying capacity of the Sugarscape lattice is measured that there are distinct differences between SU and AU. The carrying capacity was consistently lower when SU was applied to the simulation. The carrying capacity of the lattice determines the maximum population density that the lattice can support. Given this fact the differences are not unexpected especially in light of previous findings that higher agent densities lead to bigger differences in outcomes.

The differences between the two approaches in stressed environments, e.g. where food resources on the lattice are at a minimum, also lead to bigger differences in outcomes. This again is in line with expectations as even small changes in behaviour can have large effects on a population that is always on the edge of starvation.

SU based simulations result in less variation in outcomes over repeated experiments. This can be accounted for by the fact that AU implicitly uses randomisation to determine updating order within each step unlike SU which is more deterministic in nature. In SU randomisation has to be explicitly introduced through the collision resolution rules employed.

It has proven difficult to predict, in advance, what differences (if any) would be exhibited between the AU and SU approaches. In the simpler Spatial Iterated Prisoners Dilemma the differences were acute and completely divergent but in the more complex Sugarscape ABM the differences were usually non-existent, sometimes minor and occasionally rather more pronounced in character.

When replicating the cultural convergence behaviour in Sugarscape the differences between the AU and SU approaches seemed less apparent. Cultural convergence took slightly longer to occur under SU and there was a tendency for more periodic changes in the dominant culture on the way to convergence. Under AU once one culture started to dominate the lattice it would inevitably reult in convergence to that culture without any changes in fortune along the way. With SU, on the other hand, once a culture started to dominate, while it was likely to become completely dominant, reversals of fortune were possible where the other culture could become dominant. This finding seems to be supported by the literature that SU leads to more cyclic or periodic fluctuations in simulation behaviour but more work is required before any firm conclusions could be drawn from this.

The unpredictability of the effects of updating strategies on complex simulations was also observed when evolution and mating were replicated. Under AU the population evolved better vision (larger viewing distance) over time but when SU was employed vision did not improve. It took closer inspection to reveal that larger viewing distances did not provide any competitive advantage under SU. It is clear that the choice of updating strategy can have unexpected effects on individual rule outcomes and through this on overall simulation outcomes.

It seems that it is only by experimentation that we can see the effects of different updating strategies on the emergent properties of simulations. This makes it more important to incorporate both updating strategies in ABM/ABSS. When differences are caused by different updating strategies the modeller should be able to give a convincing explanation of why those differences are present.

Based on our formal specification we have successfully reproduced many of the results originally shown in [Epstein and Axtell, 1996]. Where reproduction was not possible the use of different updating strategies gave us confidence in our findings and our formal specification allowed us to identify the weaknesses in the original simulation specification that makes those results difficult to reproduce. This shows the efficacy of using formal methods in ABM and ABSS. The production of a formal specification of the simulation [Kehoe, 2015a] alongside the freely available code [Kehoe, 2015b] means that all these results are reproducible by anyone wishing to confirm or challenge these results.

Chapter 8

Conclusion

8.1 Overview of Thesis

In this thesis we have surveyed the field of ABM noting in particular the unreliability of ABM results published by researchers caused by the difficulties of replicating results. These difficulties are the result of insufficient detail and precision provided by ABM researchers when defining their simulation. Replication of results through the re-execution of the original implementation is also problematic in that this can only repeat the errors in the original simulation. What is required is a robust approach that takes these factors into account by reproducing simulations in some implementation independent manner. Such an approach should also compensate for the possibility that the simulation properties might be artefacts of the updating strategy employed.

Formal specification of models alongside the use of multiple updating strategies is proposed in this thesis as a technique for solving the replicability problem. To show the validity of this approach a well-known and complex ABSS, Sugarscape, often used for comparing approaches to ABM simulation, was formally specified and implemented using multiple updating strategies.

In order to extend the type of updating strategies that can be employed in ABM, novel algorithms for extending SU to ABSS were presented and their space-time complexity derived using asymptotic analysis. The algorithms were empirically evaluated and shown to match the expectations from the asymptotic analysis. In developing new SU algorithms for ABM the differences in the properties of AU and SU were explored. these differences can be used by modellers to help decide on whether one approach or the other is more suitable for their purposes and also explain any differences in outcomes that occur. The formal specification and implementation of Sugarscape showed that multiple updating strategies can be used to demonstrate the robustness of previously obtained results.

Although these findings have been demonstrated through the Sugarscape simulation they are applicable to the full range of ABM and ABSS from real-world based simulations to more abstract and virtual simulations. They apply to any model that proposes global system properties emerge from local interactions between individual agents in a population, as long as these agents are not omniscient (that is they have access only to incomplete information[1]) They can be applied to simulations of the physical real world where locality is defined in terms of physical proximity or situations where locality is not based on physical proximity and is more dynamic in nature.

Examples of such simulations include a traffic simulation where overall traffic flow properties emerges from interactions between individual vehicles in close physical proximity to each other or the overall patterns of brain based behaviour that emerge from localised interactions between neurones. Other simulations do not rely on any notion of physical proximity such as the Twitter messaging system where overall properties (trending etc.) emerge from interactions between individual twitter users who interact directly only with their followers and those they follow. Here *locality* is independent of physical location.

8.2 What we Achieved

We have made two contributions to ABSS/ABM in this thesis:

1. An approach to improving replication in ABM;
2. The production of alternative synchronous algorithms for ABSS.

8.2.1 Improving Replication of ABM through Formal Specification

Replication is an open problem in ABM [Edmonds and Hales, 2003, Drummond, 2009]. There is an inability to compare different implementations of the same simulation due to ambiguities in the simulation definition. To allow for proper comparisons to be made between our approach and other approaches to ABM we required repeatable results. That is, we need to be able to present our new approach in such a manner that they can be compared fairly against the alternatives.

[1]Also known as local information.

Our solution is to use formal specifications, already widely studied in computer science, to document precise definitions of simulations. Formal methods have been used up to now mainly for safety critical systems in order to prove the correctness of the system definition and the algorithms that implement them. Up until now they have not been used to help researchers create simulations that are reproducible by other researchers. We demonstrate that they can also be used to provide a high level description or specification of an ABM. This specification can then be used to verify that different implementations of the same model conform to the same precise model definition.

We achieved this by choosing a representative ABSS, Sugarscape, that is well-known [Epstein and Axtell, 1996] and contains a wide variety of agent interactions of varying complexity and formally specifying it using the Z notation [Spivey, 1989]. This specification allows modellers who wish to compare simulation approaches to use a well defined simulation and so make proper comparisons. At present each Sugarscape simulation is a unique interpretation [Bigbee et al., 2007, Gilbert, 2014] making it difficult to fairly compare the outcomes between any two implementations. Our specification has identified ambiguities in the original definitions and is specified in a manner agnostic to updating approach. Once an ABM has been specified formally it solves one of the major problems of replication, that of different researchers misinterpreting the simulation definition.

The issue of whether modellers are able or willing to undertake an extensive formal specification process is not directly addressed in this thesis. However, if it is the case that modellers want the precision required to allow for replication then there are few alternatives available and none that give the same level of precision as is offered by formal specification.

Our specification, summarised briefly in Chapter 5 and more extensively in Appendix C, has been made available online [Kehoe, 2015a] along with our reference implementation [Kehoe, 2015b].

This approach of using formally specification to help tackle the replication problem in ABM has been published and presented at the Multi Agent Based Simulation Workshop held at AAMAS in 2016 [Kehoe, 2016a].

8.2.2 Detecting Artefacts through Multiple Updating Strategies

The use of AU within ABM [Huberman and Glance, 1993] and the attendant issues were surveyed [Cornforth et al., 2005], [Caron-Lormier et al., 2008]. Our analysis shows that AU can, especially in higher density populations, allow agents to gain non-local knowledge instantaneously - something not allowed within ABM. This allows us to explain some of the differences in outcomes between synchronous and asynchronous updating already identified by other researchers. These are :

1. Higher density populations in a simulation where bounded rationality is important will result in bigger differences between the asynchronous and synchronous outcomes [Caron-Lormier et al., 2008] than lower density populations in the same simulations;

2. Asynchronous updating will cause simulations to be less likely to display periodicity [May, 1973].

The identification of these issues provides guidance for modellers choosing between an asynchronous or synchronous approach to simulation design.

Following on from our study of these updating techniques we developed algorithms that allow SU to be extended to ABM. Our algorithms handle all types of interaction, even the most complex, thus increasing the scope of synchronous updating from simple CA-based simulations to the more complex ABSS.

We proved, using asymptotic analysis that our algorithms belong to the $n \log n$ complexity class. Although this is slightly worse than asynchronous updating which is linearly bound, n, it is still polynomially bound - a time complexity that is generally considered acceptable. We also established that the algorithms demonstrate good *Weak Scalability* and are *Deadlock* and *Livelock* free. These SU algorithms are deterministic in execution, further differentiating them from the alternative AU algorithms.

Finally, we produced a framework for our algorithms to show their practicality. This framework serves as a starting point for any researchers thinking of using a synchronous approach in an ABM. We used our framework to implement a cross section of Sugarscape rules (or behaviours) ensuring that we implemented at least one rule belonging to each category or type of interaction. We verified that they could all be implemented synchronously and with empirical testing (summarised in Appendix A) confirmed that they conform to our asymptotic analysis.

Using this framework this thesis demonstrated that Sugarscape is robust to the updating strategy employed, something that was unproven until now. The emergent properties of Sugarscape were shown to be independent of the updating strategy employed. In all cases the results show little divergence and identical emergent properties. This is a new finding as until now the use of SU within ABM has been rare, due to the lack of any useful SU algorithms that can handle ABM interactions. The comparisons shown here have previously only been attempted for simple (CA-based) simulations that do not contain the complex types of interactions present in an ABM such as Sugarscape.

The analysis in section 6.5.1 showed that CA simulations are more suspectable than ABM to instantaneous information transfer accross localities. It suggests that the major differences that have been shown to occur in CA-based simulations under SU and AU updating strategies [Huberman and Glance, 1993] do not necessarily occur in more complex ABM. This is bourne out by the comparisons in Chapter 7.

The results obtained by our framework (Chapter 7) on the use of multiple updating strategies in reproducibility have been published at the European Simulation and Modelling conference in 2016 [Kehoe, 2016b].

Appendix A

Empirical Benchmarks

A.1 Overview of Benchmarks

These benchmarks contain behaviours as complex (if not more so) as those displayed in any other synchronous algorithms from the literature. We have shown that we can implement complex behaviours, their time complexity matches the expectations from the asymptotic analysis and that the algorithms scale as expected.

Sequential execution of our algorithms, using pseudo random numbers gives deterministic execution. In the concurrent version of our algorithms there is some nondeterminism caused by the sharing of a single random number generator amongst all threads. This can be removed by a more sophisticated approach to pseudo random number generation.

As we have already stated the framework is not optimised. This is to make the code as easy to understand as possible. When performing agent behaviours the framework iterates over all locations instead of iterating over only locations that contain agents. For this reason we are not overly concerned with the absolute speeds obtained, although they are fast, but with how it scales as (i) the size and complexity of the simulation grows and (ii) the number of simultaneous threads increases.

Every major simulation toolkit (Swarm, Repast, Mason and NetLogo) comes with a partial implementation of Sugarscape that demonstrates that toolkit's approach to simulation [Railsback et al., 2006a, Berryman, 2008, Inchiosa and Parker, 2002]. In chapter four we produced a formal specification of Sugarscape thus making proper analysis of our results possible and allowing other researchers to compare their own implementations to ours. If we compare Sugarscape to the *StupidModel*

[Railsback et al., 2005] benchmark we can see that although *StupidModel* has been used by some ABM researchers to show how fast their approach to implementing ABM is [Lysenko and D'Souza, 2008], its most complex behaviour only matches that of the simplest benchmark presented here. This makes the results from our framework more complete and relevant. We note in passing that the implementation of even the sophisticated Sugarscape algorithms is simplified by the use of our algorithm. This is because all concurrency is handled transparently by our algorithm.

A.1.1 Sequential Benchmarks

Introduction

We have tested our framework on five different Sugarscape rule combinations. The rules employed in each benchmark are given below:

Movement *Metabolism, Growback, Movement*;

Culture *Metabolism, Growback, Movement, Culture*;

Disease *Metabolism, Growback, Movement, Disease Transmission*;

Combat *Metabolism, Growback, Combat*;

Mating *Metabolism, Growback, Movement, Death, Mating*.

These represent a broad cross-section of the different rule types that are contained in Sugarscape. All combinations include the simple *Growback* and *Metabolism*[1] (categorised as *Independent* rules), and at least one *Write-dependent* rule - either *Movement* or *Combat*. They are included because without these rules and some form of movement there can be no interesting behavioural interactions between agents. As *Write-dependent* rules are the most complex, and the most difficult to implement synchronously, we have included different variants of this type of rule (*Movement*, *Combat* and *Mating*). The original rule definitions are available in Appendix A, and formally defined in Appendix C.

We note that even the simplest of our tests contains a rule that is as complex as any attempted by the other synchronous approaches we have looked at [Fatès and Chevrier, 2010, Spicher et al., 2010, Caron-Lormier et al., 2008].

[1]Metabolism is not stated as a separate rule in the original Sugarscape simulation rule. Our specification process identified this as an ambiguity and corrected this by explicitly stating it (as the *Tick* rule).

Where a rule had a random element (such as randomly choosing between different destinations when moving) we used the *Mersenne Twister* Pseudo Random Number Generator [Matsumoto and Nishimura, 1998]. In the sequential algorithm implementation, this ensures determinacy.

The tests were all executed on the same platform for consistency. In order to obtain accurate timings we repeated the test 50 times and used the average, minimum and maximum times. In each case the simulation ran for 200 steps so the times shown represent the amount of time taken (in seconds) for 200 steps to run. Running a simulation for 200 steps allows the simulation to execute a representative number of scenarios during each test run. These tests were conducted across 10 different lattice sizes with different initial agent population sizes as follows:

1. 10×10: 100 Locations with 25 Agents;
2. 20×20: 400 Locations with 100 Agents;
3. 30×30: 900 Locations with 225 Agents;
4. 40×40: 1,600 Locations with 400 Agents;
5. 50×50: 2,500 Locations with 625 Agents;
6. 60×60: 3,600 Locations with 900 Agents;
7. 70×70: 4,900 Locations with 1,225 Agents;
8. 80×80: 6,400 Locations with 1,600 Agents;
9. 90×90: 8,100 Locations with 2,025 Agents;
10. 100×100: 10,000 Locations with 2,500 Agents;

The results are graphed as time (*seconds*) against number of agents in the simulation. We chose to use this lower figure of agent population size instead of lattice size because the (time) dominant rules apply only to agents. Only the simplest rule, *Growback*, acts on every location. Using agent population size is, we feel, the more accurate and fairer measure to use and it will, if anything, underestimate the efficiency of our algorithm.

In the case of the last two tests the population size varied during the simulation as agents died (through combat, old age or starvation) and new agents were added by mating between agents. In these cases we measured the final carrying capacity (in terms of agent population) of the lattice as our final population size.

Results

The graphs for each of the five benchmarks are given below. The tables of results used to generate the graphs are included alongside the graphs. The results are within the growth rates predicted by the asymptotic analysis. In the case of the first four benchmarks (*Movement,Culture, Disease* and *Combat*) the growth rate is actually linear. This is not unexpected given the numbers of agents in the simulations. We have graphed *Disease* and *Mating* separately for clarity.

Combat is a type of movement rule. It adds extra complexity because as well as the issue of agents attempting to move to the same location we have the difficulty that different agents can simultaneously try attack the same agent, or forming conflict chains, for example agent a_1 might want to attack agent a_2 who simultaneously wants to attack agent a_3 and so on. Despite this there is almost no extra overhead in implementing this behaviour when compared to simple movement. Culture, a *Read-Dependent* rule, has the longest running time of these three rules due to the amount of work involved in the application of this rule to each agent.

Table A.1: $\{$**Growback,Movement**$\}$

Lattice Size	Initial Pop.	Average Time [sec]	Min Time [sec]	Max Time [sec]
10×10	25	0.06	0.05	0.06
20×20	100	0.22	0.21	0.24
30×30	225	0.49	0.47	0.49
40×40	400	0.85	0.83	0.86
50×50	625	1.33	1.31	1.35
60×60	900	1.91	1.89	1.94
70×70	1,225	2.62	2.58	2.84
80×80	1,600	3.49	3.40	3.60
90×90	2,025	4.34	4.30	4.57
100×100	2,500	5.42	5.37	5.55

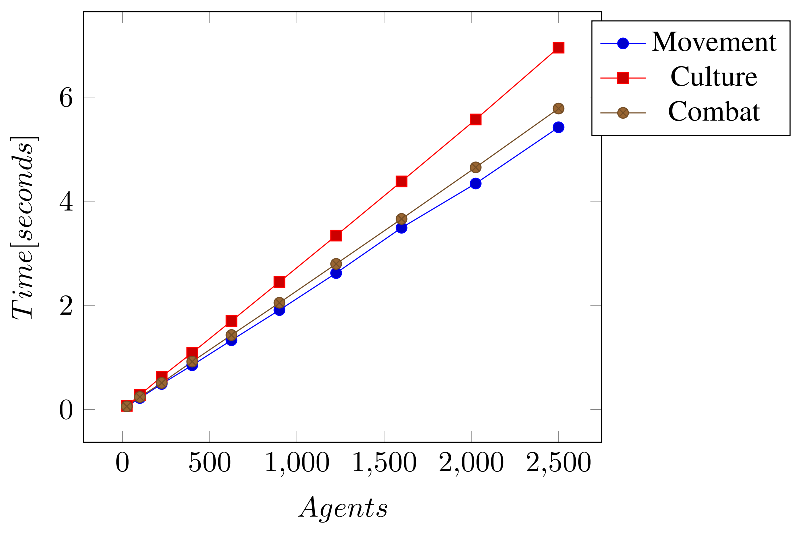
Table A.2: {**Growback, Combat**}

Lattice Size	Initial Population	Average Time [sec]	Min Time [sec]	Max Time [sec]
10×10	25	0.06	0.05	0.07
20×20	100	0.24	0.23	0.25
30×30	225	0.51	0.49	0.52
40×40	400	0.92	0.91	0.94
50×50	625	1.43	1.42	1.45
60×60	900	2.05	2.03	2.06
70×70	1,225	2.79	2.76	2.82
80×80	1,600	3.66	3.64	3.70
90×90	2,025	4.65	4.61	4.69
100×100	2,500	5.78	5.72	5.90

Table A.3: {**Growback, Movement, Culture**}

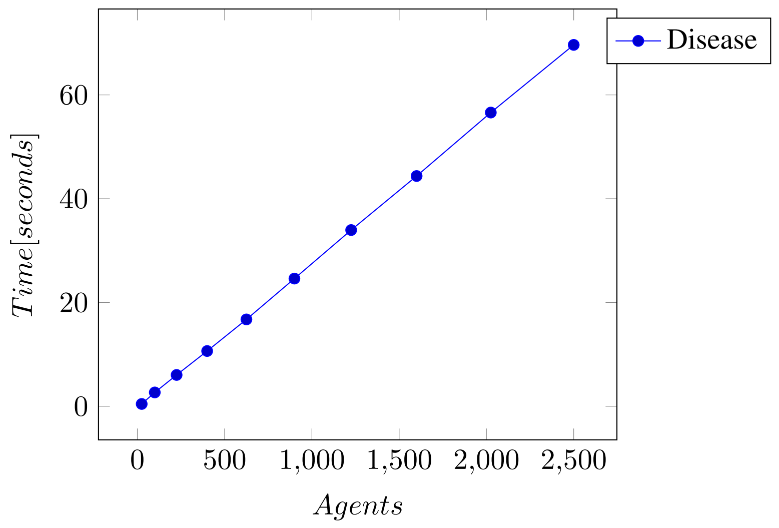
Lattice Size	Initial Population	Average Time [sec]	Min Time [sec]	Max Time [sec]
10×10	25	0.07	0.07	0.08
20×20	100	0.28	0.27	0.28
30×30	225	0.63	0.60	0.63
40×40	400	1.09	1.08	1.13
50×50	625	1.70	1.68	1.73
60×60	900	2.45	2.43	2.47
70×70	1,225	3.34	3.32	3.39
80×80	1,600	4.38	4.33	4.41
90×90	2,025	5.57	5.53	5.64
100×100	2,500	6.95	6.91	7.01

1. Movement, Culture and Combat



Disease is, like culture, a *Read-Dependent* rule. Surprisingly it takes the longest time of all our tests to complete. This has to do with the complexity of the rule, rather than any properties of our synchronous algorithm, as it requires large amounts of substring checking and calculation of *Hamming distances* [Russell and Cohn, 2012] during each interaction.

2. Disease



The final test, *Mating* (see table A.5), is the most complex. It contains a total of five different rules, three *Read-dependent* (*Growback, Metabolism* and *Death*) and two *Write-dependent* (*Movement* and *Mating*). The mating rule requires three collaborators to work together exclusively: one male agent, one female agent and

Table A.4: {**Growback, Movement, Disease Transmission**}

Lattice Size	Initial Pop.	Average Time [sec]	Min Time [sec]	Max Time [sec]
10×10	25	0.45	0.42	0.69
20×20	100	2.67	2.60	2.94
30×30	225	6.04	5.92	5.99
40×40	400	10.65	10.60	10.71
50×50	625	16.74	16.60	16.96
60×60	900	24.61	24.42	25.30
70×70	1,225	33.96	33.76	34.43
80×80	1,600	44.37	44.04	45.16
90×90	2,025	56.60	55.95	61.71
100×100	2,500	69.64	69.17	73.96

one empty location to place their offspring in. There is, therefore, more scope for clashes between agents searching for both partners and empty locations. In addition the mating rule requires that an agent mates with *all* its available neighbours within the one step, as opposed to just one. This requires us to apply the mating rule up to four times for each agent during a single step. We would expect that higher density populations will cause more clashes and therefore require more work. We also have a guarantee that the Mating rule will, by definition, provide those higher populations.The population size increases quickly and settles on the maximum carrying capacity (where the number of deaths and births cancel each other out) of the lattice.

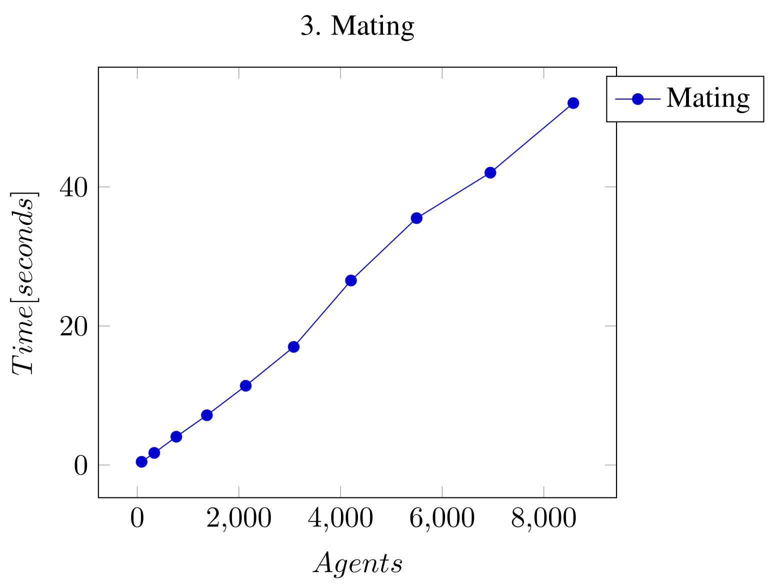


Table A.5: {**Growback, Movement, Death, Mating**}

Lattice Size	Initial Pop.	Final Pop.	Avg. Time [sec]	Min Time [sec]	Max Time [sec]
10 × 10	25	85.76	0.46	0.44	0.54
20 × 20	100	334.84	1.74	0.04	1.82
30 × 30	225	768.96	4.07	3.92	4.06
40 × 40	400	1,370.72	7.16	7.03	7.27
50 × 50	625	2,135.22	11.40	11.27	11.51
60 × 60	900	3,079.10	16.99	16.72	17.09
70 × 70	1,225	4,206.00	26.54	23.83	26.89
80 × 80	1,600	5,497.38	35.51	35.05	35.82
90 × 90	2,025	6,948.50	42.05	40.94	45.79
100 × 100	2,500	8,581.00	52.06	51.17	53.92

A.1.2 Concurrent Benchmarks

Introduction

We measured the *speed-up* and *efficiency* (see section 3.3) attained by our concurrent implementation across a range of simulation sizes. We show the test results on five increasing simulation sizes from 100 × 100 to 500 × 500. The tests were also run on lattice sizes from 10 × 10 to 100 × 100. The tests returned the same results as those of the larger lattices so for clarity we show only the larger lattice results. However, for the sake of completeness, the graphs showing the results of the smaller lattice sizes are shown in Appendix C.

The code has been parallelised by using OpenMP (section 3.3.5) on the original sequential code. This provided a number of advantages. Most importantly, given the structure of our algorithms (see section 5.7) we can be sure that, if the sequential version is correct, the parallelisation of the algorithm using OpenMP will also be correct.

The parallelised code benchmarked here is unoptimised and does not use the tiling algorithm from Chapter Five. The framework is a proof of concept and although the tiling algorithm is $\Theta(1)$ the naive algorithm here is still just $\Theta(n \log n)$. The tiling algorithm also has higher overhead, so tile sizes would need to be quite large before it became a viable alternative to the naive algorithm. Tiling[2] is more suited to HPC or GPGPU scenarios.

The code was tested on a four core hyper threaded processor allowing eight simultaneous threads. The tests were run using one, two, four and eight threads.

Results

Overall the best speed ups are obtained when we employ eight threads, giving a four to six-fold speedup across all benchmarks. This is not unexpected as the processor can run a maximum of eight threads simultaneously. The efficiency results, on the other hand, are highest, giving an efficiency of 90% or higher, when we use only two threads. Again this is what we would expect as adding more concurrency increases overhead. The general trend is for increased speed-up and decreasing efficiency as we increase the number of threads up to the maximum capacity of the processor.

When the thread count exceeds the number of available processors (cores) speed-up decreases along with efficiency. For example, when we increase the thread count to sixteen threads on the *Disease* benchmark we find that speed-up changes from the peak of a six-fold speed up to a four-fold one. Similarly efficiency drops, when using sixteen threads, to just 24%.

In general terms the speed-up and efficiency figures do not vary much as the size of the simulation increases. The benchmark reporting the best speed-up is disease. This is due to the fact that the disease rule requires a large amount of

[2]The tiling algorithm is implemented within the framework.

work and is only a *Read-Dependent* rule. *Read-Dependent* rules are embarrassingly parallel so they can be parallelised easily and efficiently. When we combine this with the larger amount of work each agent interaction requires the resulting configuration means that we get more work done with less overhead.

Similarly we note that the simplest benchmark, *Movement*, gives the smallest speed-up, again due to the small amount of work that is required in each step. As the amount of work in a step increases it is easier to split this work into tasks large enough to outweigh the overhead in managing those tasks. A sophisticated implementation would be used to estimate, at run-time, how many tasks a step should be broken up into and choose appropriately.

Table A.6: {**Growback,Movement**} Speed-up

Lattice Size	Two Threads	Four Threads	Eight Threads
100×100	1.856592383	2.316808729	4.040902376
200×200	1.898244521	3.256262641	4.404530931
300×300	1.896369805	2.320339753	3.308738612
400×400	1.873241962	2.716595605	3.504801884
500×500	1.844613145	2.941228447	3.639691039

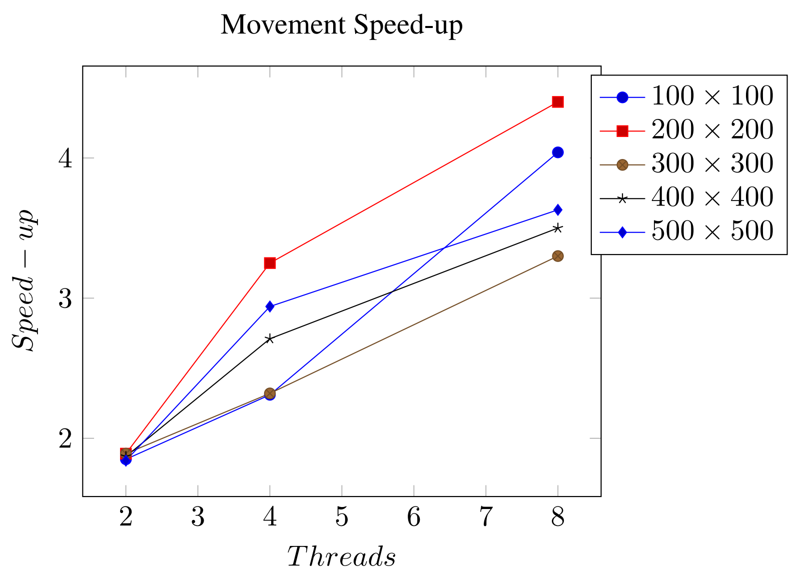
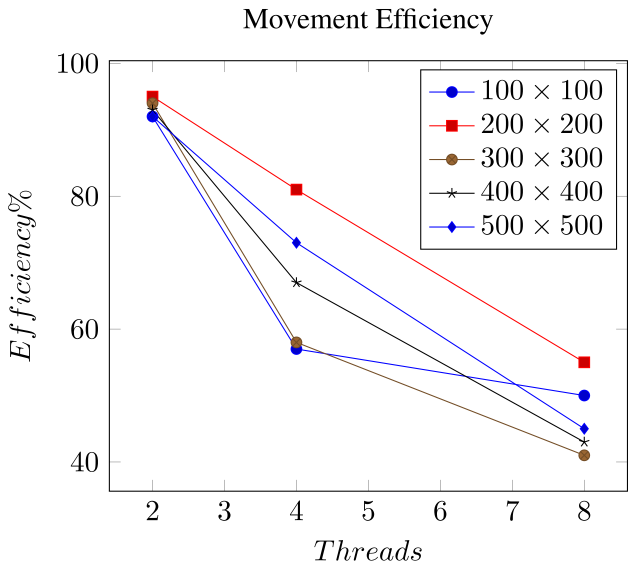


Table A.7: {**Growback,Movement**} Efficiency

Lattice Size	Two Threads	Four Threads	Eight Threads
100 × 100	0.928296191	0.579202182	0.505112797
200 × 200	0.94912226	0.81406566	0.550566366
300 × 300	0.948184902	0.580084938	0.413592327
400 × 400	0.936620981	0.679148901	0.438100235
500 × 500	0.922306573	0.735307112	0.45496138



The *Combat* rule is a replacement for *Movement* and has very similar characteristics. It is, as we would expect, more efficiently parallelised beacuse each thread does more work for the same amount of overhead (required for the management of the threads).

Table A.8: {**Growback,Combat**} Speed-up

Lattice Size	Two Threads	Four Threads	Eight Threads
100 × 100	1.844365113	3.221019987	3.840129208
200 × 200	1.869642776	3.340392034	4.110385641
300 × 300	1.888391438	2.78676231	4.246000715
400 × 400	1.900131374	3.07059183	4.248822599
500 × 500	1.887168414	3.132662766	4.268017797

Combat Speed-up

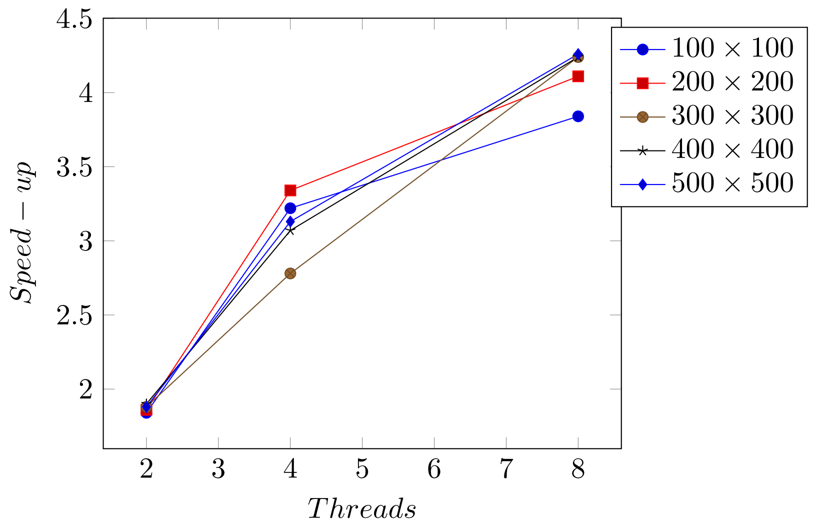
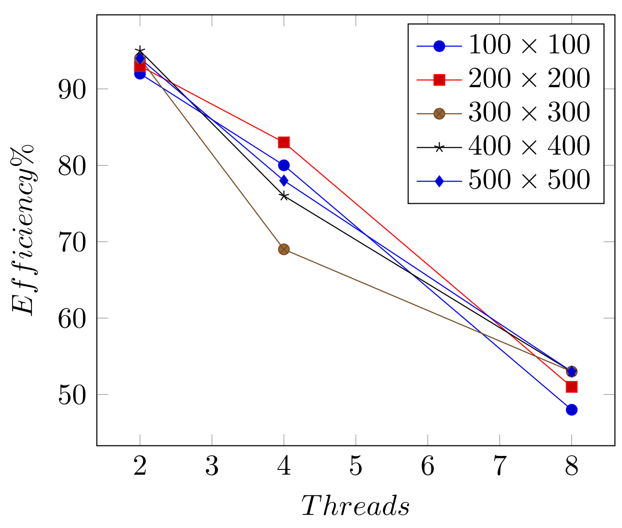


Table A.9: {**Growback,Combat**} Efficiency

Lattice Size	Two Threads	Four Threads	Eight Threads
100 × 100	0.922182557	0.805254997	0.480016151
200 × 200	0.934821388	0.835098008	0.513798205
300 × 300	0.944195719	0.696690578	0.530750089
400 × 400	0.950065687	0.767647958	0.531102825
500 × 500	0.943584207	0.783165691	0.533502225

Combat Efficiency



Culture follows a similar trend to *Movement* and *Combat* but with slightly better characteristics. This is because it adds a *Read-Dependent* rule that requires slightly more work. This combination of an easy to parallelise behaviour that contains enough work to outweigh the overhead introduced by concurrency explains why this is the case.

Table A.10: {**Growback,Movement,Culture**} Speed-up

Lattice Size	Two Threads	Four Threads	Eight Threads
100×100	1.873178192	3.059282307	4.165631711
200×200	1.917506126	3.320972395	4.482960397
300×300	1.916701711	3.358060763	4.492373229
400×400	1.886418791	3.313277367	4.306234614
500×500	1.930745683	2.902163545	4.26821781

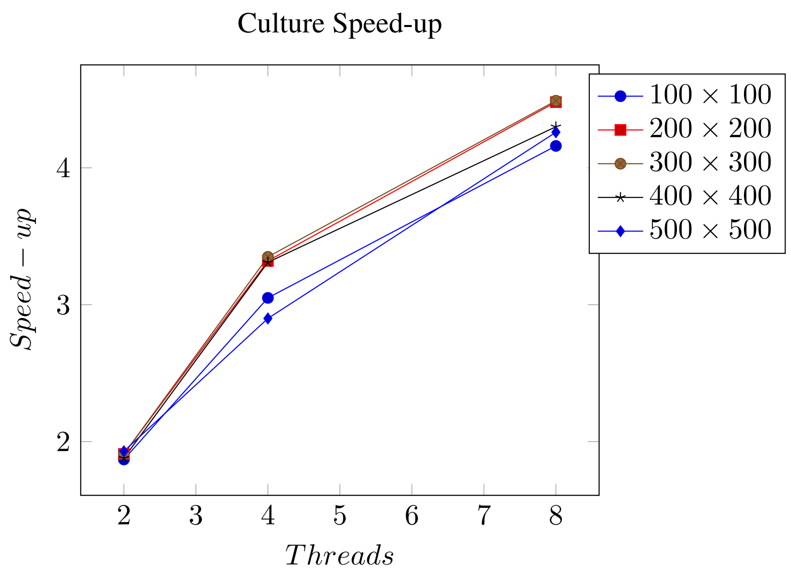
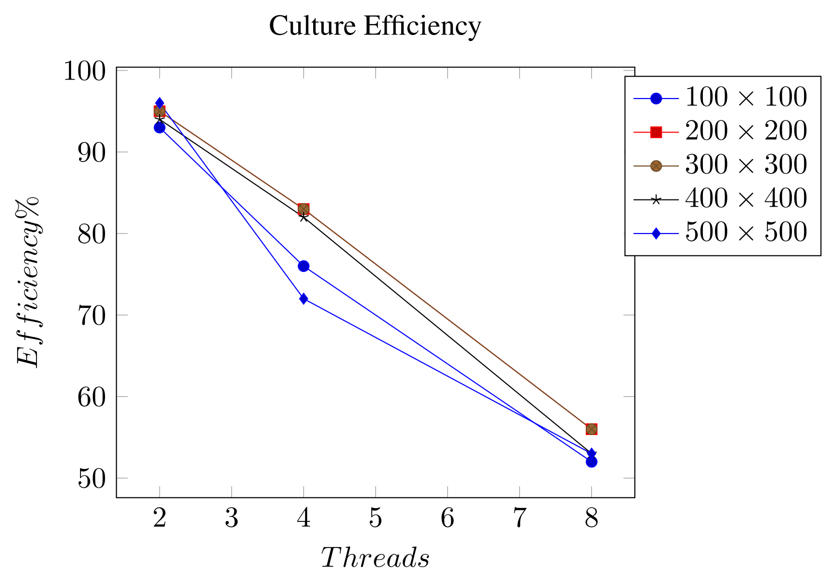


Table A.11: {**Growback,Movement,Culture**} Efficiency

Lattice Size	Two Threads	Four Threads	Eight Threads
100×100	0.936589096	0.764820577	0.520703964
200×200	0.958753063	0.830243099	0.56037005
300×300	0.958350856	0.839515191	0.561546654
400×400	0.943209395	0.828319342	0.538279327
500×500	0.965372842	0.725540886	0.533527226



Disease is the most computationally expensive rule we test. As it is *Read-Dependent* we expect it to scale well. It follows the trends of the other tests but we note that for lattices up to 300 × 300 it scales much better (six-fold speed-up) than the other rules. Larger lattice sizes have smaller speed-ups but are still in line with (four-fold speed-ups) other tests. This is most apparent in the graph of efficiency. We can see here that the smaller lattice sizes show an increase in efficiency with eight threads. Overall it is more efficient than the other tests.

Table A.12: {**Growback,Movement,Disease**} Speed-up

Lattice Size	Two Threads	Four Threads	Eight Threads
100 × 100	1.90135969	2.691835305	6.451354003
200 × 200	1.933222497	2.7844115	6.341162233
300 × 300	1.926430771	2.62880582	6.003707175
400 × 400	1.956418239	3.002138473	5.029375532
500 × 500	1.957396173	3.204833584	4.837984509

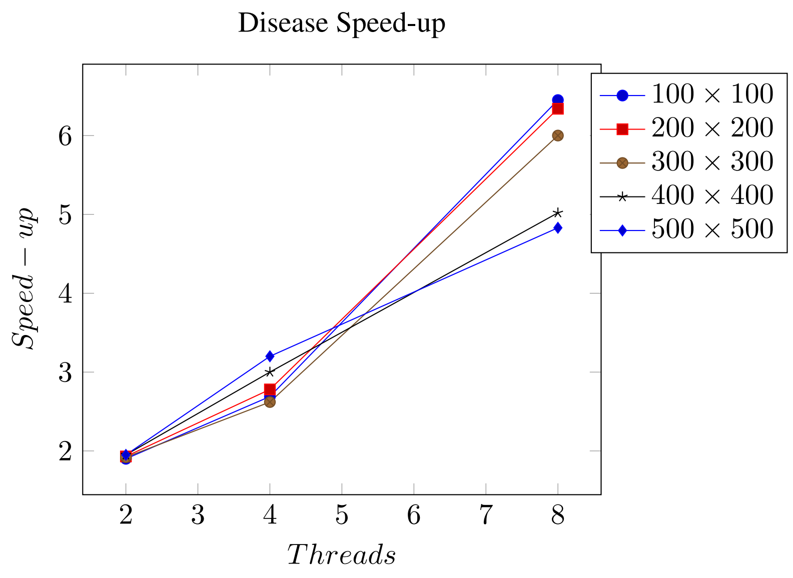
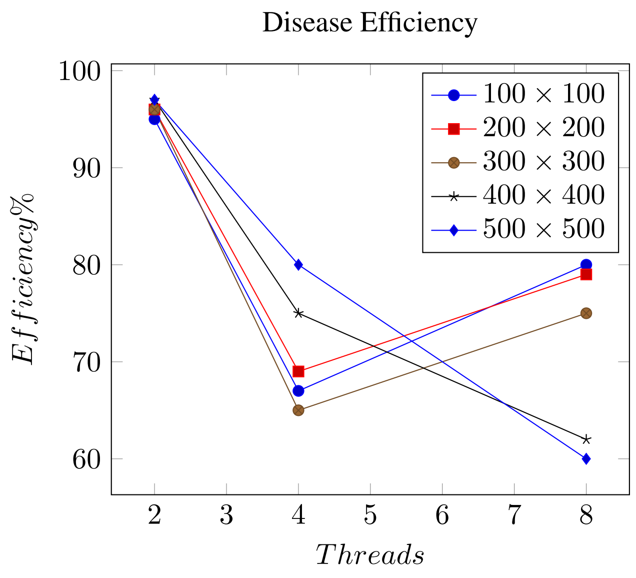


Table A.13: {**Growback,Movement,Disease**} Efficiency

Lattice Size	Two Threads	Four Threads	Eight Threads
100 × 100	0.950679845	0.672958826	0.80641925
200 × 200	0.966611248	0.696102875	0.792645279
300 × 300	0.963215385	0.657201455	0.750463397
400 × 400	0.97820912	0.750534618	0.628671942
500 × 500	0.978698086	0.801208396	0.604748064



The final test employs the *Mating* rule and so each step has two *Write-Dependent* rules. These are the least efficiently parallelised and the *Mating* rule uses repeated iterations of the rule in each step to allow an agent to mate with every available neighbour. The results do not deviate from what we would expect and are in line with the other tests.

Table A.14: {**Growback,Movement,Mating,Death**} Speed-up

Lattice Size	Two Threads	Four Threads	Eight Threads
100×100	1.827330351	2.665150018	4.60940405
200×200	1.835303044	2.730033149	4.605288412
300×300	1.828112653	2.737235586	4.2809175
400×400	1.8242413	2.829385238	3.880333266
500×500	1.819870414	2.821508628	3.881703571

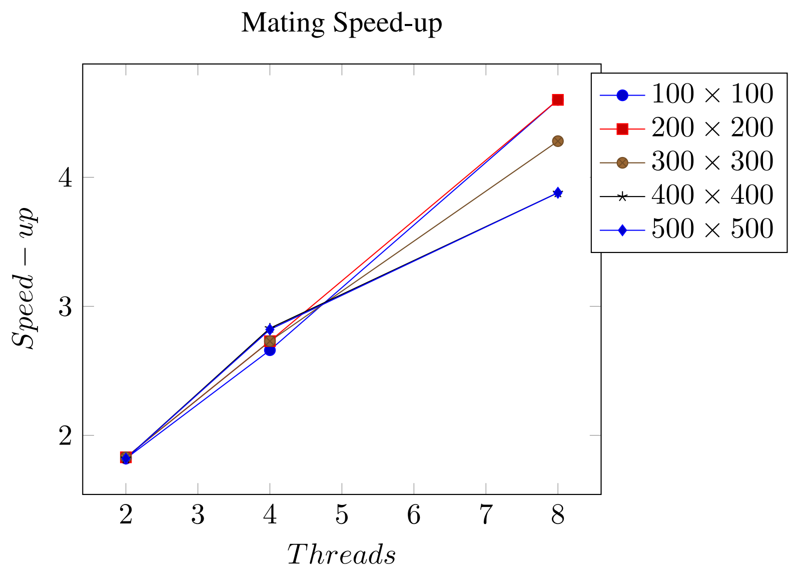
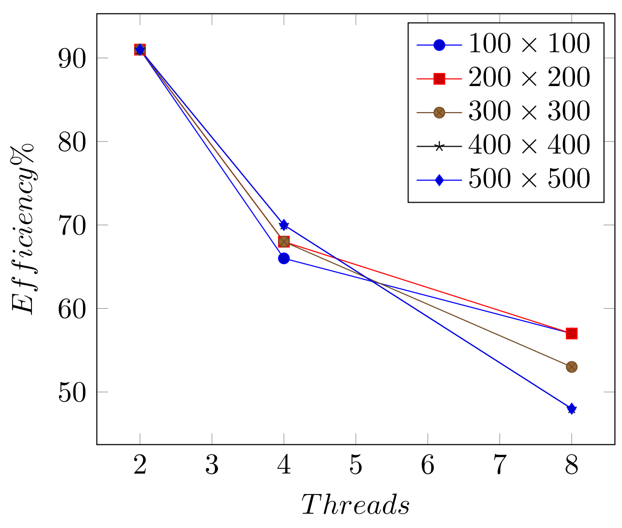


Table A.15: {**Growback,Movement,Mating,Death**} Efficiency

Lattice Size	Two Threads	Four Threads	Eight Threads
100×100	0.913665175	0.666287504	0.576175506
200×200	0.917651522	0.682508287	0.575661052
300×300	0.914056326	0.684308897	0.535114688
400×400	0.91212065	0.707346309	0.485041658
500×500	0.909935207	0.705377157	0.485212946

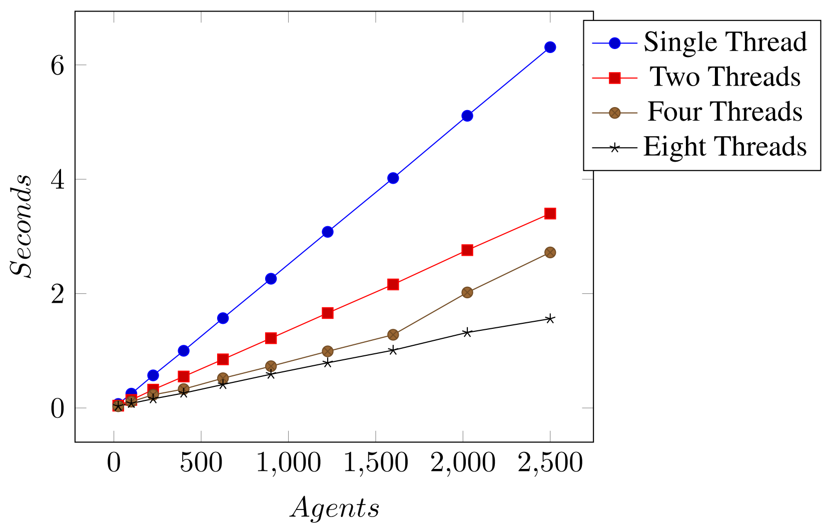
Mating Efficiency

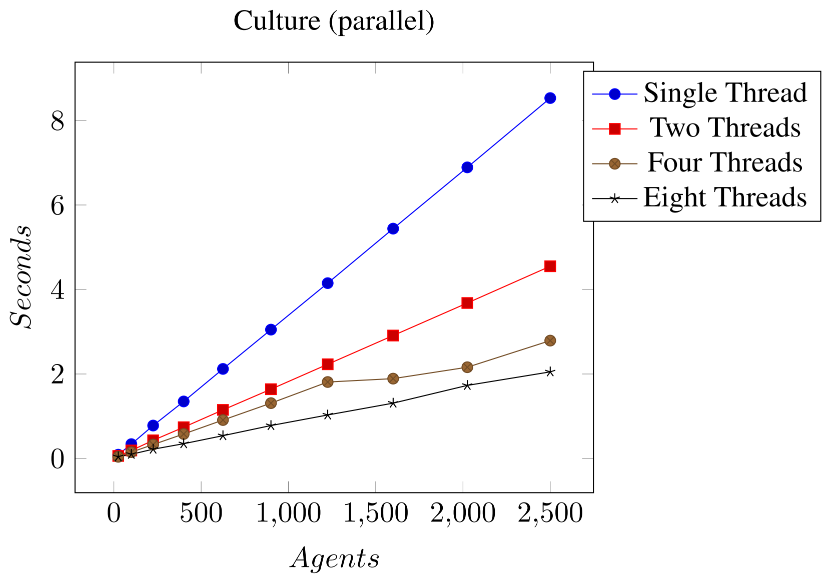


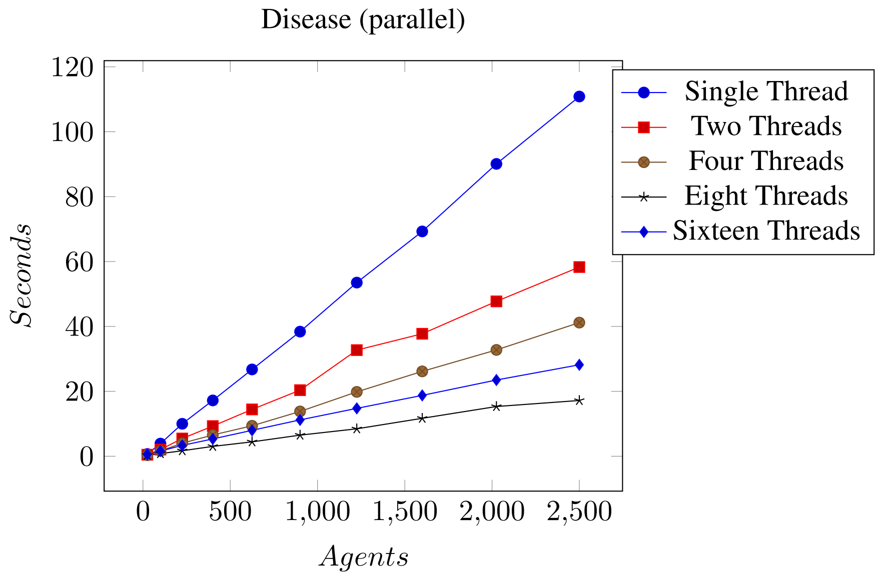
A.2 Concurrency Graphs

The graphs below show the concurrency results obtained from lattice sizes ranging from 10×10 to 100×100 in increments of 10. The results here match those of our tests on *efficiency* and *speed-up* for larger lattice sizes given in chapter Six. They are listed here for completeness.

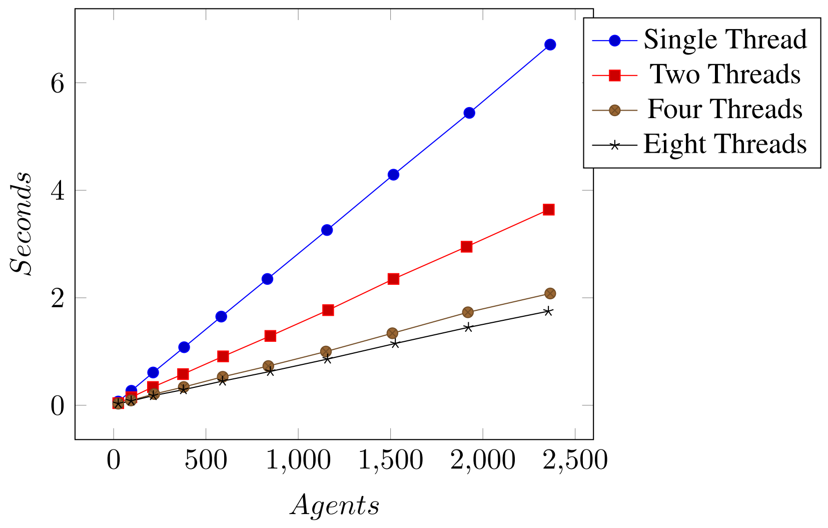
Movement (parallel)



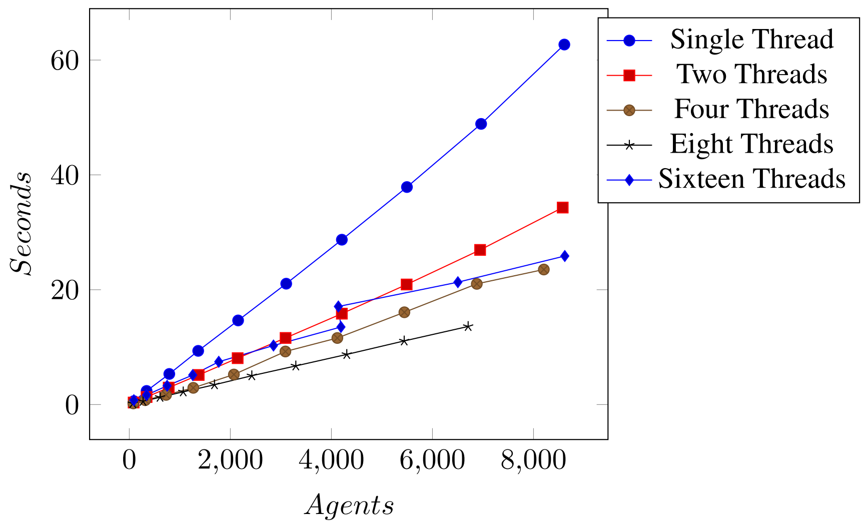

Culture (parallel)
Seconds
Agents
0
2
4
6
8
0
500
1,000
1,500
2,000
2,500
Single Thread
Two Threads
Four Threads
Eight Threads


Disease (parallel)
Seconds
Agents
0
20
40
60
80
100
120
0
500
1,000
1,500
2,000
2,500
Single Thread
Two Threads
Four Threads
Eight Threads
Sixteen Threads

Combat (parallel)



Mating (parallel)



Appendix B

Using the Framework

B.1 Introduction

We will show how different behaviour or interaction types can be created using our framework. The framework is available online for download and is licensed under the GNU GPL licence. We will demonstrate how to create a behaviour of each of the four types and then run the simulation with these behaviours.

B.2 Independent Behaviour Creation

Independent behaviours are the simplest possible agent behaviour. Every Sugarscape simulation contains at least one independent behaviour, usually *Growback*. We will show how to create this behaviour in the framework. First, we create a class (*Growback.h*) to contain our behaviour. This class must inherit from the *IndependentAction* class. It only needs to contain two member functions, a constructor and an *executeAction* function.

```

class Growback:public IndependentAction{
public:
    Growback(World *s, Strategy *theStrategy);  //Always pass World pointer into constructor
    virtual bool executeAction(Location *, group *);
};
```

Listing B.1: Growback.h

All we need to do now is implement these functions in our *Growback.cpp* file.

```


Growback::Growback(World *sim, Strategy *theStrategy):
    IndependentAction(sim,theStrategy){
    //nothing to do here
}

bool Growback::executeAction(Location * place, group*){
    place->setSugar(place->getSugar()+sim->getSugarGrowth());
    return true;
}
```

Listing B.2: Growback.cpp

The constructor just passes the World pointer back to the base class constructor. The action itself is very simple. The location that we are applying the action to is passed in as the *Location**. The second parameter contains the group of agents/locations we interact with to execute this action. In the case of an *Independent* action this is always empty (a *nullptr*) as these actions do not interact with anyone else. The code itself just updates the Sugar held at the location by the set sugar growth value held in *sim* (the pointer to the World object that contains the entire simulation). We return true to show we were successful. That is all we have to do.

Next we will create a *Read-Dependent* action and see how behaviours are applied to agents instead as well as locations.

B.3 Read-Dependent Behaviour Creation

The Agent Culture rule causes agents to become culturally aligned with their neighbours. It works quite simply as follows: Take each neighbour in turn and flip one of my culture bits (picked at random) to match theirs.

We create an AgentCulture class and this time inherit from *ReadAction*. Inheriting from ReadAction is the only difference between this and our Growback class.

```
class AgentCulture:public ReadAction{
public:
    AgentCulture(World *s, Strategy *theStrategy);
    virtual bool executeAction(Location *, group *);
};
```

Listing B.3: AgentCulture.h

The ".cpp" file is also almost identical. There is a difference in the *execute-Action* in that here we only act on agents not locations. Since we are passed in a location we just check to see if the location contains an agent. If it does then we get the agent and update it; otherwise we do nothing. If an agent is present we return true after applying the behaviour, if there is no agent we return false. The World pointer *sim* allows us to interrogate the world to get information on our surroundings. For full details on the *sim* API check the framework code.

```


AgentCulture::AgentCulture(World *s, Strategy *theStrategy):
    ReadAction(s,theStrategy){
    //nothing to do
}

bool AgentCulture::executeAction(Location *loc, group* grp){
    if (loc->hasAgent()) {
        //get our position on lattice
        std::pair<int,int> ourPosition=loc->getPosition()
        //get all agents within a distance  of 1 of our position
         std::vector<Agent*> neighbours=sim->getNeighbours(
    ourPosition, 1);
         for(auto a:neighbours){//iterate through all neighbours
             std::vector<bool> tags=a->getCulture();//get culture
    tags
             //pick one of their tags at random
             int index=sim->getRnd(0, sim->getCultureCount()-1);
             //make our tag match theirs
             loc->getAgent()->setTag(index,tags[index]);
         }
         return true;
    }else{
         return false;
    }
}
```

Listing B.4: AgentCulture.cpp

B.4 Write-Dependent Behaviour Creation

We will now look at more complex *Write-Dependent* behaviours. The simplest behaviour in this category is movement. The class definition is the same as before except now we must add a *formGroup* function.

```

class AgentBasicMove :public WriteAction{
public:
    AgentBasicMove(World *s, Strategy *theStrategy);
    virtual bool executeAction(Location *, group *);
    virtual group* formGroup(Location *);

};
```

Listing B.5: AgentBasicMove.h

The constructor does not differ from before (except obviously our base class is not *WriteAction*). Taking the other two functions in turn:

formGroup This function forms a group consisting of the agent making the move and the chosen destination location. In itself this is a simple function to write. Ask *sim* for all empty location within our vision. Pick the best one and add it to the group. Rank this group by distance from agent to destination (nearer is better) and return this group. If there is nowhere to move to then return a group containing our current location as destination.

executeAction The chosen group is passed in as a parameter. As before we do a simple check to ensure there is an agent present at this location first. All we have to do if there is an agent present is move the agent to this new location and update sugar levels.

```

AgentBasicMove::AgentBasicMove(World *s, Strategy *theStrategy)
    :WriteAction(s,theStrategy)
{
    //our work is done
}

bool AgentBasicMove::executeAction(Location *loc, group * grp)
{
    if (loc->hasAgent()) {
        Agent* theAgent=loc->getAgent();
        std::pair<int,int> currPosition=theAgent->getPosition();
        std::pair<int,int> newPosition=grp->getMembers()[0]->
    getPosition();
        sim->setAgent(currPosition, nullptr);/*!< remove old
    location ptr to agent */
```

```
        theAgent->setPosition(newPosition);/*!< set new position to new location */
        sim->setAgent(newPosition,theAgent);/*!< add ptr to agent at new location */
        theAgent->incSugar(grp->getMembers()[0]->getSugar());/*!< eat sugar at new location */
        grp->getMembers()[0]->setSugar(0);/*!< sugar at new location now consumed */
        return true;
    }else{
        std::cerr << "exectued basicMove on location with no agent! " << std::endl;
        return false;/*!< no agent present so do nothing */
    }
}

group* AgentBasicMove::formGroup(Location *loc)
{
    group *ourChoice = nullptr;
    if (loc->hasAgent()) {/*!< Agent at this location */
        ourChoice = new group();
        Agent* theAgent=loc->getAgent();
        /*!< find all empty locations */
        std::vector<Location*> possibleDestinations=sim->getEmptyNeighbourhood(theAgent->getPosition(), theAgent->getVision());
        if (possibleDestinations.size()!=0) {/*!< check to see if we can move anywhere */
            int index=pickIndex(possibleDestinations);
            int rank=possibleDestinations[index]->getPosition().first-theAgent->getPosition().first
                    +possibleDestinations[index]->getPosition().second-theAgent->getPosition().second;
            if (rank<0) rank =-rank;
            ourChoice->push_back(possibleDestinations[index]);
            ourChoice->setRank(rank);
            ourChoice->setPrimeMover(loc);
            //one active participant per group - the agent moving
            ourChoice->setActiveParticipants(1);
        }
        else{/*!< nowhere to move so stay here */
            ourChoice->push_back(loc);
            ourChoice->setRank(0);
            ourChoice->setPrimeMover(loc);
            ourChoice->setActiveParticipants(1);
        }

    }
    return ourChoice;/*!< is NOT nullPtr only if we assigned it a
    value earlier */
}
```

Listing B.6: AgentBasicMove.cpp

We do not have to worry about scheduling, concurrency or clashes with other agents. Our algorithm handles all of that transparently.

B.5 Iterative Write-Dependent Behaviour

$Iterative\,Write-Dependent$ Actions are the most complex type of action. They are a special case of $Write - Dependent$ actions with an added proviso that the action is to be applied to more than one grouping in sequence. There are three examples of this type of action in Sugarscape: Credit, Mating and Trade.

Taking Mating as an example we can see that the rule for mating requires that an agent mate with all available partners during a single step. As mating can only be undertaken with one agent at a time these multiple mating events must occur in sequence. That is, the rule defines a sequence (or iteration) of mating events for each agent. Multiple mating events can still happen a the same time but only if there is no overlap between the mating partners involved.

The class definition for AgentMating is the same as before except we add a $participantCount$ function that determines how many agents will participate in this event during a single step.

```
class AgentMating:public IterativeWriteAction{
protected:
    bool suitable(Location*);
public:
    AgentMating(World*,Strategy*);
    virtual bool executeAction(Location *, group *);
    virtual group* formGroup(Location *);
    virtual int participantCount(int,int,int);
};
```

Listing B.7: AgentMating.h

Each function is fairly simple to implement.

participantCount This returns the number of participants that will apply this rule. We only count male[1] agents for efficiency. Note that we only count agents who are not *done*. Because the action is iterated we need to keep track of how many iterations we have done and how many remain. This is implemented both within the $formGroup$ function which marks all neighbours we cannot mate with as *done* and within the execute action which marks each neighbour we mate with as *done*. The iterations stop when all neighbours of each agent are done, that is when $participantCount$ returns 0.

formGroup This function forms a group consisting of the agent we will mate with and the location to place the new child agent at. If no such mate or location exists we return the empty group. For this rule, as mating is asymmetric we only form groups for male agents otherwise each parent would appear twice - once with the male agent as the originator and once with the female agent as the originator.

executeAction The chosen group is passed in as a parameter. As before we do a simple check to ensure there is an agent present at this location first. If the group contains two members then the first is the agent we mate with and the second is the location we can place the offspring at. The offspring inherits characteristics from its parents. If the group does not contain two members then there is no one to mate with.

suitable This is a helper function that returns true if an agent is suitable for mating with. That is, the agent is female and has enough sugar reserves.

```

AgentMating::AgentMating(World *sim, Strategy *theStrategy)
:IterativeWriteAction(sim,theStrategy)
{
    //our work is done
}



bool AgentMating::executeAction(Location *loc, group *grp)
{
    if (loc->hasAgent()) {
```

[1]Only counting female agents would work equally here.

```
        Agent* theMale=loc->getAgent();
        if (grp->getMembers().size()==2) {
            std::pair<int,int> newPosition=grp->getMembers()[1]->getPosition();/*!< second in group is empty location */
            Agent *theFemale=grp->getMembers()[0]->getAgent();/*!< first in group is female agent*/
            Agent *offSpring= new Agent(sim,theMale,theFemale,newPosition);
            sim->setAgent(newPosition, offSpring);/*!< Put offspring in empty location */
            offSpring->incSugar(grp->getMembers()[1]->getSugar());/*!< eat sugar at new location */
            offSpring->incSugar(theMale->getSugar()/2);
            theMale->incSugar(-theMale->getSugar()/2);
            offSpring->incSugar(theFemale->getSugar()/2);
            theFemale->incSugar(-theFemale->getSugar()/2);
            offSpring->makeUnavailable();
            grp->getMembers()[1]->setSugar(0);/*!< sugar at new location now consumed */

            /*!< mark this agent as done, when all agents are done stop iterating*/
            if (loc->getCardinal(0)->getAgent()==theFemale) {
                theMale->markNeighbour(0);
            } else if(loc->getCardinal(1)->getAgent()==theFemale){
                theMale->markNeighbour(1);
            } else if (loc->getCardinal(2)->getAgent()==theFemale) {
                theMale->markNeighbour(2);
            }else if (loc->getCardinal(3)->getAgent()==theFemale){
                theMale->markNeighbour(3);
            }else{
                std::cerr <<"MATING CANNOT FIND FEMALE"<<std::endl;
            }
            return true;
        }else{
            return false;/*!< group has no female or no location so no mating here */
        }
    }else{
        std::cerr << "executed mating on location with no agent! " << std::endl;
        return false;/*!< no agent present so do nothing */
    }

}


group* AgentMating::formGroup(Location *loc)
{
    if (loc->hasAgent()) {
        if (loc->getSex()==Sex::male && loc->getAgent
    ()->getInitialSugar()<loc->getAgent()->getSugar()) {
            Agent *me=loc->getAgent();
            Agent *mate=nullptr;
            group *grp= new group();/*!< create group */
            grp->setRank(1);
            grp->setActiveParticipants(1);
            grp->setPrimeMover(loc);

            for (int i=0; i<4 && mate==nullptr; ++i) {/*!< find
    mate first */

                if (me->getAvail(i) && suitable(loc->getCardinal(i
    ))==true) {
                    mate=loc->getCardinal(i)->getAgent();
                }
                me->markNeighbour(i);
            }
            if (mate==nullptr) {/*!< no mate available return
    empty group */
                return grp;
            }
            std::vector<Location*> destinations=sim->
    getEmptyNeighbourhood(me->getPosition(),1);/*!< get empty
    locations around me for offspring */
            std::vector<Location*> moreDestinations=sim->
    getEmptyNeighbourhood(mate->getPosition(),1);/*!< get more
    empty locations around mate for offspring */
            destinations.insert(destinations.end(),
    moreDestinations.begin(),moreDestinations.end());/*!< combine
    location vectors */
            for (auto it = destinations.begin(); it !=
    destinations.end();) {/*!< remove slots that are taken (done!)
    */
                if ((*it)->isDone()) {
                    it = destinations.erase( it ); // reset
    iterator to a valid value post-erase
                }
                else {
                        ++it;
                }
            }
            if (destinations.size()==0) {/*!< nowhere for
    offspring beside me or mate so return empty group */
                return grp;
            }
            grp->push_back(mate->getLocation());/*!< if I get here
     then all is fine create group */
            grp->push_back(destinations[0]);
            return grp;
        }
    }
    return nullptr;//no male agent here so do nothing

}


int AgentMating::participantCount(int startX, int startY, int
    dimSize)
{
    int pcount=0;
    Agent *theAgent=nullptr;
    for (int i=startX; i<startX+dimSize; ++i) {
        for (int k=startY; k<startY+dimSize; ++k) {
            theAgent=sim->getAgent(std::pair<int,int>(i, k));
            if (theAgent!=nullptr) {
                if (theAgent->getSex()==Sex::male && theAgent->
    allDone()==false){
                    ++pcount;
                }
            }
        }
    }
    return pcount;
}

bool AgentMating::suitable(Location *loc){
    //DO WE NEED TO ADD CHECK FOR RESERVES OF SUGAR AS WELL?
    return loc->hasAgent() && loc->getAgent()->getSex() == Sex::
    female
```

```
            && loc->getAgent()->getSugar()>loc->getAgent()->
    getInitialSugar() ? true : false;
}
```

Listing B.8: AgentMating.cpp

B.6 Adding Behaviours to a Simulation

Once we have created our behaviours applying them to a simulation is easy. We first create a world of some size (say a 50×50 lattice). The world is initialised and the dual states are synced together. We create one action object for each behaviour we want to run in the simulation and add these rules to the world. Then to take a step just call *applyRules*.

Of course we would need to add code to output each step to the screen as well but this is separate to the implementation of the synchronous behaviours.

```
        //Initialise World 50 by 50 in size
        World theWorld(40);
        theWorld.init();
        theWorld.sync();
        /*!< Declare all possible strategies here */
        Strategy baseStrategy(theWorld);
        NewSweepStrategy newSweep(theWorld);
        LineByLineStrategy lineByLine(theWorld);
        RndAsyncStrategy rndAsync(theWorld);
        IndependentStrategy independent(theWorld);
        IterativeWriteStrategy iterativeWrite(theWorld);
        ReadDependentStrategy readDependent(theWorld);
        WriteStrategy writeDependent(theWorld);
        /*!< create behaviour objects */
        Growback growback(&theWorld,&independent);
        AgentBasicMove move(&theWorld,&writeDependent);
        AgentCulture agentCulture(&theWorld,&readDependent);
        //Add the rules we are using to the world
        theWorld.addRule(&growback);
        theWorld.addRule(&move);
        theWorld.addRule(&agentCulture);
        /*!< run the simulation */
        for (int k=0; k<100; ++k) {//do 100 steps
            theWorld.applyRules();
        }
```

Listing B.9: main.cpp

B.7 Changing Updating Algorithms

There are three AU algorithms predefined as subclasses of $Strategy$.

LineByLineStrategy The locations in the lattice representing the simulation space are updated in the order they appear in the lattice (usually left to right, top-down);

RndAsyncStrategy The order that is used is determined randomly at the start of the simulation and this order is used for every step in the simulation;

NewSweepStrategy The order that the agents are updated in is determined randomly at the start of each step (each step uses a different order).

Changing to one of these strategies is achieved using the *setStrategy* function call.

```
        // pass in new strategy to rules
       growback.setStrategy(&lineByLine);
       move.setStrategy(&lineByLine);
       agentCulture.setStrategy(&lineByLine);
```

Listing B.10: main.cpp

Appendix C

The Formal Specification of Sugarscape

C.1 Introduction

C.1.1 Agent Based social Simulations

Sugarscape was the first large scale Agent Based Social Simulation (ABSS). It was developed by Epstein and Axtell and presented in their book *Growing Artificial Societies* [Epstein and Axtell, 1996]. The release of this simulation is considered an important event in the emerging field of Agent Based Social Simulation.

The Sugarscape ABSS was used to investigate how individual behaviour can influence and cause different social dynamics within large populations. It has been used to show how, for example, inheritance of wealth affects resource distribution in populations and how disease can spread through a population. It remains influential today and every major simulation toolkit (Swarm, Repast, Mason and NetLogo) [Railsback et al., 2006a, Berryman, 2008, Inchiosa and Parker, 2002] comes with a partial implementation of Sugarscape that demonstrates that toolkit's approach to simulation. Since Sugarscape first appeared ABSSs have been applied to fields as diverse as Anthropology[Campillo et al., 2012], Biomedical Science, Ecology, Social Science [Axtell and Axtell, 2000], Epidemic modelling and Market Analysis[Macal and North, 2009, Troitzsch, 2009, Gilbert, 2004].

ABSS's employs a bottom-up approach to modelling populations. Instead of precomputing the overall population behaviour, as done in equation based models, individual agents and their local interactions within the population are modelled. The behaviour of the overall population is left to emerge from these local interactions. This approach allows us to address failings in the top-down approach and demonstrates the causal factors behind the emergence of group dynamics. In cases where we do not know what the overall behaviour will be or where we are trying to find out the causes of this behaviour, bottom-up based ABSSs are the only possible approach.

C.1.2 Issues with Sugarscape

Currently, social science simulations are starting to embrace concurrency in an effort to allow for bigger, more complete and faster implementations of ABMs. Different concurrency researchers have used the Sugarscape model as a testbed for benchmarking different approaches to parallelising ABMs [Perumalla, 2006, Richmond et al., 2010]. However although the rules of Sugarscape have been defined there is no general agreement on their exact meaning. These difficulties hamper the ability of researchers both to properly compare their approaches, provide complete implementations of Sugarscape or replicate their results.

Most of the rules require some form of conflict resolution. We have specified the rules in a manner consistent with the original intention (agents acting concurrently) but independent of any particular approach to how this concurrency is implemented. That is, we have refrained from imposing any specific conflict resolution rules.

By formalising Sugarscape and providing a single precisely defined reference for the rules we can produce a standard definition of Sugarscape. Compliance with this single reference will allow proper comparisons to be made between different approaches. It also leaves it open to the implementer to decide what approach to conflict resolution they wish to take. We detected ambiguities present in the current rule definitions, provided precise interpretations, where possible, and flagged irresolvable problems where not.

We made the decision to restrict the initial specification to one pollution type and one resource type in an effort to guarantee clarity. While the rules were designed so that they could be extended to arbitrary numbers of resources and pollutants, explicitly specifying for an arbitrary number of resources and pollutants would make the specification even more difficult to understand and thus more likely to either contain or cause mistakes.

Once we had a specification for the single resource scenario we extended the specification to a two resource situation. This allowed us to specify the final rule, $Trade$, as that rule requires two resources to function.

This allowed for:

1. A simpler and easier to understand specification of the rules that use only one resource (trading clarity against completeness);
2. A complete (but separate) specification for simulations that use two resources.

We do not provide specifications for multiple pollutants as multiple pollutants were never actually implemented in Sugarscape[1].

Similarly we did not provide a specification for more than two resources as we deem the benefits of doing so counterbalanced by both the complexity of the resulting specification and the lack of any requirement to use such a complex simulation for benchmarking purposes. Sugarscape has only ever been implemented with two resources types, known respectively as *sugar* and *spice*. Anyone wishing to extend Sugarscape further can use the two resource specification for guidance.

C.1.3 Synchronous and Asynchronous Updating

Originally the rules were stated with an explicit assumption that the underlying implementation would be sequential. Concurrency was simulated through randomisation of the order of each rule application on the individual agents, and models that follow this regime are termed asynchronous.

> All results reported here have been produced by running the model on a serial computer; therefore only one agent is“active” at any instant. In principle the model could be run on parallel hardware, permitting agents to move simultaneously (although **M** would have to be supplemented with a conflict resolution rule to handle cases in which two or more agents simultaneously decide to inhabit the same site).
> [Footnote 12, Chapter II]

The alternative to asynchronous updating is synchronous updating. Synchronous updating assumes that all updates occur concurrently. While it is clear that the original authors have no objection to employing synchronous updating on sugarscape it is well known that asynchronous and synchronous updating produce different results. What is not known is how divergent these results are in the case of complex ABMs such as Sugarscape r indeed how to apply synchronous updating to all the complex interaction types in Sugarscape.

[1]We leave this open as an exercise for the reader.

In order to answer these questions we present initially a specification that assumes a synchronous updating regime, as this is the most novel approach. Following this we give the equivalent Asynchronous updating version for comparison.

C.2 Single Resource Sugarscape

Sugarscape is a discrete turn based simulation composed of a set of interacting agents that move across an environment. The environment, or simulation space, is modelled as a two dimensional M by M grid or matrix of discrete locations known as the *lattice*. This lattice is toroidal in nature, that is, it wraps around on all four edges. Every lattice location has a position denoted by its x and y coordinates. For any lattice location [i, j] there are four direct (von Neumann) neighbours (up, down, left and right) at positions $[i, (j + 1)\%M]$, $[i, (j - 1)\%M]$, $[(i - 1)\%M, j]$ and $[(i + 1)\%M, j]$. We denote this set of von Neumann neighbours as $N_1(i, j)$, and further use $N_k(i, j)$ to denote the set of von Neumann neighbours where each is a maximum distance of k locations from the location [i,j].

Each location can hold a number of resource and pollution types. While there is no limit placed on how many resource or pollution types can exist in a Sugarscape simulation we are unaware of any Sugarscape derived simulation that uses more than two resource types and one pollution type. When there is only one resource type it is called *sugar* and if there are two then the second resource is known as *spice*. These amounts are measured as natural numbers ($\geq$ 0). Each individual location has limits placed on the maximum amount of resources of any type it may carry at any one time. These limits are defined at simulation startup and remain fixed during a simulation run. Agents consume the resources at their current location. Locations replenish their resources by some defined amount during each time step. Each location can also hold at most one agent at a time.

Agents reside at locations within the lattice but are mobile and can change location at most once per step. At a minimum each agent has the following attributes:

Metabolism Rate (one per resource type) The rate at which an agents resource stores decrease during each simulation step. Different resource types have independent metabolism rates. Once an agent runs out of resources it *dies* (is removed from the simulation);

Age The number of steps that the agent has been present in the simulation;

Maximum Age The maximum number of steps that an agent is allowed to exist during the simulation run. Once an agent reaches its maximum age it is removed from the simulation;

Resource Store (one for each resource type) The amount of each resource that the agent currently has;

Vision How far in each of the cardinal directions that the agent can see. An agent can only interact with locations and agents that are in its neighbourhood N_{vision}. To ensure locality all agent values for vision will be less than some predefined maximum and this maximum will be much smaller than the lattice dimension size (M).

In the more complex versions of Sugarscape agents can also have a "culture" identifier (identifying which tribe the agent belongs to), a set of outstanding loans of resources that the agent has given to (or received from) other agents, a set of diseases that the agent has contracted and, an immunity system that gives each agent immunity from certain diseases.

A simulation run consist of a series of turns or steps during which certain rules are applied to each location and agent. Each rule is applied concurrently and instantaneously to each agent and/or location. The rules are generally fairly simple and the only information that an agent (or location) can use when deciding how to apply a rule is local information, that is an agent or location at position [i,j] can only access information from locations and/or agents that are within the set N_k, where k$\leq$ vision (in most cases k=1).

The rules for locations decide how resources are replenished and how pollution is created or spread. The rules for agents are more varied and determine agent movement and interaction. Agent interaction can range from spreading disease, trading, entering financial agreements and even combat. There are a large number of rules but not all rules need to be (or indeed can be) applied in the same simulation run. The rules are chosen based on what we wish to model. A simulation that wishes to see the effect of trading on wealth distribution would have no need for the combat or culture rules while one modelling disease transmission would only require the movement and disease transmission rules.

C.2.1 Basic Types and Constants

First we identify the basic types and any required constants. Many are self explanatory or will become clear when their associated rules are specified. A simulation is defined by the values given to these constants and the combination of rules employed.

$$
\begin{array}{|ll}
M : \mathbb{N}_1 & (1) \\
CULTURECOUNT : \mathbb{N}_1 & (2) \\
MAXVISION : \mathbb{N}_1 & (3) \\
MINMETABOLISM, MAXMETABOLISM : \mathbb{N} & (4) \\
SUGARGROWTH : \mathbb{N}_1 & (5) \\
MAXAGE, MINAGE : \mathbb{N}_1 & (6) \\
MAXSUGAR : \mathbb{N}_1 & (7) \\
DURATION : \mathbb{N}_1 & (8a) \\
RATE : \mathbb{A} & (8b) \\
INITIALSUGARMIN, INITIALSUGARMAX : \mathbb{N} & (9) \\
WINTERRATE, SEASONLENGTH : \mathbb{N}_1 & (10) \\
PRODUCTION, CONSUMPTION : \mathbb{N} & (11) \\
COMBATLIMIT : \mathbb{N} & (12) \\
IMMUNITYLENGTH : \mathbb{N} & (13) \\
INITIALPOPULATIONSIZE : \mathbb{N} & (14) \\
POLLUTIONRATE : \mathbb{N} & (15) \\
CHILDAMT : \mathbb{N} & (16) \\
\hline
CULTURECOUNT \bmod 2 = 1 & \\
MINMETABOLISM < MAXMETABOLISM & \\
MAXAGE < MINAGE & \\
MAXVISION < M & \\
INITIALSUGARMIN < INITIALSUGARMAX & \\
INITIALPOPULATIONSIZE \leq M * M &
\end{array}
$$

1. The simulation space is represented by a two dimensional M by M matrix of locations. Each location in the simulation space is referenced by two indices representing its position in this matrix;

2. $CULTURECOUNT$ determines the size of the bit sequence used to rep-

resent cultural allegiances. This is always equal to an odd number so that the number of 1's in the sequence is never equal to the number of 0's;

3. Agents can only "see" in the four cardinal directions, that is the locations to the north, south, east and west. Agents are endowed with a random vision strength that indicates how many locations the can "see" in each direction. This endowment is always less than $MAXVISION$ and $MAXVISION$ is always less than M;

4. Agents consume an amount of sugar (resources) during each turn. This sugar represents the amount of energy required to live. Each agent is endowed, on creation, with a random metabolism between $MINMETABOLISM$ and $MAXMETABOLISM$;

5. Agents consume sugar (resources) from the location they occupy. Each location can renew its sugar at a rate determined by $SUGARGROWTH.$ After each turn up to a maximum of $SUGARGROWTH$ units of sugar are added to each location (in accordance with the $Growback$ rule);

6. $MAXAGE$ and $MINAGE$ are, respectively, the maximum and minimum allowable lifespan for any agent;

7. $MAXSUGAR$ is the maximum amount of sugar that any location can possibly hold. This is known as the carrying capacity of a location;

8. $RATE$ and $DURATION$ are used for determining the rate of interest charged for loans and the duration of a loan;

9. $INITIALSUGARMIN$ and $INITIALSUGARMAX$ are the lower and upper limits for initial endowment of sugar given to a newly created agent;

10. If seasons are enabled then two seasons, winter and summer are allowed with a duration of $SEASONLENGTH$ turns (ticks) and a new separate lower seasonal grow back rate calculated using $WINTERRATE$ (as determined by the $SeasonalGrowback$ rule);

11. Pollution can occur at a rate determined by the production and consumption of resources determined by the $PRODUCTION$ and $CONSUMPTION$ constants respectively;

12. The combat rule posits the maximum reward $COMBATLIMIT$ that can be given to an agent through killing another agent;

13. Immunity in agents is represented using a fixed size sequence of bits of length $IMMUNITYLENGTH$;

14. We have some predetermined initial population size $INITIALPOPULATIONSIZE$ that is used to initialise the simulation;

15. $POLLUTIONRATE$ determines the number of steps that elapse before pollution levels diffuse to their neighbours;

16. A certain amount of sugar reserves, $CHILDAMT$, are required for an agent to have children.

$$[AGENT] \tag{1}$$
$$POSITION == 0\,..\,M-1 \times 0\,..\,M-1 \tag{2}$$
$$SEX ::= male \mid female \tag{3}$$
$$BIT ::= 0 \mid 1 \tag{4}$$
$$affiliation ::= red \mid blue \tag{5}$$
$$boolean ::= true \mid false$$

1. $AGENT$ is used as a unique identifier for agents;

2. $POSITION$ is also used to make specifying indices within the grid so as to make the schemas easier to read and more compact;

3. All agents have a sex attribute;

4. BITs are used to encode both culture preferences and diseases of agents;

5. Every agent has a cultural affiliation of either belonging to the blue tribe or red tribe.

Agents can, using the Mating rule, have offspring if they are fertile. Fertility is determined by the age of the agent, where fertility starts at some predefined age and ends at another. These boundaries are defined for all agents. The numbers are set out by Epstein and Axtelland although there appears to be no special significance attached to these numbers we will stick with the originals. Male fertility ends 10 turns later than female fertility.

$$
\begin{array}{|l}
FEMALEFERTILITYSTART, FEMALEFERTILITYEND : \mathbb{N} \\
MALEFERTILITYSTART, MALEFERTILITYEND : \mathbb{N} \\
\hline
12 \leq FEMALEFERTILITYSTART \leq 15 \\
40 \leq FEMALEFERTILITYEND \leq 50 \\
12 \leq MALEFERTILITYSTART \leq 15 \\
50 \leq MALEFERTILITYEND \leq 60 \\
MALEFERTILITYEND = FEMALEFERTILITYEND + 10
\end{array}
$$

The replacement rule requires a sugar allocation be given to new agents set between 5 and 25. Again there appears to be no special significance attached to these numbers.

$$
\begin{array}{|l}
STARTSUGARMIN, STARTSUGARMAX : \mathbb{N} \\
\hline
STARTSUGARMIN = 5 \\
STARTSUGARMAX = 25
\end{array}
$$

C.2.2 The Sugarspace Lattice

The simulation space in Sugarscape consists of a finite discrete two-dimensional array of locations. Each location is identified its row and column value. Each location contains a number of resources. While only two resources are ever used it is clear that the intention of the original authors was that the simulation could be extended so that any number of different resources can be present.

Similarly each location can contain a number of pollutant levels. In practice, although the rule is explicitly defined for an arbitrary number of pollution types only one is ever used. Again, in line with actual Sugarscape usage and to make the specification more readable we assume only one pollution type. Pollution fluxes are used in the rules to help calculate how pollution levels change over time. Although explicitly referenced in the Pollution rule these do not need to be explicitly modelled in the specification.

Lattice

$$\begin{array}{lr}
sugar : POSITION \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} & (1) \\
maxSugar : POSITION \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} & (2) \\
pollution : POSITION \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} & (3) \\
\hline
\operatorname{dom} sugar = \operatorname{dom} maxSugar = \operatorname{dom} pollution = POSITION & (4) \\
\forall\, x : POSITION \bullet sugar(x) \leq maxSugar(x) \leq MAXSUGAR & (5)
\end{array}$$

1. $sugar$ is a mapping that stores the amount of sugar stored at each position;
2. $maxSugar$ is a mapping that records the maximum amount of sugar that can be stored (carried) in each position;
3. $pollution$ records the amount of pollution at each location;
4. Every position has a sugar level, a maximum allowed sugar level (or carrying load) and a pollution level;
5. Every position's sugar level is less than or equal to the maximum allowed amount for that position which is in turn less than or equal to the $MAXSUGAR$ constant;

We need to track the number of turns that have occurred in the simulation. Each turn consists of the application of all rules that form part of the simulation.

Step

$$step : \mathbb{N}$$

C.2.3 Agents

Every agent is situated on a location within the grid and each location is capable of containing only one agent at a time (putting an upper limit on the number of possible agents). Agents are mobile, that is they can move to a new location if a suitable unoccupied location is available. Movement is both discrete and instantaneous, it is possible for an agent to move to a new location instantly while skipping over all intermediate locations. The attributes that every agent has are:

Vision How far in the four cardinal directions that an agent can see;

Age Number of turns of the simulation that an agent has been alive;

Maximum Age Age at which an agent dies;

Sex Agents are either male or female;

Sugar Level The amount of sugar that an agent currently holds. There is no limit to how much sugar an agent can hold;

Initial Sugar The amount of sugar the agent was initialised with on creation;

Metabolism The amount of energy, defined by sugar (or resource) consumption, used during every turn of the simulation;

Culture Tags A sequence of bits that represents the culture of an agent;

Children For each agent we track its children (if any). To apply the Inheritance rule the full list of an agents children is required.

Loans Under the credit rule agents are allowed lend and/or borrow sugar for set durations and interest rates so we need to track these loans. For each loan we need to know the lender, the borrower, the loan principal and the due date (represented as the step number);

Diseases Diseases are sequences of bits that can be passed between agents. An agent may carry more than one disease;

Immunity Each agent has an associated bit sequence that confers immunity against certain diseases. If the bit sequence representing a disease is a subsequence of an agents immunity bit sequence then that agent is considered immune to that disease.

$$
\begin{array}{l}
\textit{Agents} \\
\hline
population : \mathbb{P}\, AGENT \\
position : AGENT \rightarrowtail POSITION \\
sex : AGENT \nrightarrow SEX \\
vision : AGENT \nrightarrow \mathbb{N}_1 \\
age : AGENT \nrightarrow \mathbb{N} \\
maxAge : AGENT \nrightarrow \mathbb{N}_1 \\
metabolism : AGENT \nrightarrow \mathbb{N} \\
agentSugar : AGENT \nrightarrow \mathbb{N} \\
initialSugar : AGENT \nrightarrow \mathbb{N} \\
agentCulture : AGENT \nrightarrow \operatorname{seq} BIT \\
children : AGENT \nrightarrow \mathbb{P}\, AGENT \\
loanBook : AGENT \leftrightarrow (AGENT \times (\mathbb{N}, \mathbb{N})) \\
agentImmunity : AGENT \nrightarrow \operatorname{seq} BIT \\
diseases : AGENT \nrightarrow \mathbb{P} \operatorname{seq} BIT \\
\hline
population = \\
\qquad \operatorname{dom} position = \operatorname{dom} sex = \operatorname{dom} vision = \operatorname{dom} agentImmunity \\
\qquad = \operatorname{dom} maxAge = \operatorname{dom} agentSugar = \operatorname{dom} children = \operatorname{dom} diseases \\
\qquad = \operatorname{dom} agentCulture = \operatorname{dom} metabolism = \operatorname{dom} age \qquad (1) \\
\operatorname{dom} loanBook \subseteq population \qquad (2) \\
\operatorname{dom}(\operatorname{ran} loanBook) \subseteq population \qquad (3) \\
\forall\, x, y : AGENT;\ d : \operatorname{seq} BIT \bullet \\
x, y \in population \land x \neq y \Rightarrow \qquad (4) \\
\qquad ((age(x) \leq maxAge(x) \land MINAGE \leq maxAge(x) \leq MAXAGE \\
\qquad \land \#\, agentCulture(x) = CULTURECOUNT \\
\qquad \land \#\, agentImmunity(x) = IMMUNITYLENGTH \\
\qquad \land vision(x) \leq MAXVISION \\
\qquad \land MINMETABOLISM \leq metabolism(x) \leq MAXMETABOLISM \\
\qquad \land position(x) = position(y) \Leftrightarrow x = y) \\
d \in \operatorname{ran} diseases(x) \Rightarrow \#\, d < IMUNITYLENGTH \\
\hline
\end{array}
$$

1. Every existing agent has an associated age, sex, vision, etc. Note that the population holds only the currently existing agent IDs;

2. Only current members of the population can be lenders;

3. Only current members of the population can be borrowers

4. Every agent in the population is guaranteed to have a current age less than the maximum allowed age for that agent, a maximum age less than or equal to the global $MAXAGE$, a metabolism between the allowed limits and vision less than or equal to the maximum vision. The sequence of bits representing its culture tags is $CULTURECOUNT$ in size while those representing immunity is $IMMUNITYLENGTH$ in size. All diseases are represented by sequences of bits that are shorter than the immunity sequence.

The entire simulation consists of locations, agents and a counter holding the tick count. We combine them all in the schema $SugarScape$.

$SugarScape$

$Agents$
$Lattice$
$Step$

The initial state of the schema when the simulation begins must also be stated.

$InitialSugarScape$

$Sugarscape'$

$$step' = 0 \quad (1)$$
$$\# population' = INITIALPOPULATIONSIZE \quad (2)$$
$$loanBook' = \varnothing \quad (3)$$
$$\begin{aligned}&\forall a : AGENT \bullet \quad (4)\\ &a \in population' \Rightarrow\\ &(age(a) = 0 \wedge diseases'(a) = \varnothing \wedge children'(a) = \varnothing\\ &\wedge INITIALSUGARMIN \leq agentSugar'(a) \leq INITIALSUGARMAX)\\ &\wedge initialSugar'(a) = agentSugar'(a)\end{aligned}$$

1. *step* is set to zero;

2. The population is set to some initial size;

3. There are no loans as yet;

4. Every agent in the starting population has an age of zero, no diseases or children and some initial sugar level within the agreed limits. The other attributes have random values restricted only by the invariants;

C.2.4 Rules

There are a number of rules that can be employed in different combinations to give different simulations. We will quote each rule as laid out in the appendix of [Epstein and Axtell, 1996] and follow, where necessary, with a more detailed explanation of the rule. In many cases the simple rule definitions are not complete. Extra information, embedded in the original text, has been extracted where necessary to help complete these rules. The majority of rule definitions assume only one resource (sugar) and it is these that are specified in this section.

The simulation is discrete with each time interval representing one complete set of rule applications. We use the $step$ variable in the $SugarScape$ schema to keep track of the current time interval number.

Where there exist ambiguities in the rule definitions we will identify them and propose one or more possible interpretations consistent with what we believe to be the authors intentions. Throughout the rule definitions constants such as α, β are used but they have different meanings in each rule. For the sake of clarity we will give each constant a meaningful and globally unique name.

C.2.5 Tracking Steps

While not defined explicitly as a rule, we must ensure that we record the current step number. We increment the $Step$ variable before every sequence of rule applications that compose a single turn of the simulation.

There is an issue with metabolism in that every turn of the simulation requires that agents use up their sugar reserves at a rate determined by their metabolism. It is not explicitly stated when or where this sugar deduction occurs within the rules. It could be placed, for example, in the movement rule but it can also be placed, just as validly, within any rule that is guaranteed to be applied during every turn. Since there is no obvious reason why one is superior to the other, as long as it is consistently applied, we choose to place the metabolism deduction within the $Tick$ schema. This new rule can be stated simply as follows:

Tick At the start of every time interval increase every agents age by one and decrease every agents sugar level by their metabolism rate.

$$
\begin{array}{l}
\textit{Tick} \\
\hline
\Delta Agents \\
\Delta Step \\
\hline
population' = population \\
position' = position \\
sex' = sex \\
vision' = vision \\
maxAge' = maxAge \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
agentCulture' = agentCulture \\
children' = children \\
loanBook' = loanBook \\
agentImmunity' = agentImmunity \\
diseases' = diseases \\
step' = step + 1 \qquad (1) \\
\forall x : AGENT \bullet x \in population \Rightarrow \\
\quad age'(x) = age(x) + 1 \qquad (2) \\
\quad \wedge\ agentSugar'(x) = agentSugar(x) - metabolism(x) (3) \\
\hline
\end{array}
$$

1. Add one to the step count;

2. Increase everyone's age by one;

3. Decrease everyone's agentSugar by their metabolism.

C.2.6 Sugarscape Growback$_\alpha$

Sugarscape Growback$_\alpha$ At each Lattice position, sugar grows back at a rate of α units per time interval up to the capacity at that position.

Growback determines the rate at which location resources are replenished. The integer constant α indicates the amount by which resources grow during a single step or time interval. If $\alpha = \infty$ then each resource returns to its maximum value during each turn, i.e. it is instantly fully replenished after each step. The rule only refers to a single resource, *sugar*, but the book explicitly defines one other resource *spice* and it is clear that generalisations allowing an arbitrary number of resource types to be held at each Lattice position are acceptable.

Since we are dealing only with one resource, sugar, we only need to define α for this resource . The constant $SUGARGROWTH$ represents $alpha$ in this rule and we use this to update the sugar level of each position.

Since the maximum carrying level of each resource cannot be exceeded we will set the resource levels to its maximum value if application of the replenishment rate would result in a value greater than this maximum. With these definitions we can express the $Growback$ rule in a simple manner. The last line in the schema (see below) does the work of updating the resource levels of every location.

$Growback$

$\Delta Lattice$

$$pollution' = pollution$$
$$maxSugar' = maxSugar$$
$$sugar' = \{x : POSITION \bullet$$
$$x \mapsto min(\{sugar(x) + SUGARGROWTH, maxSugar(x)\})\}(1)$$

1. The new sugar levels are calculated using a simple formula to either, the maximum possible level for that location or the old level plus the $SUGARGROWTH$ whichever is the smaller.

C.2.7 Seasonal Growback $S_{\alpha,\beta,\gamma}$

Seasonal Growback $S_{\alpha,\beta,\gamma}$ Initially it is *summer* in the top, half of the Sugarscape and *winter* in the bottom half. Then every γ time periods the seasons flip - in the region where it was summer it becomes winter and vice versa. For each site, if the season is summer then sugar grows back at a rate of α units per time interval; if the season is winter then the grow back rate is α units per β time intervals.

Seasonal growback is an alternative to the previous grow back rule. Which rule is chosen will depend on what the simulation is trying to demonstrate. Seasonal grow back allow us to introduce seasonal factors into the original $Growback$ rule. There are two seasons (representing summer and winter) and each lasts γ turns before switching. We rename γ to $SEASONLENGTH$. α is the summer season $SUGARGROWTH$ rate and α/β is the winter season rate. We use the existing $SUGARGROWTH$ to hold the summer rate and introduce $WINTERRATE$ as β.

Determining what season it is during a turn is fairly trivial. When $seasonLength$ divides into the $Step$ variable evenly it is summer in the top half and winter in the bottom half (and vice versa).

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\textit{SeasonalGrowback}} \\
\hline
\Delta Lattice & \\
\Xi Step & \\
\hline
pollution' = pollution & \\
maxSugar' = maxSugar & \\
\forall x : POSITION \bullet & \\
(step \operatorname{div} SEASONLENGTH) \operatorname{mod} 2 = 0 \Rightarrow sugar' = & (1) \\
\quad \{x : POSITION \mid first(x) < M \operatorname{div} 2 \bullet & \\
\quad x \mapsto min(\{sugar(x) & \\
\qquad + SUGARGROWTH, maxSugar(x)\})\} & (1a) \\
\quad \cup & \\
\quad \{x : POSITION \mid first(x) \geq M \operatorname{div} 2 \bullet & (1b) \\
\quad x \mapsto min(\{sugar(x) & \\
\qquad + SUGARGROWTH \operatorname{div} WINTERRATE, maxSugar(x)\})\} & \\
(step \operatorname{div} SEASONLENGTH) \operatorname{mod} 2 \neq 0 \Rightarrow sugar' = & (2) \\
\quad \{x : POSITION \mid first(x) < M \operatorname{div} 2 \bullet & \\
\quad x \mapsto min(\{sugar(x) & \\
\qquad + SUGARGROWTH \operatorname{div} WINTERRATE, maxSugar(x)\})\} & (2a) \\
\quad \cup & \\
\quad \{x : POSITION; \ y : \mathbb{N} \mid first(x) \geq M \operatorname{div} 2 \bullet & \\
\quad x \mapsto min(\{sugar(x) + SUGARGROWTH, maxSugar(x)\})\} & (2b) \\
\hline
\end{array}
$$

1. If the season is summer then:

   a) Top half of grid is updated as normal;

   b) Bottom half is updated at winter rate.

2. Otherwise if it is winter:

    a) Top half of grid is updated at winter rate;

    b) Bottom half is updated as normal.

C.2.8 Movement - M

Movement - M

- Look out as far as vision permits in each of the four lattice directions, north, south, east and west;
- Considering only unoccupied lattice positions, find the nearest position producing maximum welfare;
- Move to the new position
- Collect all resources at that location

The previous rules affected only the locations but the remaining rules affect agents as well as locations. The Movement rule determines how agents select their next location. There are a number of different versions of this rule. We will specify the simplest rule first as it is the only movement rule explicitly defined in the appendix but we will also specify the other movement rules defined in the text. We add a subscript to the rule title (M_{basic}) to distinguish between the different movement rule specifications.

Not explicitly stated within the rule but stated as a footnote to the rule is the restriction that the order in which the lattice directions are searched should be random. This comes into play when two or more available sites exist with the same welfare score.

This rule does not guarantee that an agent will move to the best location. To see why this is the case consider what happens if two agents both try to move to the same location. Only one can succeed and the other will have to move to a less advantageous location. How we decide which agent succeeds is not defined. We assume that either a conflict resolution or conflict avoidance rule is available to make this decision but it is not stated what this rule should be. The original implementation is sequential with agents assumed to be moving in a random order thus enforcing collision avoidance. No guidance is provided for concurrent implementations.

To help make the specification clear we define some simple helper functions. The distance between two positions is only defined for positions that are directly horizontal or vertical to each other. This function must take into account the torus-like (wrap around) structure of the simulation.

$$
\begin{array}{|l}
distance : POSITION \\
\times POSITION \\
\rightarrow \mathbb{N} \\
\hline
\forall\, x1, x2, y1, y2 : \mathbb{N} \bullet \\
distance((x1, y1), (x1, y2)) = \qquad (1) \\
\quad min(\{\mid y2 - y1 \mid, M - \mid y2 - y1 \mid\}) \\
distance((x1, y1), (x2, y2)) = \qquad (2) \\
\quad min(\{\mid x1 - x2 \mid, M - \mid x1 - x2 \mid\}) \\
distance((x1, y1), (x2, y2)) = \infty \Leftrightarrow \\
\quad x1 \neq x2 \wedge y1 \neq y2 \qquad (3)
\end{array}
$$

1. If two agents are vertically aligned we calculate distance based on the horizontal distance;
2. If two agents are horizontally aligned we calculate distance based on the vertical distance;
3. Otherwise the distance is defined as infinity.

We use this to define the $adjacent$ function that lets us know if two agents are directly beside each other.

$$
\begin{array}{|l}
adjacent : POSITION \\
\times POSITION \\
\rightarrow boolean \\
\hline
\forall\, a, b : POSITION \bullet \\
adjacent(a, b) \Leftrightarrow distance(a, b) = 1
\end{array}
$$

$visibleAgents$ takes an agent, a function mapping agents to positions and the vision range of the agent and returns the set of agents that are within that agent's neighbourhood.

$$
\begin{array}{l}
visibleAgents : AGENT \\
\times (AGENT \rightarrowtail\mapsto POSITION) \\
\times \mathbb{N} \\
\nrightarrow \mathbb{F}\, AGENT \\
\hline
\forall\, agent : AGENT;\ pos : AGENT \rightarrowtail\mapsto POSITION;\ range : \mathbb{N} \bullet \\
visibleAgents(agent, pos, range) = \\
\qquad \{ag : AGENT \mid ag \in \mathrm{dom}\, pos \wedge \\
\qquad 1 \leq distance(pos(ag), pos(agent)) \leq range\}
\end{array}
$$

$$Movement_{basic}$$

$$\Delta SugarScape$$

$$\begin{aligned}
&step' = step\\
&population' = population\\
&maxSugar' = maxSugar\\
&pollution' = pollution\\
&sex' = sex\\
&vision' = vision\\
&age' = age\\
&maxAge' = maxAge\\
&agentCulture' = agentCulture\\
&loanBook' = loanBook\\
&diseases' = diseases\\
&agentImmunity' = agentImmunity\\
&children' = children\\
&metabolism' = metabolism\\
&initialSugar' = initialSugar\\
&\forall\, a : AGENT;\ l : POSITION \bullet\\
&a \in population' \Rightarrow && (1)\\
&\qquad distance(position'(a), position(a)) \leq vision(a)\\
&\\
&(distance(position(a), l) \leq vision(a) \land (l \notin \operatorname{ran} position')) \Rightarrow && (2)\\
&\qquad sugar(l) \leq sugar(position'(a)) && (2a)\\
&\land\ (distance(l, position(a)) < distance(position'(a), position(a))) && (2b)\\
&\qquad \Rightarrow sugar(l) < sugar(position'(a))\\
&\\
&agentSugar' = \{\forall\, a : AGENT \mid a \in population' \bullet\\
&\qquad a \mapsto agentSugar(a) + sugar(position'(a))\} && (3)\\
&sugar' = sugar \oplus \{\forall\, l : POSITION \mid l \in \operatorname{ran} position' \bullet l \mapsto 0\} && (4)
\end{aligned}$$

After the rule is applied the following will be the case for every agent:

1. They will be located within one of the locations in their original neighbourhood (possibly the same position as before);

2. After every agent has moved:

a) There will exist no remaining available locations from the original neighbourhood of an agent that would have given a better welfare score than the location that agent now inhabits (we picked the maximum reward);

b) If there was more than one location with maximum reward then the agent moved to the closest location.

3. Agent sugar levels increase because they consume all the sugar at their new location (even if the new location is the same as their old location);

4. Location sugar levels are set to zero everywhere there is an agent present.

The specification states what is true after the application of the rule but not how we achieve that state. In any implementation some conflict resolution strategy will be needed but in this specification we remain agnostic as to what it should be.

The rule is well stated but requires that we precisely define $welfare$. For a single resource simulation welfare is precisely equal to the amount of sugar available at a location. We will define welfare for multiple resource simulations later.

C.2.9 Pollution Formation $P_{\Pi,\chi}$

Pollution Formation $P_{\alpha,\beta}$ When sugar quantity s is gathered from the Sugarscape, an amount of production pollution is gathered in quantity αs. When sugar amount m is consumed (metabolised), consumption pollution is generated according to βm. The total pollution on a site at time t, p^t, is the sum of the pollution present at the previous time, plus the pollution resulting from production and consumption activities, that is, $p^t = p^{t-1} + \alpha s + \beta m$.

This single resource pollution rule is easiest to understand and the most common form of the pollution rule. When pollution is incorporated into the Sugarscape the movement rule is changed so that the welfare of a location is now defined using the sugar to pollution ratio - the greater the ratio the greater the welfare. This ratio is defined as $sugar/(1 + pollution)$ where the "plus one" prevents division by zero.

As the pollution rule requires that we know both the sugar consumed and sugar metabolised during the last move of an agent to that location it is simpler to incorporate the $PollutionFormation$ rule into the movement rule. The alternative is to track the sugar consumed during each move which would require another attribute defined in the $Agent$ schema.

$$
\begin{array}{l}
\textit{Movement}_{pollution} \\
\hline
\Delta \textit{SugarScape} \\
\hline
step' = step \\
maxSugar' = maxSugar \land sex' = sex \\
population' = population \\
vision' = vision \land age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
loanBook' = loanBook \\
children' = children \\
agentImmunity' = agentImmunity \\
diseases' = diseases \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
\forall a : AGENT;\ l : POSITION \bullet \\
a \in population' \Rightarrow distance(position'(a), position(a)) \leq vision(a) \\
(distance(position(a), l) \leq vision(a) \land (l \notin \operatorname{ran} position')) \\
\qquad \Rightarrow [sugar(l)/(1 + pollution(l))] \\
\qquad\qquad \leq [sugar(position'(a))/(1 + pollution(position'(a)))]\ (1) \\
\land\ (distance(l, position(a)) < distance(position'(a), position(a))) \\
\qquad \Rightarrow sugar(l)/(1 + pollution(l)) \\
\qquad\qquad < sugar(position'(a))/(1 + pollution(position'(a))) \\
sugar' = sugar \oplus \{\forall l : POSITION \mid l \in \operatorname{ran} position' \bullet l \mapsto 0\} \\
agentSugar' = \{\forall a : AGENT \mid a \in population \bullet \\
\qquad a \mapsto agentSugar(a) + sugar(position'(a))\} \\
pollution' = pollution \oplus \{\forall l : POSITION;\ x : AGENT \mid \\
position'(x) = l \bullet \\
\qquad l \mapsto (PRODUCTION * sugar(l) + \\
\qquad\qquad CONSUMPTION * metabolism(x))\} \qquad (2) \\
\hline
\end{array}
$$

1. We use our new formula to calculate the desirability of a location;
2. The new pollution value for any location that an agent is present at is calculated as per rule definition.

The rule as stated in the appendix is the generalised rule defined for an arbitrary number of pollutants and resources. We have specified the simpler version as it is easier to grasp. The more complex version has not been used in any of the Sugarscape simulations. We state the generalised rule below for completeness but do not present a formal specification of it.

Pollution Formation $P_{\Pi,\chi}$ For n resources and m pollutants, when n-dimensional resource vector **r** is gathered from the Sugarscape the m-dimensional pollution production vector **p** is produced according to $\mathbf{p} = \blacksquare\mathbf{r}$, where $\blacksquare$ is an $m \times n$ matrix; when n-dimensional (metabolism) vector **m** is consumed then m-dimensional consumption pollution vector **c** is produced according to $\mathbf{c} = \chi\mathbf{m}$, where χ is an $m \times n$ matrix.

C.2.10 Pollution Diffusion D_α

Pollution Diffusion D_α

- Each α time periods and at each site, compute the pollution flux the average pollution level over all its von Neumann neighbouring sites;
- Each site's flux becomes its new pollution level.

This rule determines how pollution diffuses over grid. Pollution diffusion is calculated every α turns and is computed as the average pollution level of all the locations von Neumann neighbours. We use the constant $POLLUTIONRATE$ in place of $alpha$.

The von Neumann neighbours of a location are those immediately above, below, left and right of the current locations (aka north, south, east and west). We define the four cardinal directions taking into account to fact that the grid wraps around at its edges (i.e. it is a torus).

$$
\begin{array}{|l}
north : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
south : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
east : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
west : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION \\
\hline
\forall x, y : \mathbb{N} \bullet \\
\qquad west((x, y)) = ((x - 1) \bmod M, y) \\
\qquad east((x, y)) = ((x + 1) \bmod M, y) \\
\qquad south((x, y)) = (x, (y - 1) \bmod M) \\
\qquad north((x, y)) = (x, (y + 1) \bmod M)
\end{array}
$$

We use this to define a function that returns true if two agents are von Neumann neighbours. It takes as parameters the two agents and a function that maps each agent onto their location in the simulation.

$$
\begin{array}{|l}
vonNeumanNeighbour : AGENT \times AGENT \times (AGENT \rightarrowtail POSITION) \\
\qquad\qquad \rightarrow boolean \\
\hline
\forall a, b : AGENT;\ position : AGENT \rightarrowtail POSITION \bullet \\
vonNeumanNeighbour(a, b, position) \Leftrightarrow \\
\qquad position(a) = north(position(b)) \\
\qquad \vee\ position(a) = south(position(b)) \\
\qquad \vee\ position(a) = east(position(b)) \\
\qquad \vee\ position(a) = west(position(b))
\end{array}
$$

$$
\begin{array}{|l}
\hline
PollutionDiffusion \\
\Delta Lattice \\
\Xi Step \\
\hline
maxSugar' = maxSugar \\
sugar' = sugar \\
(step \bmod POLLUTIONRATE \neq 0) \Rightarrow pollution' = pollution \\
(step \bmod POLLUTIONRATE = 0) \Rightarrow pollution' = \\
\qquad \{\forall l : POSITION \bullet l \mapsto (pollution(north(l)) + pollution(south(l)) \\
\qquad\qquad + pollution(east(l)) + pollution(west(l))) \operatorname{div} 4\} \\
\hline
\end{array}
$$

The $Pollution\,Diffusion$ Rule can be simplified slightly, by removing the redundant mention of $flux$.

Pollution Diffusion D_α After every α time periods and at each location, the average pollution level over all a site's von Neumann neighbouring locations becomes its new pollution level.

C.2.11 Replacement - $R_{[a,b]}$

Replacement - $R_{[a,b]}$ When an agent dies it is replaced by an agent of age 0 having random genetic attributes, random position on the Sugarscape, random initial endowment, and a maximum age selected from the range [a,b].

The two constants a and b we have defined already as $LOWERAGELIMIT$ and $UPPERAGELIMIT$ and we assume that the range is inclusive. It is not stated whether the new agents immediately consume the resources at the location they are placed in. We assume they do not, but accept that the alternative interpretation is equally valid. Although not part of the rule definition in the appendix it is stated elsewhere in the book that new agents will have initial resource levels set between 5 and 25. We have defined $STARTSUGARMIN$ and $STARTSUGARMAX$ for this purpose.

Although the simulation can be run without employing the replacement rule (in an effort, for example, to determine the total carrying load - maximum tolerable population of agents - of a simulation space) there is no stated separate death rule. We will first add a schema that defines "death" explicitly to ensure consistency.

Death When an agent reaches its maximum allowed age or runs out of resources it is removed from the simulation and all its associated loans (either as borrower or lender) are considered void.

Death

$$\Delta Agents$$

$$\begin{aligned}
&population' = population \setminus \\
&\qquad \{a : AGENT \mid age(a) = maxAge(a) \lor agentSugar(a) = 0\} && (1)\\
&loanBook' = population' \lhd loanBook \rhd && (2)\\
&\quad \{x : AGENT \times (\mathbb{N} \times \mathbb{N}) \mid first(x) \in population'\}\\
&\forall a : AGENT \bullet && (3)\\
&a \in population' \Rightarrow\\
&\quad (sex(a) = sex'(a) \land vision(a) = vision'(a)\\
&\quad \land maxAge(a) = maxAge'(a) \land agentCulture(a) = agentCulture'(a)\\
&\quad \land position(a) = position'(a) \land age(a) = age'(a)\\
&\quad \land agentSugar(a) = agentSugar'(a)\\
&\quad \land metabolism'(a) = metabolism(a)\\
&\quad \land diseases'(a) = diseases(a)\\
&\quad \land agentImmunity'(a) = agentImmunity(a)\\
&\quad \land children'(a) = children(a))\\
&\quad \land initialSugar'(a) = initialSugar(a)
\end{aligned}$$

1. We remove from the population all agents who have reached their maximum age or who have no sugar reserves;

2. We remove all loans owed by or owing to these dying agents;

3. Any agent not being removed still has all attributes completely unchanged.

The replacement rule follows readily from this rule, the only addition being the generation of new agents to replace the agents being removed. In effect we have broken the replacement rule into two parts *Death* followed by *Replacement*; although the *Death* rule may be used in isolation the *Replacement* rule must always be preceded by the application of the *Death* rule.

Replacement

$\Delta Agents$

$$\begin{aligned}
&\#population' = INITIALPOPULATIONSIZE && (1)\\
&loanBook' = loanBook\\
&\forall a : AGENT \bullet\\
&a \in population \Rightarrow && (2)\\
&\qquad (a \in population'\\
&\qquad \wedge sex(a) = sex'(a) \wedge vision(a) = vision'(a)\\
&\qquad \wedge maxAge(a) = maxAge'(a) \wedge agentCulture(a) = agentCulture'(a)\\
&\qquad \wedge position(a) = position'(a) \wedge age(a) = age'(a)\\
&\qquad \wedge agentSugar'(a) = agentSugar(a)\\
&\qquad \wedge metabolism'(a) = metabolism(a)\\
&\qquad \wedge diseases'(a) = diseases(a)\\
&\qquad \wedge agentImmunity'(a) = agentImmunity(a)\\
&\qquad \wedge children'(a) = children(a))\\
&\qquad \wedge initialSugar'(a) = initialSugar(a)\\
&a \in population' \setminus population \Rightarrow\\
&\qquad (age'(a) = 0 && (3)\\
&\qquad \wedge STARTSUGARMIN \leq agentSugar'(a) \leq STARTSUGARMAX\\
&\qquad \wedge initialSugar'(a) = agentSugar'(a)\\
&\qquad \wedge diseases'(a) = \varnothing \wedge children'(a) = \varnothing)
\end{aligned}$$

1. The new population has the correct number of members;

2. The existing agents remain unchanged and part of the new population;

3. All new agents have new values initialised within the allowed limits (those not stated explicitly are random values within the ranges set by the specification invariants.

We do not state the positions of any new agents because they are chosen randomly. Our schema invariants ensure that they are on the grid in a location not occupied by any other agent.

We need to add some extra information to this rule definition to ensure that:

1. Newly created agents have no diseases, children or loans;

2. Their initial endowment of resources is within a set range.

C.2.12 Agent Mating S

Agent Mating S

- Select a neighboring agent at random;
- If the neighboring agent is of the opposite sex and if both agents are fertile and at least one of the agents has an empty neighboring site then a newborn is produced by crossing over the parents' genetic and cultural characteristics;
- Repeat for all neighbors.

This rule determines how mating takes place amongst agents to produce offspring. An agent is fertile if its age is within preset boundaries. This is represented by the simple $isFertile$ function below.

$$isFertile : (\mathbb{N} \times SEX) \rightarrowtail boolean$$

$$\begin{aligned}
&\forall\, age : \mathbb{N} \bullet \\
&isFertile(age, male) \Leftrightarrow \\
&\quad MALEFERTILITYSTART \leq age \leq MALEFERTILITYEND \\
&isFertile(age, female) \Leftrightarrow \\
&\quad FEMALEFERTILITYSTART \leq age \leq FEMALEFERTILITYEND
\end{aligned}$$

We define two functions that take in an agent and a mapping from parents to offspring and returns the father or mother of the agent.

$$\begin{aligned}
&father : AGENT \times ((AGENT \times AGENT) \rightarrowtail\!\!\!\!\rightarrow AGENT) \rightarrowtail\!\!\!\!\rightarrow AGENT \\
&mother : AGENT \times ((AGENT \times AGENT) \rightarrowtail\!\!\!\!\rightarrow AGENT) \rightarrowtail\!\!\!\!\rightarrow AGENT
\end{aligned}$$

$$\begin{aligned}
&\forall\, x, m, f : AGENT;\ Offspring : (AGENT \times AGENT) \rightarrowtail\!\!\!\!\rightarrow AGENT \bullet \\
&father(x, Offspring) = m \Leftrightarrow Offspring((m, f)) = x \\
&mother(x, Offspring) = f \Leftrightarrow Offspring((m, f)) = x
\end{aligned}$$

The issues encountered with the mating rule are similar to those with movement. If two sets of parent try to produce offspring in the same vacant location only one can succeed. As there is no preferred conflict resolution rule we cannot state any preference for which agents succeed in producing children and which do not. All we can state is that the maximum number of offspring will be produced given the space constraints but we cannot always be sure which offspring make it into this set. Neighbours in this rule refers to *von Neumann Neighbours* only.

Mating although proceeding concurrently throughout the population is an exclusive event. That is, if agent A is mating with agent B then A cannot be mating with any other agent at the same time: you can only ate with one partner at a time. The rule itself specifies that each agent will mate with all available partners so the execution of the rule can involve a sequence of mating events for specific agents.

Although it is not stated in the rule definition the accompanying book mentions that each parent should gift half of its sugar to its offspring and will only mate if it has a sugar level equal to or greater than its initial sugar level (that is its sugar level on creation). This significantly complicates the rule and dramatically changes its definition and characteristics. However we will assume that this information was inadvertently omitted from the rule definition as the rule makes more sense if we include these extra factors.

Since each individual agent can involve itself in a sequence of up to four mating events during rule execution we require a specification that retains global concurrency while still imposing a sequential ordering based on these constraints. We do this by collecting all possible potential mating partners into a set and then dividing this set into a sequence of maximally sized sets where each subset contains only mating events that can occur concurrently. These sets are produced using a conflict resolution rule that ensures that only pairing that can occur simultaneously appear within each such subset. The rule then proceeds by executing mating events within each subset concurrently while the sets are evaluated in sequence.

$$
\begin{array}{l}
\textit{AgentMating} \\
\hline
\Xi Lattice \\
\Delta Agents \\
\hline
loanBook' = loanBook \\
\exists potentialMatingPairs : \mathbb{P}(AGENT \times AGENT) \mid \qquad (1) \\
potentialMatingPairs = \{(a : AGENT, b : AGENT) \mid sex(a) \neq sex(b) \\
\quad \land isFertile(age(a), sex(a)) \land isFertile(age(head), sex(head)) \\
\quad \land adjacent(position(a), position(head))\} \\
(population', position', vision', agentSugar', agentCulture', metabolism' \\
\quad , children', diseases', agentImmunity', age', sex', initialSugar') = (2) \\
concurrentMating(getConfictFreePairs(potentialMatingPairs), \\
\quad population, position, vision, \\
\quad agentSugar, agentCulture, metabolism, children, \\
\quad diseases, agentImmunity, age, maxAge, sex, initialSugar)
\end{array}
$$

1. Generate the set of all possible mating pairs;

2. Recursively proceed with concurrent mating within the conflict free subsets.

$$
\begin{array}{l}
getConfictFreePairs : \mathbb{P}(AGENT \times AGENT) \\
\rightarrow \\
\mathrm{seq}(\mathbb{P}(AGENT \times AGENT)) \\
\hline
\forall AllPairs : \mathbb{P}(AGENT \times AGENT);\ a : AGENT; \bullet \\
\exists conflictFreeSet : \mathbb{P}(AGENT \times AGENT) \mid \\
conflictFreeSet \subseteq AllPairs \qquad (1) \\
\land a \in \mathrm{ran}\, conflictFreeSet \Rightarrow a \notin \mathrm{dom}\, conflictFreeSet \\
\land a \in \mathrm{dom}\, conflictFreeSet \Rightarrow a \notin \mathrm{ran}\, conflictFreeSet \\
\forall otherSet : \mathbb{P}(AGENT \times AGENT) \mid \qquad (2) \\
otherSet \subseteq AllPairs \land a \in \mathrm{dom}\, otherSet \Rightarrow a \notin \mathrm{ran}\, AllPairs \\
\land a \in \mathrm{ran}\, otherSet \Rightarrow a \notin \mathrm{dom}\, AllPairs \bullet \#otherSet \leq \#conflistFreeSet \\
getConfictFreePairs(\varnothing) = \varnothing \\
getConfictFreePairs(AllPairs) = \qquad (3) \\
\langle conflictFreeSet \rangle \frown getConfictFreePairs(AllPairs \setminus conflictFreeSet)
\end{array}
$$

1. Generate a collision free (conflict resolved) set where each agent can only once within the set;
2. Ensure this set is as large as possible;
3. Recurse through the remaining pairs dividing them into more conflict free sets.

$$
\begin{array}{l}
concurrentMating : \mathrm{seq}\,\mathbb{P}(AGENT \times AGENT) \\
\times \mathbb{P}\,AGENT \times AGENT \rightarrowtail POSITION \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathrm{seq}\,BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{P}\,AGENT \times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\,BIT \\
\times AGENT \nrightarrow \mathrm{seq}\,BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow SEX \\
\times AGENT \nrightarrow \mathbb{N} \\
\leftrightarrow \\
\mathbb{P}\,AGENT \times AGENT \rightarrowtail POSITION \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathrm{seq}\,BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{P}\,AGENT \times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\,BIT \\
\times AGENT \nrightarrow \mathrm{seq}\,BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow SEX \\
\times AGENT \nrightarrow \mathbb{N} \\
\hline
\forall\, tail : \mathrm{seq}\,\mathbb{P}(AGENT \times AGENT);\ head : \mathbb{P}(AGENT \times AGENT); \\
population : \mathbb{P}\,AGENT;\ position : AGENT \rightarrowtail POSITION; \\
vision : AGENT \nrightarrow \mathbb{N}_1;\ agentSugar : AGENT \nrightarrow \mathbb{N}; \\
agentCulture : AGENT \nrightarrow \mathrm{seq}\,BIT;\ metabolism : AGENT \nrightarrow \mathbb{N}; \\
children : AGENT \nrightarrow \mathbb{P}\,AGENT;\ diseases : AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\,BIT; \\
agentImmunity : AGENT \nrightarrow \mathrm{seq}\,BIT;\ age : AGENT \nrightarrow \mathbb{N}; \\
maxAge : AGENT \nrightarrow \mathbb{N}_1;\ sex : AGENT \nrightarrow SEX; \\
initialSugar : AGENT \nrightarrow \mathbb{N};
\end{array}
$$

$$
\begin{array}{l}
\exists\, newpopulation : \mathbb{P}\, AGENT;\\
newposition : AGENT \rightarrowtail POSITION;\\
newvision : AGENT \twoheadrightarrow \mathbb{N}_1;\\
newagentSugar : AGENT \twoheadrightarrow \mathbb{N};\\
newagentCulture : AGENT \twoheadrightarrow \mathrm{seq}\, BIT;\\
newmetabolism : AGENT \twoheadrightarrow \mathbb{N};\\
newchildren : AGENT \twoheadrightarrow \mathbb{P}\, AGENT;\\
newdiseases : AGENT \twoheadrightarrow \mathbb{P}\,\mathrm{seq}\, BIT;\\
newagentImmunity : AGENT \twoheadrightarrow \mathrm{seq}\, BIT;\\
newage : AGENT \twoheadrightarrow \mathbb{N};\\
newmaxAge : AGENT \twoheadrightarrow \mathbb{N}_1;\\
newsex : AGENT \twoheadrightarrow SEX;\ newinitialSugar : AGENT \twoheadrightarrow \mathbb{N};\ |\\
(newpopulation, newposition, newvision, newagentSugar, newagentCulture,\\
\quad newmetabolism, newchildren, newdiseases,\\
\quad newagentImmunity, newage, newmaxAge,\\
\quad newsex, newinitialSugar) =\\
applyMating(asSeq(head), population, position, vision,\\
\quad agentSugar, agentCulture, metabolism, children,\\
\quad diseases, agentImmunity, age, maxAge, sex, initialSugar) \bullet\\
concurrentMating(\langle\rangle, population, position, vision,\\
\quad agentSugar, agentCulture, metabolism, children,\\
\quad diseases, agentImmunity, age, maxAge, sex, initialSugar) =\\
(population, position, vision, agentSugar, agentCulture, metabolism,\\
\quad children, diseases, agentImmunity, age, maxAge, sex, initialSugar)\\
concurrentMating(\langle head\rangle \frown tail, population, position, vision,\\
\quad agentSugar, agentCulture, metabolism, children,\\
\quad diseases, agentImmunity, age, maxAge, sex, initialSugar) =\\
concurrentMating(tail, newpopulation, newposition, newvision,\\
\quad newagentSugar, newagentCulture, newmetabolism, newchildren,\\
\quad newdiseases, newagentImmunity, newage, newmaxAge,\\
\quad newsex, newinitialSugar)
\end{array}
$$

$$
\begin{array}{l}
applyMating : \mathrm{seq}(AGENT \times AGENT) \\
\times \mathbb{P}\, AGENT \times AGENT \rightarrowtail POSITION \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathrm{seq}\, BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{P}\, AGENT \times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\, BIT \\
\times AGENT \nrightarrow \mathrm{seq}\, BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow SEX \\
\times AGENT \nrightarrow \mathbb{N} \\
\leftrightarrow \\
\mathbb{P}\, AGENT \times AGENT \rightarrowtail POSITION \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathrm{seq}\, BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{P}\, AGENT \times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\, BIT \\
\times AGENT \nrightarrow \mathrm{seq}\, BIT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow SEX \\
\times AGENT \nrightarrow \mathbb{N} \\
\hline
\forall\, population : \mathbb{P}\, AGENT;\ position : AGENT \rightarrowtail POSITION; \\
sex : AGENT \nrightarrow SEX;\ vision : AGENT \nrightarrow \mathbb{N}_1; \\
age : AGENT \nrightarrow \mathbb{N};\ initialSugar : AGENT \nrightarrow \mathbb{N}; \\
maxAge : AGENT \nrightarrow \mathbb{N}_1;\ metabolism : AGENT \nrightarrow \mathbb{N}; \\
agentSugar : AGENT \nrightarrow \mathbb{N};\ agentCulture : AGENT \nrightarrow \mathrm{seq}\, BIT; \\
children : AGENT \nrightarrow \mathbb{P}\, AGENT;\ agentImmunity : AGENT \nrightarrow \mathrm{seq}\, BIT; \\
diseases : AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\, BIT;\ head : AGENT \times AGENT; \\
tail : \mathrm{seq}(AGENT \times AGENT);\ \bullet
\end{array}
$$

$$
\begin{array}{l}
\exists offspring, a, b : AGENT;\ newsex : AGENT \nrightarrow SEX;\\
newvision : AGENT \nrightarrow \mathbb{N}_1;\\
newmetabolism, newagentSugar, newinitialSugar : AGENT \nrightarrow \mathbb{N};\\
newmaxAge : AGENT \nrightarrow \mathbb{N}_1;\ newagentCulture : AGENT \nrightarrow \mathrm{seq}\, BIT;\\
newchildren : AGENT \nrightarrow \mathbb{P}\, AGENT;\\
newagentImmunity : AGENT \nrightarrow \mathrm{seq}\, BIT;\\
inheritedImmunity : \mathrm{seq}\, BIT;\ inheritedCulture : \mathrm{seq}\, BIT;\\
\mid offspring \notin populationa = first(head) \wedge b = second(head)\\
\\
newchildren = children \cup \{offspring \mapsto \varnothing\\
\qquad\qquad\qquad , a \mapsto children(a) \cup \{offspring\},\\
\qquad\qquad\qquad b \mapsto children(b) \cup \{offspring\}\}\\
newsex = sex \cup \{offspring \mapsto male\}\\
\quad \vee\ newsex = sex \cup \{offspring \mapsto female\}\\
newvision = vision \cup \{offspring \mapsto vision(a)\}\\
\quad \vee\ newvision = vision \cup \{offspring \mapsto vision(b)\}\\
newmaxAge = maxAge \cup \{offspring \mapsto maxAge(a)\}\\
\quad \vee\ newmaxAge = maxAge \cup \{offspring \mapsto maxAge(b)\}\\
newmetabolism = metabolism \cup \{offspring \mapsto metabolism(a)\}\\
\quad \vee\ newmetabolism = metabolism \cup \{offspring \mapsto metabolism(b)\}\\
newinitialSugar = initialSugar \oplus\\
\quad \{offspring \mapsto initialSugar(a)/2 + initialSugar(b)/2,\\
\quad a \mapsto initialSugar(a)/2, b \mapsto initialSugar(b)/2\}\\
newagentSugar = agentSugar \cup \{offspring \mapsto initialSugar\}\\
\quad \wedge \forall n : 1 \mathrel{..} IMMUNITYLENGTH \bullet\\
\qquad (inheritedImmunity(n) = agentImmunity(a)(n)\\
\qquad \vee\ inheritedImmunity(n) = agentImmunity(b)(n))\\
newagentImmunity = agentImmunity \cup\\
\qquad \{offspring \mapsto inheritedImmunity\}\\
\quad \wedge \forall n : 1 \mathrel{..} CULTURECOUNT \bullet\\
\qquad (inheritedCulture(n) = agentCulture(a)(n)\\
\qquad \vee\ inheritedCulture(n) = agentCulture(b)(n))\\
newagentCulture : agentCulture \cup \{offspring \mapsto inheritedCulture\}
\end{array}
$$

$$\begin{aligned}
&applyMating(\langle\rangle, population, position, vision, agentSugar, agentCulture,\\
&\quad metabolism, children, diseases, agentImmunity,\\
&\quad age, maxAge, sex, initialSugar) =\\
&(population, position, vision, agentSugar, agentCulture, metabolism,\\
&\quad children, diseases, agentImmunity, age, maxAge, sex, initialSugar)
\end{aligned}$$

$$\begin{aligned}
&applyMating(\langle head\rangle \frown tail, population, position, vision, agentSugar,\\
&\quad agentCulture, metabolism, children, diseases, agentImmunity, age,\\
&\quad maxAge, sex, initialSugar) =\\
&\textbf{if}((\exists\, loc : POSITION \mid (adjacent(loc, position(ag)))\\
&\qquad \lor\ adjacent(loc, position(head)) \land loc \notin \operatorname{dom} position)\\
&\qquad \land\ (agentSugar(head) > initialSugar(head))\\
&\qquad \land\ (agentSugar(ag) > initialSugar(ag)))\\
&\textbf{then} && (2a)\\
&\quad applyMating(tail, population \cup \{offspring\},\\
&\quad position \cup \{offspring \mapsto loc\},\\
&\quad newvision, newagentSugar, newagentCulture, newmetabolism,\\
&\quad newchildren, diseases \cup \{offspring \mapsto \varnothing\}, newagentImmunity,\\
&\quad age \cup \{offspring \mapsto 0\}, newmaxAge, newsex, initialSugar)\\
&\textbf{else} && (2b)\\
&\quad applyMating(tail, population, position, vision, agentSugar,\\
&\qquad agentCulture, metabolism, children, diseases,\\
&\qquad agentImmunity, age, maxAge, sex, initialSugar)
\end{aligned}$$

C.2.13 Agent Inheritance I

Agent Inheritance I When an agent dies its wealth is equally distributed among all its living children.

The rule definition is deceptively simple but some assumptions must be made in order to give it a precise definition. These assumptions are required because of the discrete nature of the simulation. Only living children can inherit from a parent. If a child is alive but scheduled to die at the same time as their parent then (because all agents who are due to die will die simultaneously) this child should not inherit from their parent. If we were to allow them to inherit we would either have to impose an ordering on the allocation of inheritance making the rule more complex or accept than the ordering will sometimes result in part of an inheritance disappearing. This extra complexity brings no real benefit to the simulation so we discount it.

The second assumption is that we allow for rounding errors. Resources ($sugar$) come in discrete amounts so division between children requires integer division. This is also true of division of the loans amongst an agents children. We just accept any rounding errors as part of the discrete nature of the simulation.

Finally we note that inheritance is separate from the actual death or replacement rule, it reallocates the resources of agents due to die but it does not remove those agents from the simulation. We leave that to the actual Replacement or Death rule and assume that one of these rules is applied *after* the inheritance rule. This simplifies the $Inheritance$ schema.

To enable inheritance to handle the loan book (when an agent dies its loans are passed on to its children) we introduce some helper functions. The $asSeq$ function turns a set of items into a sequence of items. It does not specify the ordering in the sequence.

$$[X]$$

$$asSeq : \mathbb{P}\, X \rightarrow \operatorname{seq} X$$

$$\forall\, x : \mathbb{P}\, X;\ y : \operatorname{seq} X \bullet$$
$$asSeq(x) = y \Leftrightarrow (\operatorname{ran} y = x \wedge \#\, y = \#\, x)$$

The second function $disperseLoans$ takes in the loan book, a sequence containing all the dying agents and the children of the agents and produces an updated loan book with the loans of the dying agents now dispersed amongst their children. To do this it employs a third function $oneAgentLoans$ that takes in a single agent (who is marked for removal) the loans (in a sequence) held by that agent and the set containing its children. It outputs a new set of loans generated by dispersing all this agents loans amongst its children. In both cases we use sequences for the parameter we are recursing over as it makes the recursion easier to specify.

$$\begin{array}{|l}
disperseLoans : (\mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \times \operatorname{seq} AGENT \\
\qquad\qquad \times (AGENT \nrightarrow \mathbb{P}\, AGENT)) \\
\rightarrow \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\hline
\forall\, Loans : \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})));\ a : AGENT; \\
\qquad tail : \operatorname{seq} AGENT;\ Children : AGENT \nrightarrow \mathbb{P}\, AGENT \bullet \\
disperseLoans(Loans, \langle\rangle, Children) = Loans \\
\\
disperseLoans(Loans, \langle a \rangle \frown tail, Children) = \\
disperseLoans((\{a\} \mathbin{⩤} Loans) \cup \\
\qquad\qquad oneAgentLoans(a, asSeq(\operatorname{ran}(\{a\} \lhd Loans)), \\
\qquad\qquad Children(a)), tail, Children)
\end{array}$$

$$\begin{array}{|l}
oneAgentLoans : AGENT \times \operatorname{seq}(AGENT \times (\mathbb{N} \times \mathbb{N})) \times \mathbb{P}\, AGENT \\
\qquad \rightarrowtail \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\hline
\forall\, a, borrower, inheritor : AGENT;\ Children : \mathbb{P}\, AGENT; \\
\qquad\qquad amt, dur, newAmt : \mathbb{N}; \\
\qquad\qquad tail : \operatorname{seq}(AGENT \times (\mathbb{N} \times \mathbb{N})) \bullet \\
oneAgentLoans(a, \langle\rangle, Children) = \varnothing \\
\\
oneAgentLoans(a, \langle (borrower, (amt, dur)) \rangle \frown tail, Children) = \\
\qquad \{x : AGENT \mid x \in Children \bullet \\
\qquad\qquad (x, (borrower, (amt \operatorname{div} \# Children, dur)))\} \\
\qquad \cup\, oneAgentLoans(a, tail, Children)
\end{array}$$

The getMother and getFather functions simply take in an agent and the $children$ set and finds the mother (father) of the agent from this set.

$$
\begin{array}{|l}
getMother : AGENT \times (AGENT \pfun \mathbb{P}\, AGENT) \times AGENT \\
\qquad \pfun SEX \pfun AGENT \\
getFather : AGENT \times (AGENT \pfun \mathbb{P}\, AGENT) \times AGENT \\
\qquad \pfun SEX \pfun AGENT \\
\hline
\forall\, child, parent : AGENT;\ children : AGENT \pfun \mathbb{P}\, AGENT \bullet \\
getMother(child, children, sex) = \\
\qquad parent \Leftrightarrow child \in children(parent) \\
\qquad \land\ sex(parent) = female \\
getFather(child, children, sex) = \\
\qquad parent \Leftrightarrow child \in children(parent) \\
\qquad \land\ sex(parent) = male
\end{array}
$$

An agent can inherit from at most two different agents, one male and one female. We use this to facilitate the specification by treating each sex separately.

$Inheritance$

$\Delta Agents$

$$population' = population \wedge sex' = sex$$
$$position' = position \wedge vision' = vision$$
$$age' = age \wedge maxAge' = maxAge$$
$$agentCulture' = agentCulture \wedge children' = children$$
$$metabolism' = metabolism \wedge diseases' = diseases$$
$$agentImmunity' = agentImmunity \wedge initialSugar' = initialSugar$$

$$\exists\, dying : \mathbb{P}\, AGENT;$$
$$inheritFromFemale, inheritFromMale : AGENT \nrightarrow \mathbb{N};$$
$$\forall\, x : AGENT;\ \exists\, p : AGENT \mid x \in population \setminus dying \bullet \quad (1)$$
$$\operatorname{dom} inheritFromFemale = \operatorname{dom} inheritFromMale = population \setminus dying$$
$$dying = \{x : AGENT \mid x \in population \wedge \wedge\, (age(x) = maxAge(x) \vee agentSugar(x) = 0)\}$$
$$getMother(x, children, sex) \notin dying \Rightarrow inheritFromFemale(x) = 0 \quad (1a)$$
$$p = getMother(x, children, sex) \wedge p \in dying \Rightarrow inheritFromFemale(x) = agentSugar(p) \operatorname{div} \#(population \cap children(p) \setminus dying))$$

$$getFather(x, children, sex) \notin dying \Rightarrow inheritFromMale(x) = 0 \quad (1b)$$
$$p = getFather(x, children, sex) \wedge p \in dying \Rightarrow inheritFromMale(x) = agentSugar(p) \operatorname{div} \#(population \cap children(p) \setminus dying))$$
$$x \in dying \Rightarrow agentSugar'(x) = 0 \quad (3)$$
$$x \notin dying \Rightarrow agentSugar'(x) = agentSugar(x) + inheritFromMale(x) + inheritFromFemale(x) \quad (4)$$
$$loanBook' = disperseLoans(loanBook, asSeq(dying), children) \quad (5)$$

1. First we construct the set of dying agents. Then using this set of dying agents we can construct two functions, one mapping amounts inherited from a female parent and one mapping amounts inherited from a male parent. These

sets are then used to update the $sugar$ of each agent;

a) The function giving the amount each inheriting agent gets from its female parent is constructed by finding all healthy agents who have a dying mother and determining their share of their dying mother's resources;

b) The function listing amounts each agent gets from a male parent is constructed in an almost identical manner.

2. If an agent is dying its $sugar$ level is set to zero (because it is being reallocated to its children);

3. Otherwise the agents sugar level is its old level plus whatever it inherits from both dying parents;

4. Finally we update the loanBook using our $disperseLoans$ function.

C.2.14 Agent Culture K

Agent cultural transmission

- Select a neighboring agent at random;
- Select a tag randomly;
- If the neighbor agrees with the agent at that tag position, no change is made; if they disagree, the neighbor's tag is flipped to agree with the agent's tag;
- Repeat for all neighbors.

Group membership Agents are defined to be members of the Blue group when 0s outnumber 1s on their tag strings, and members of the Red group in the opposite case.

Agent Culture K Combination of the "agent cultural transmission" and "agent group membership" rules given immediately above.

Group membership is defined with the assumption that there are always an odd number of tags. $tribe$ returns the affiliation of an agent based on the number of bits of each type in its culture sequence. If the majority of bits in a sequence are 0 then it belongs to the $blue$ tribe, otherwise it belongs to the red tribe. This is used by the culture rule.

$$tribe : \mathrm{seq}\, BIT \rightarrow affiliation$$

$$\begin{array}{l}
\forall\, aSeq : \mathrm{seq}\, BIT \bullet \\
tribe(aSeq) = blue \Leftrightarrow \#(aSeq \rhd \{0\}) > \#(aSeq \rhd \{1\}) \\
tribe(aSeq) = red \Leftrightarrow \#(aSeq \rhd \{0\}) < \#(aSeq \rhd \{1\})
\end{array}$$

$flipTags$ is a recursive function that takes in a culture tag sequence belonging to an agent, a sequence of neighbouring agents and the mapping containing all agent's culture tag sequences. It returns a new tag sequence generated by each neighbouring agent flipping one bit chosen at random of the original agent's tag sequence. It is aided in this by the function $flipBit$ that takes in two bit sequences and returns a new sequence equal to the first bit sequence with one bit changed at random to match the other sequence at that position.

$$flipBit : \mathrm{seq}\, BIT \times \mathrm{seq}\, BIT \leftrightarrow \mathrm{seq}\, BIT$$

$$\begin{array}{l}
\forall\, original, other, new : \mathrm{seq}\, BIT \bullet \\
flipBit(original, other) = new \Leftrightarrow \\
\quad \#\, original = \#\, other = \#\, new \wedge \\
\quad \exists\, i : 0 \mathrel{..} \#\, original \bullet \forall\, j : 0 \mathrel{..} \#\, original \bullet \\
\qquad (i \neq j \Rightarrow new(j) = original(j)) \wedge new(i) = other(i)
\end{array}$$

$$\begin{array}{l}
flipTags : \mathrm{seq}\, BIT \times \mathrm{seq}\, AGENT \times (AGENT \mathrel{+\!\!\!\twoheadrightarrow} \mathrm{seq}\, BIT) \\
\leftrightarrow \mathrm{seq}\, BIT
\end{array}$$

$$\begin{array}{l}
\forall\, aSeq : \mathrm{seq}\, BIT;\ ag : AGENT;\ tail : \mathrm{seq}\, AGENT; \\
\quad culturalResources : AGENT \mathrel{+\!\!\!\twoheadrightarrow} \mathrm{seq}\, BIT \bullet \\
flipTags(aSeq, \langle\rangle, culturalResources) = aSeq \\
flipTags(aSeq, \langle ag \rangle \frown tail, culturalResources) = \\
\quad flipTags(flipBit(aSeq, culturalResources(ag)), \\
\qquad tail, culturalResources)
\end{array}$$

The sequence of neighbours is provided by the $Culture$ scheme which employs the $asSeq$ function to convert a set of neighbours into a sequence.

$$
\begin{array}{l}
\textit{Culture} \\
\hline
\Delta Agents \\
\hline
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentSugar' = agentSugar \\
children' = children \\
loanBook' = loanBook \\
diseases' = diseases \\
metabolism' = metabolism \\
agentImmunity' = agentImmunity \\
initialSugar' = initialSugar \\
\forall a : AGENT \bullet a \in population \Rightarrow agentCulture'(a) = \\
\quad flipTags(agentCulture(a), \qquad (1) \\
\quad\quad asSeq(\{b : AGENT \mid adjacent(position(a), position(b))\}) \\
\quad\quad\quad , agentCulture) \\
\hline
\end{array}
$$

1. For every agent a in the population we allow each other agent that counts a as a neighbour to flip one bit at random of a's culture bit sequence.

C.2.15 Combat C_α

Agent Combat C_α

- Look out as far as vision permits in the four principle lattice directions;
- Throw out all sites occupied by members of the agent's own tribe;
- Throw out all sites occupied by members of different tribes who are wealthier then the agent;
- The reward of each remaining site is given by the resource level at the site plus, if it is occupied, the minimum of α and the occupant's wealth;
- Throw out all sites that are vulnerable to retaliation;

- Select the nearest position having maximum reward and go there;
- Gather the resources at the site plus the minimum of α and the occupants wealth if the site was occupied;
- If the site was occupied then the former occupant is considered "killed" - permanently removed from play.

$reward$ is used by the combat rule and values a position based on its sugar content and the sugar reserves held by any agent at that position. The combat rule is really an extension of the movement rule where we are now allowed to move to locations occupied by other agents under certain predefined conditions.

$$
\begin{array}{l}
reward : POSITION \times (POSITION \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathbb{N}) \\
\quad \times (AGENT \rightarrowtail\!\!\!\!\!\twoheadrightarrow POSITION) \\
\quad \times (AGENT \nrightarrow \mathbb{N}) \times \mathbb{N} \\
\rightarrow \mathbb{N} \\
\hline
\forall\, l : POSITION;\ sugar : POSITION \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathbb{N}; \\
\quad agentSugar : AGENT \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathbb{N}; \\
\quad positions : AGENT \rightarrowtail\!\!\!\!\!\twoheadrightarrow POSITION \bullet \\
\textbf{if}\ l \in \operatorname{ran} positions\ \textbf{then} \\
\quad reward(l, sugar, positions, \\
\qquad agentSugar, COMBATLIMIT) = \\
\qquad sugar(l) + min(\{COMBATLIMIT, \\
\qquad\quad agentSugar(positions^{\sim}(l))\}) \\
\textbf{else} \\
\quad reward(l, sugar, positions, \\
\quad agentSugar, COMBATLIMIT) = \\
\qquad sugar(l)
\end{array}
$$

$availMoves$ returns the set of all safe moves that an agent can make.

$$
\begin{array}{l}
availMoves : AGENT \times (AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION) \times \\
(POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}) \times \\
(AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow \operatorname{seq} BIT) \times \mathbb{N} \\
\quad \nrightarrow \mathbb{P}\, POSITION
\end{array}
$$

$$
\begin{array}{ll}
\forall\, agent : AGENT;\ positions : AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION; & \\
\quad sugar : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N};\ agentSugar : AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}; & \\
\quad vision : \mathbb{N};\ culture : AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow \operatorname{seq} BIT \bullet & \\
availMoves(agent, positions, sugar, agentSugar, culture, vision) = & \\
\{l : POSITION;\ x : AGENT \mid & \\
\qquad distance(l, positions(agent)) \leq vision & (1) \\
\wedge\ positions(x) = l \Rightarrow (agentSugar(x) < agentSugar(agent) & (2) \\
\quad \wedge\ tribe(culture(x)) \neq tribe(culture(agent))) & \\
\wedge\ ((distance(positions(x), l) \leq vision) & (3) \\
\quad \wedge\ tribe(culture(x)) \neq tribe(culture(agent))) \Rightarrow & \\
\qquad agentSugar(x) < agentSugar(agent) & \\
\qquad +reward(l, sugar, positions, & \\
\qquad\quad agentSugar, COMBATLIMIT)) \bullet l\} &
\end{array}
$$

1. Only locations within an agents neighbourhood are considered;
2. If a location is occupied it must be occupied by an agent belonging to a different tribe who has lower sugar levels;
3. We only consider a position already containing an agent from another tribe if there are no other agents from a different tribe within the neighbourhood of that location who are stronger than we will be once we have consumed the resources of the new location (that is agents who may retaliate against us for killing an agent belonging to their own tribe).

We note that the rule as stated means we consider retaliation under all conditions even if we are just moving to an empty location. It is unclear from the definition given as to how exactly we check for retaliation. Do we base our check on agents visible from our current position or from the proposed position. We have assumed that it is based on the proposed position but it could easily be otherwise. We also assume that the range used is based on the vision of the moving agent as this seems logical.

The synchronous version of the combat rule assumes that all combat occurs instantaneously (concurrently). We note that it is simpler to specify in that we just state the before and after states and make no mention of orderings of combat.

Combat

$\Delta SugarScape$

$$step' = step \land maxSugar' = maxSugar \land pollution' = pollution$$

$$loanBook' = population' \lhd loanBook \rhd (population' \lhd (\operatorname{ran} loanBook)) \quad (1)$$

$$population' \subseteq population \quad (2)$$

$$sugar' = sugar \oplus \{p : POSITION \mid p \in \operatorname{ran} position' \bullet p \mapsto 0\} \quad (3)$$

$$\forall ag : AGENT;\ l : POSITION \bullet$$

$$ag \in population' \Rightarrow \quad (4)$$

$$(sex'(ag) = sex(ag)$$
$$\land vision'(ag) = vision(ag) \land age'(ag) = age(ag)$$
$$\land maxAge'(ag) = maxAge(ag) \land children'(ag) = children(ag)$$
$$\land agentCulture'(ag) = agentCulture(ag)$$
$$\land agentImmunity'(ag) = agentImmunity(ag)$$
$$\land metabolism'(ag) = metabolism(ag)$$
$$\land diseases'(ag) = diseases(ag)$$
$$\land initialSugar'(a) = initialSugar(a)$$
$$\land agentSugar'(ag) = agentSugar(ag) + reward(position'(ag), sugar, position, agentSugar, COMBATLIMIT) \quad (5)$$
$$\land position'(ag) \in availMoves(ag, position, sugar, agentSugar, agentCulture, vision(ag))) \quad (6)$$
$$)$$

$$ag \in population \setminus population' \Rightarrow \quad (7)$$
$$\exists x : AGENT \bullet position'(x) = position(ag) \land tribe(culture(x)) \neq tribe(culture(ag))$$

$$(l \in availMoves(ag, position, sugar, agentSugar, agentCulture, vision(ag)) \quad (8)$$
$$\land reward(l, sugar, position, agentSugar, COMBATLIMIT) \geq reward(position'(ag), sugar, position, agentSugar, COMBATLIMIT)$$
$$\land distance(position(ag), l) < distance(position(ag), position'(ag)))$$
$$\Rightarrow \exists x : AGENT \bullet position'^{\sim}(l) = x \land position(x) \neq l$$

1. Every agent that is removed from the simulation is also removed from the $loanBook$;
2. No new agents are introduced;
3. Location sugar levels are updated;
4. Every agent that remains in the population has all its attributes unchanged apart from (possibly) position and sugar;
5. We update the sugar levels of each agent using the reward function;
6. Every agent has moved somewhere within their old neighbourhood;
7. Every agent that is no longer part of the population was removed by combat, that is, there is another agent (the agent that killed them) now situated in their old position;
8. If a location available to an agent and the reward of that location is better or equal to that agent's new position and it was closer than that agents new position to its old position then it must be the case that some other agent has just moved to that location (otherwise we would have moved there);

We have had to make some assumptions here. It is not stated what happens when there are no available moves, for example if all sites are subject to retaliation. We have assumed that a move is preferable to staying still and that the only time that an agent stays in the same position is when there are no available moves. That is, if every site, including our current one, is subject to retaliation then we do not move anywhere. A more complex interpretation would be to for an agent that cannot escape retaliation to attack another agent anyway and hope for the best but purely in the interests of simplicity we have agents remain where they are.

C.2.16 Credit L_{dr}

Credit L_{dr}

- An agent is a potential lender if it is too old to have children, in which case the maximum amount it may lend is one-half of its current wealth;
- An agent is a potential lender if it is of childbearing age and has wealth in excess of the amount necessary to have children, in which case the maximum amount it may lend is the excess wealth;

- An agent is a potential borrower if it is of childbearing age and has insufficient wealth to have a child and has income (resources gathered, minus metabolism, minus other loan obligations) in the present period making it credit-worthy for a loan written at terms specified by the lender;
- If a potential borrower and a potential lender are neighbors then a loan is originated with a duration of d years at the rate of r percent, and the face value of the loan is transferred from the lender to the borrower;
- At the time of the loan due date, if the borrower has sufficient wealth to repay the loan then a transfer from the borrower to the lender is made; else the borrower is required to pay back half of its wealth and a new loan is originated for the remaining sum;
- If the borrower on an active loan dies before the due date then the lender simply takes a loss;
- If the lender on an active loan dies before the due date then the borrower is not required to pay back the loan, unless inheritance rule I is active, in which case the lender's children now become the borrower's creditors.

$totalOwed$ calculates the total amount owed from a given sequence of loans. We have assumed that interest is simple interest and not compound.

$$totalOwed : \mathrm{seq}(AGENT \times (\mathbb{N} \times \mathbb{N})) \rightarrow \mathbb{N}$$

$$\begin{aligned}
&\forall\, a : AGENT;\ amt, dur : \mathbb{N};\ tail : \mathrm{seq}(AGENT \times (\mathbb{N} \times \mathbb{N})) \bullet \\
&totalOwed(\langle\rangle) = 0 \\
&totalOwed(\langle(a, (amt, dur))\rangle \frown tail) = \\
&\qquad (amt + amt * RATE * DURATION) \\
&\qquad +totalOwed(tail)
\end{aligned}$$

$canLend$ and $willBorrow$ are simple rules. The definition of what determines credit-worthiness is missing so we have assumed it means an agent has enough money to pay *all* their outstanding loans.

$$
\begin{array}{|l}
canLend : \mathbb{N} \times SEX \times \mathbb{N} \rightarrow boolean \\
\hline
\forall\, age, sugar : \mathbb{N} \bullet \\
canLend(age, male, sugar) \Leftrightarrow \\
\quad age > MALEFERTILITYEND \\
\quad \lor (MALEFERTILITYSTART \leq age \\
\qquad \leq MALEFERTILITYEND \\
\qquad \land sugar > CHILDAMT) \\
canLend(age, female, sugar) \Leftrightarrow \\
\quad age > FEMALEFERTILITYEND \\
\quad \lor (FEMALEFERTILITYSTART \leq age \\
\qquad \leq FEMALEFERTILITYEND \\
\qquad \land sugar > CHILDAMT)
\end{array}
$$

$$
\begin{array}{|l}
willBorrow : \mathbb{N} \times SEX \times \mathbb{N} \times \mathbb{P}(AGENT \times (\mathbb{N} \times \mathbb{N})) \rightarrow boolean \\
\hline
\forall\, age, sugar : \mathbb{N};\ loans : \mathbb{P}(AGENT \times (\mathbb{N} \times \mathbb{N})) \bullet \\
willBorrow(age, male, sugar, loans) \Leftrightarrow \\
\quad (MALEFERTILITYSTART \leq age \\
\qquad \leq MALEFERTILITYEND \\
\qquad \land sugar < CHILDAMT) \\
\qquad \land sugar > totalOwed(asSeq(loans)) \\
willBorrow(age, female, sugar, loans) \Leftrightarrow \\
\quad (FEMALEFERTILITYSTART \leq age \\
\qquad \leq FEMALEFERTILITYEND \\
\qquad \land sugar < CHILDAMT) \\
\qquad \land sugar > totalOwed(asSeq(loans))
\end{array}
$$

$amtAvail$ depends on whether an agent can still have children. If they are no longer fertile then they can loan out half their available sugar. If that are still fertile then they have to retain enough sugar to have children.

```
$amtAvail : \mathbb{N} \times SEX \times \mathbb{N} \rightarrow \mathbb{N}$
--------------------------------
$\forall\, age, sugar : \mathbb{N} \bullet$
$amtAvail(age, male, sugar) =$
**if**$(age > MALEFERTILITYEND)$**then**
    $sugar$ div 2
**else if**$(isFertile(age, male) \land sugar > CHILDAMT)$**then**
    $sugar - CHILDAMT$
**else**
    $0$
$amtAvail(age, female, sugar) =$
**if**$(age > FEMALEFERTILITYEND)$**then**
    $sugar$ div 2
**else if**$(isFertile(age, female) \land sugar > CHILDAMT)$**then**
    $sugar - CHILDAMT$
**else**
    $0$
```

$amtReq$ is the amount that a lender requires. This is not defined so we can only use a best guess as to what it is. We assume that the amount required is that which gives the borrower enough sugar to have children. This is the simplest sensible definition we can think of.

```
$amtReq : \mathbb{N} \rightarrow \mathbb{N}$
--------------------------------
$\forall\, sugar : \mathbb{N} \bullet$
$amtReq(sugar) = CHILDAMT - sugar$
```

We supply some simple helper functions that extract the borrower and lender from a loanBook entry, calculate the amount due from a loan, the principal and the due date (defined as the $step$ when payment is due).

$$
\begin{array}{|l}
lender : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \rightarrowtail AGENT \\
borrower : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \rightarrowtail AGENT \\
amtDue : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \rightarrowtail \mathbb{N} \\
principal : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \rightarrowtail \mathbb{N} \\
due : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \rightarrowtail \mathbb{N} \\
\hline
\forall\, l, b : AGENT;\ p, d : \mathbb{N} \bullet \\
lender(l, (b, (p, d))) = l \\
borrower(l, (b, (p, d))) = b \\
amtDue(l, (b, (p, d))) = p + p * RATE * DURATION \\
principal(l, (b, (p, d))) = p \\
due(l, (b, (p, d))) = d
\end{array}
$$

Finally, using these functions we can present the $PayLoans$ schema.

$$
\begin{array}{|l}
\hline
PayLoans \\
\hline
\Delta Agents \\
\Xi Step \\
\hline
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
agentImmunity' = agentImmunity \\
children' = children \\
diseases' = diseases \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
\exists\, dueLoans, newLoans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \bullet \\
dueLoans = loanBook \rhd \\
\qquad (\operatorname{ran}(loanBook) \rhd \{a : (\mathbb{N} \times \mathbb{N}) \mid second(a) = step\}) \\
(loanBook', agentSugar') = payExclusiveLoans(\\
\qquad chooseConflictFreeSets(dueLoans), agentSugar, loanBook) \\
\hline
\end{array}
$$

This schema is complicated by the fact that it is possible that an agent has a loan due and cannot pay this loan off. In this case, according to the rule definition, the borrower must pay half of its sugar to the lender and renegotiate another loan to cover the remainder of its debt. Under this rule some issues will arise if the borrower has more than one due loan and cannot pay these loans off. The lender must pay each borrower in sequence the amount of half its sugar. This cannot be performed simultaneously (for example if we owe three loans we cannot give each lender half our sugar as this would mean giving out more sugar than we actually have). In order to remain true to the rule definition we must, when we have more than one loan due, pay each loan in some sequence (defined using a conflict resolution rule e.g. pay biggest loan first). The helper function $chooseConflictFreeLoans$ returns a sequence of groups of loans that are conflict free (i.e. a borrower can only appear once in each group).

The function $payExclusiveLoans$ takes in this sequence of loan sets and processes each set concurrently in the same manner as the $Mating$ rule.

$$chooseConflictFreeLoans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N})))$$
$$\leftrightarrow$$
$$\mathrm{seq}(AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N})))$$

$$\begin{aligned}
&\forall\, a : AGENT;\ dueLoans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \bullet\\
&chooseConflictFreeLoans(\varnothing) = \langle\rangle\\
&chooseConflictFreeLoans(dueLoans) =\\
&\quad \exists\, maxSet : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \mid\\
&\qquad maxSet \subseteq dueLoans\\
&\quad \land \#(\{a\} \lhd (\mathrm{ran}\, dueLoans)) > 0 \Rightarrow\\
&\qquad \#(\{a\} \lhd (\mathrm{ran}\, maxSet)) = 1 \qquad (1)\\
&\quad \langle maxSet\rangle \frown chooseConflictFreeLoans(dueLoans \setminus maxSet)
\end{aligned}$$

1. We choose the largest convict free set possible where a set is deemed conflict free if all borrowers only appear in that set at most once.

$$
\begin{array}{|l}
payExclusiveLoans : \mathrm{seq}(AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\times AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} \\
\times (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\leftrightarrow \\
((AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\times AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N}) \\
\hline
\forall tail : \mathrm{seq}(AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))); \\
head, loanBook : (AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))); \\
agentSugar : AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} \bullet \\
payExclusiveLoans(\langle\rangle, agentSugar, loanBook) = \\
\quad (loanBook, agentSugar) \\
payExclusiveLoans(\langle head \rangle \frown tail, agentSugar, loanBook) = \\
\quad \exists newAgentSugar : AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N}; \\
\quad newLoans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \mid \\
\quad (newLoans, newAgentSugar) = \\
\qquad makePayments(asSeq(head), \varnothing, agentSugar) \bullet \\
\quad payExclusiveLoans(tail, \\
\qquad newAgentSugar, (LoanBook \setminus head) \cup newLoans)
\end{array}
$$

$makePayments$ is a recursive function that goes through a sequence of loans and makes the final payment on each one. It is used in the $PayLoans$ schema where it takes in a sequence of the due loans and the agents current sugar levels and returns a set of renegotiated loans, where payment is unable to be made, and the new agent sugar levels.

$$\begin{array}{l}
makePayments : \mathrm{seq}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})))\times \\
\quad \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \times (AGENT \rightarrow\!\!\!\!\!+ \mathbb{N}) \\
\rightarrow \\
(\mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \times (AGENT \rightarrow\!\!\!\!\!+\!\!\!\!\!+ \mathbb{N}))
\end{array}$$

$$\begin{array}{l}
\forall\, renegotiatedLoans, new : \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))); \\
\quad updatedSugar, agentSugar : AGENT \rightarrow\!\!\!\!\!+ \mathbb{N}; \\
\quad loan : (AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))); \\
\quad tail : \mathrm{seq}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \bullet
\end{array}$$

$$makePayments(\langle\rangle, renegotiatedLoans, updatedSugar) = \tag{1}$$
$$\quad (renegotiatedLoans, updatedSugar)$$
$$makePayments(\langle loan\rangle \frown tail, new, agentSugar) = \tag{2}$$
$$\textbf{if}\ amtDue(loan) \leq agentSugar(borrower(loan))\ \textbf{then} \tag{2a}$$
$$\begin{array}{l}
\quad makePayments(tail, new, agentSugar \\
\qquad \oplus\{lender(loan) \mapsto agentSugar(lender(loan)) \\
\qquad\quad +amtDue(loan), \\
\qquad borrower(loan) \mapsto agentSugar(borrower(loan)) \\
\qquad\quad -amtDue(loan)\})
\end{array}$$
$$\textbf{else} \tag{2b}$$
$$\begin{array}{l}
\quad makePayments(tail, new \cup \{(lender(loan), \\
\quad (borrower(loan), (amtDue(loan) \\
\qquad -agentSugar(borrower(loan))\ \mathrm{div}\ 2, \\
\qquad due(loan) + DURATION)))\}, \\
\quad agentSugar \oplus \{lender(loan) \mapsto agentSugar(lender(loan)) \\
\qquad +agentSugar(borrower(loan))\ \mathrm{div}\ 2, \\
\qquad borrower(loan) \mapsto agentSugar(borrower(loan))\ \mathrm{div}\ 2\})
\end{array}$$

For the final part of the *Credit* rule we need to be able to work out the total owed by an agent over all loans. First we define two helper functions: *sumLoans* and *totalOwed*.

$$sumLoans : \mathrm{seq}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \rightarrowtail \mathbb{N}$$

$$\begin{aligned}
&\forall\, tail : \mathrm{seq}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})));\\
&\qquad top : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \bullet\\
&sumLoans(\langle\rangle) = 0\\
&sumLoans(\langle top \rangle \frown tail) = sumLoans(tail) + amtDue(top)
\end{aligned}$$

$$\begin{aligned}
&totalOwed : AGENT \times (AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \rightarrowtail \mathbb{N}\\
&totalLoaned : AGENT \times (AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \rightarrowtail \mathbb{N}
\end{aligned}$$

$$\begin{aligned}
&\forall\, agent : AGENT;\ loans : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \bullet\\
&totalOwed(agent, loans) =\\
&\qquad sumLoans(asSeq(loans \rhd (\{agent\} \lhd (\mathrm{ran}\, loans))))\\
&totalLoaned(agent, loans) =\\
&\qquad sumLoans(asSeq(\{agent\} \lhd loans))
\end{aligned}$$

$MakeLoans$

$$\begin{array}{l}
\Delta Agents \\
\Xi Step
\end{array}$$

$$\begin{array}{ll}
population' = population \land sex' = sex & \\
position' = position \land vision' = vision & \\
age' = age \land maxAge' = maxAge & \\
initialSugar' = initialSugar & \\
agentCulture' = agentCulture \land agentImmunity' = agentImmunity & \\
diseases' = diseases \land children' = children \land metabolism' = metabolism & \\
\exists\, newLoans : \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))); & \\
\forall\, ag, lender, borrower : AGENT;\ amt, due : \mathbb{N} \bullet & \\
loanBook' = loanBook \cup newLoans & (1) \\
ag \in \operatorname{dom} newLoans \Rightarrow & \\
\qquad agentSugar'(ag) = agentSugar(ag) - totalLoaned(ag, newLoans) & (2a) \\
ag \in \operatorname{dom}(\operatorname{ran} newLoans) \Rightarrow & (2b) \\
\qquad agentSugar'(ag) = agentSugar(ag) + totalOwed(ag, newLoans) & \\
ag \notin \operatorname{dom}(newLoans) \cup \operatorname{dom}(\operatorname{ran} newLoans) \Rightarrow & (2c) \\
\qquad agentSugar'(ag) = agentSugar(ag) & \\
willBorrow(age(ag), sex(ag), agentSugar'(ag), & \\
\qquad \operatorname{ran}(loanBook' \cap \{a : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) & \\
\qquad\qquad\qquad \mid borrower(a) = borrower(loan)\})) \Rightarrow & (2d) \\
\qquad\qquad \neg\, \exists\, ag2 : AGENT \bullet & \\
\qquad\qquad\qquad canLend(age(ag2), sex(ag2), agentSugar'(ag2)) & \\
\qquad\qquad\qquad \land\, adjacent(position(ag2), position(ag)) & \\
totalLoaned(ag, newLoans) \leq amtAvail(age(ag), sex(ag), agentSugar(ag)) & (3) \\
totalOwed(ag, newLoans) \leq amtReq(agentSugar(ag)) & (4) \\
(lender, (borrower, (amt, due))) \in newLoans \Rightarrow & (5) \\
\qquad (canLend(age(lender), sex(lender), agentSugar(lender)) & (5a) \\
\qquad \land\, willBorrow(age(borrower), sex(borrower), agentSugar(borrower), & \\
\qquad\qquad \{borrower\} \lhd (\operatorname{ran} loanBook)) & \\
\qquad \land\, amt \leq min(\{amtAvail(age(lender), sex(lender), & \\
\qquad\qquad\qquad\qquad agentSugar(lender)), & \\
\qquad\qquad\qquad amtReq(agentSugar(borrower))\}) & (5b) \\
\qquad \land\, due = step + DURATION & (5c) \\
\qquad \land\, adjacent(position(lender), position(borrower))) & (5d)
\end{array}$$

1. The new loan book is the old book plus the new loans;
2. The following properties ensure sugar is updated correctly and that the correct amount of borrowing has taken place:
    a) If an agent is a lender then their new sugar levels decrease by the amount the have lent;
    b) If an agent is a borrower then their sugar has increased by the amount they have borrowed;
    c) Any agent that neither borrowed or lent has the same sugar levels as before;
    d) If there remain any agents who still need to borrow then it is because there are no agents in their neighbourhood who are still in a position to borrow.
3. The total amount loaned by any agent is no greater than the amount that agent had available;
4. The total amount borrowed is less than or equal to the amount required by the borrower;
5. Every loan in this set must have the following properties:
    a) The lender must be in a position to lend;
    b) The borrower must need to borrow;
    c) The amount is less than or equal to the minimum of (i) the amount required by the borrower and (ii) the maximum amount available from the lender;
    d) The due date of the loan is set by the $DURATION$ constant;
    e) the borrower and lender must be neighbours.

C.2.17 Agent Disease E

Agent immune response

- If the disease is a substring of the immune system then end (the agent is immune), else (the agent is infected) go to the following step;

- The substring in the agent immune system having the smallest Hamming distance from the disease is selected and the first bit at which it is different from the disease string is changed to match the disease.

Disease transmission For each neighbor, a disease that currently afflicts the agent is selected at random and given to the neighbor.

Agent disease processes E Combination of "agent immune response" and "agent disease transmission" rules given immediately above

$subseq$ is a function for determining whether one sequence is a subsequence of another. $hammingDist$ determines the number of bit differences in two sequences of the same size.

$$
\begin{array}{l}
subseq : \mathrm{seq}\, BIT \times \mathrm{seq}\, BIT \rightarrow boolean \\
\hline
\forall\, mid, aSequence : \mathrm{seq}\, BIT \bullet \\
subseq(mid, aSequence) \Leftrightarrow \\
\quad \exists\, prefix, suffix : \mathrm{seq}\, BIT \bullet prefix \frown mid \frown suffix = aSequence
\end{array}
$$

$$
\begin{array}{l}
hammingDist : \mathrm{seq}\, BIT \times \mathrm{seq}\, BIT \rightarrow \mathbb{N} \\
\hline
\forall\, tail, rest : \mathrm{seq}\, BIT \bullet \\
hammingDist(\langle\rangle, \langle\rangle) = 0 \\
hammingDist(\langle 1\rangle \frown tail, \langle 1\rangle \frown rest) = hammingDist(tail, rest) \\
hammingDist(\langle 0\rangle \frown tail, \langle 0\rangle \frown rest) = hammingDist(tail, rest) \\
hammingDist(\langle 0\rangle \frown tail, \langle 1\rangle \frown rest) = 1 + hammingDist(tail, rest) \\
hammingDist(\langle 1\rangle \frown tail, \langle 0\rangle \frown rest) = 1 + hammingDist(tail, rest)
\end{array}
$$

$applyDiseases$ takes in a bit sequence representing the immunity of an agent and a list of diseases that affect the agent and produces a new immunity bit sequence that is updated by the disease list. More precisely, for every disease not in the immunity sequence a single bit in the closest subsequence that matches the disease is flipped to make the sequence more closely match the disease. It uses another function $processInfection$ to process each disease in the disease set.

$$applyDiseases : \mathrm{seq}\, BIT \times \mathrm{seq}\,\mathrm{seq}\, BIT \rightarrow \mathrm{seq}\, BIT$$

$$\begin{aligned}
&\forall\, I, d : \mathrm{seq}\, BIT;\ tail : \mathrm{seq}\,\mathrm{seq}\, BIT \bullet\\
&applyDiseases(I, \langle\rangle) = I\\
&applyDiseases(I, \langle d \rangle \frown tail) =\\
&\qquad applyDiseases(processInfection(I, d), tail)
\end{aligned}$$

$$processInfection : \mathrm{seq}\, BIT \times \mathrm{seq}\, BIT \rightarrow \mathrm{seq}\, BIT$$

$$\begin{aligned}
&\forall\, I, d : \mathrm{seq}\, BIT \bullet\\
&\textbf{if}\ subseq(d, I)\,\textbf{then}\\
&\qquad processInfection(I, d) = I\\
&\textbf{else}\\
&\qquad \exists\, a, b, c : \mathrm{seq}\, BIT;\ \forall\, x : \mathrm{seq}\, BIT \bullet\\
&\qquad a \frown b \frown c = I\\
&\qquad (\#\, b = \#\, x \wedge subseq(x, I)) \Rightarrow\\
&\qquad\qquad hammingDist(b, d) \leq hammingDist(x, d)\\
&\qquad \exists\, i : 1\,..\,\#\, I \bullet (y(i) \neq d(i) \wedge \forall\, j : \mathbb{N} \bullet j < i \Rightarrow d(j) = y(j))\\
&\qquad processInfection(I, d) = I \oplus \{(i + \#\, a) \mapsto b(i)\}
\end{aligned}$$

$ImmuneResponse$ is the simplest part of this rule to specify. The recursive function $applyDiseases$ does all the work.

$$
\begin{array}{l}
\textit{ImmuneResponse} \\
\hline
\Delta Agents \\
\hline
loanBook' = loanBook \\
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
diseases' = diseases \\
children' = children \\
agentSugar' = agentSugar \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
agentImmunity' = \{a : AGENT \mid a \in population \bullet \\
\quad a \mapsto applyDiseases(agentImmunity(a), asSeq(diseases(a)))\} \\
\forall x : AGENT \bullet x \in population \Rightarrow agentSugar'(x) = agentSugar(x) - \\
\quad \#\{d : \mathrm{seq}\, BIT \mid d \in diseases(a) \land \neg\, subseq(d, agentImmunity(a)\}(1) \\
\hline
\end{array}
$$

Although not stated in the rule definition careful reading of the accompanying text [Epstein and Axtell, 1996] shows that there is a penalty that is applied to each agent carrying diseases that it has no immunity to. The text states that for every disease carried by an agent that it has no immunity to, sugar metabolism is increased by one. So if an agent carried two diseases that it has no immunity to then its metabolism rate increases by two. This extra cost can equally be deducted by the metabolism rule or the disease rule. Purely for the sake of narrative it is placed in the *ImmuneResponse* rule where it is first referenced in the original Sugarscape book. This is implemented by the final two lines (1) of the *ImmuneResponse* schema.

The transmission of diseases is the more complex part of this rule. We use a recursive helper function $newDiseases$ to construct a set of diseases that an agent can catch from its neighbours. It takes the set of neighbours and their current diseases as input and constructs a set of diseases where one disease is chosen from each neighbour.

$$
\begin{array}{|l}
newDiseases : \mathrm{seq}\, AGENT \times (AGENT \mathrel{+\!\!\!\rightarrow} \mathbb{P}(\mathrm{seq}\, BIT)) \rightarrow \mathbb{P}\, \mathrm{seq}\, BIT \\
\hline
\forall\, a : AGENT;\ tail : \mathrm{seq}\, AGENT;\ diseases : AGENT \mathrel{+\!\!\!\rightarrow} \mathbb{P}(\mathrm{seq}\, BIT) \bullet \\
newDiseases(\langle\rangle, diseases) = \varnothing \\
newDiseases(\langle a \rangle \frown tail, diseases) = \\
\quad \textbf{if}\ diseases(a) = \varnothing \textbf{then} \\
\quad\quad newDiseases(tail, diseases) \\
\quad \textbf{else} \\
\quad\quad \exists\, d : \mathrm{seq}\, BIT \mid d \in diseases(a) \bullet \\
\quad\quad\quad \{d\} \cup newDiseases(tail, diseases))
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{Transmission} \\
\hline
\Delta Agents \\
\hline
loanBook' = loanBook \\
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
agentImmunity' = agentImmunity \\
children' = children \\
agentSugar' = agentSugar \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
\forall\, a : AGENT \bullet a \in population \Rightarrow \\
\quad diseases'(a) = diseases(a) \cup \qquad\qquad (1) \\
\quad\quad newDiseases(asSeq(visibleAgents(a, position, 1)), \\
\quad\quad\quad diseases) \\
\hline
\end{array}
$$

1. $visibleAgents$ returns the set of neighbours of an agent and this set is then passed to the $newDisease$ function which returns a set of diseases, one chosen from each agent in the neighbour set.

C.2.18 Rule Application Sequence

Each tick of simulation time consists of the application of a set sequence of rules. Not all rules can be used together so we identify the allowable sequences of rules. We note that it is not stated in the book what order the rules are to be applied. In the absence of this information we will pick one ordering and restrict ourselves to this ordering.

The display the different allowable combinations of rules in any given simulation we use the following terminology.

$\{Rule\}$ The indicates that $Rule$ is optional. We can choose to include it or not in a simulation;

$RuleA \mid RuleB$ This indicates that there is a choice of which rule to apply - either one or the other but not both.

This rule ordering is for simulations using only a single resource and so omits the $Trade$ rule.

$$\begin{array}{l}
Tick \\
[\fatsemi Growback \mid\fatsemi SeasonalGrowback] \\
[\fatsemi Movement_{basic} \mid (\fatsemi Movement_{pollution} \fatsemi PollutionDiffusion) \mid\fatsemi Combat] \\
\{\fatsemi\ Inheritance\}\{\fatsemi Death\{[\fatsemi Replacement \mid\fatsemi AgentMating]\}\} \\
\{\fatsemi Culture\}\{\fatsemi PayLoans \fatsemi MakeLoans\} \\
\{\fatsemi Transmission \fatsemi ImmuneResponse\}
\end{array}$$

C.3 Asynchronous Sugarscape Specification

AU is the sequential application of rules to agents during a simulation step. If, for example, all agents move during a single step then a sequential ordering is imposed on all of the agents and they will move one at a time (that is, sequentially) based on that ordering. This is in contrast to SU where all agents will attempt to move simultaneously (concurrently). AU is easier to implement that SU as it maps directly onto the current standard sequential programming practice. AU requires no collision detection and resolution (as for example when two agents try to simultaneously move to the same location) because concurrency is excluded - only one agent can move at any one time. It is well know that the AU and SU approaches can deliver different simulation results.

Although AU and SU are both commonly used in CA based simulations, where agent interactions are simple in nature, AU is prevelent in ABM. This is due to the lack of any good SU algorithms that can handle the complex interactions such as $Movement$, $Combat$ or $Trade$ that can appear in ABM based simulations.

We have provided a specification of Sugarscape that assumes SU. For the sake of completeness and to allow us to make comparisons between synchronous and asynchronous updating in Sugarscape we will now present an AU based specification of the rules of Sugarscape.

C.3.1 Variants of Asynchronous Updating

There are a number of varieties of AU [Schönfisch and de Roos, 1999]. These variations differ in how they sequentially order agents for updating. The best known variations are:

Fixed Direction Line-By-Line The locations in the lattice representing the simulation space are updated in the order they appear in the lattice (usually left to right, top-down);

Fixed Random Sweep The order that is used is determined randomly at the start of the simulation and this order is used for every step in the simulation;

Random New Sweep The order that the agents are updated in is determined randomly at the start of each step (each step uses a different order);

Uniform Choice Each agent has an equal probability of being chosen. If there are n agents, then n agents are chosen randomly during a step. During any single step an agent may not be picked at all or may be picked more than once (in contrast Random New Sweep guarantees every agent is picked exactly once per step);

Exponential Waiting Time This is a *Time Driven* method, all the others are *step driven*. Every agent has its own clock which rings when the agent is to be updated. The waiting times for the clock are exponentially distributed (with mean 1). The probability that an event occurs at time t follows e^{-t} where t is a real number, $t \geq 0$. This is most similar to *Uniform Choice*.

We will provide a specification for each variation in turn.

Fixed Direction Line-by-Line takes in a set of agents and their positions on the lattice. It produces a sequence of agents where every agent appears once and only once in the sequence and the order of the sequence is determined by the agents position on the lattice.

$$
\begin{array}{|l}
lineByLine : AGENT \rightarrowtail POSITION \\
\rightarrow seqAGENT \\
\hline
\forall thePositions : AGENT \rightarrowtail POSITION; \\
\quad theSequence : \mathrm{seq}\, AGENT \bullet \\
lineByLine(theSet) = theSequence \\
\quad \Leftrightarrow \mathrm{ran}\, theSequence = \mathrm{dom}\, thePositions \wedge \\
\qquad \# theSequence = \# thePositions \qquad (1) \\
(n, a) \in theSequence \Leftrightarrow n = first(thePositions(a)) * DIM \\
\quad + second(thePositions(a)) \qquad (2)
\end{array}
$$

1. Each agent in the population appears in the sequence once and only once;
2. If one agent appears before another in the sequence then it also appears before that agent on the lattice.

Fixed Random Sweep returns a sequence of the agents in some fixed random ordering. This random ordering is chosen once at the start of the simulation and is fixed for the entire simulation run.

$$
\begin{array}{|l}
RANDOMORDER : \mathrm{seq}\, POSITION \\
\hline
\# RANDOMORDER = \# POSITION \qquad (1) \\
\forall n, m : \mathbb{N} \bullet RANDOMORDER(n) = RANDOMORDER(m) \quad (2) \\
\quad \Leftrightarrow n = m
\end{array}
$$

1. $RANDOMORDER$ is a globally defined sequence containing an ordering of positions on the lattice;
2. Each position on the lattice appears once and only once in this sequence.

Any ordering that satisfies these constrains is allowable according to our specification. This introduces the randomness into the sequence.

$$
\begin{array}{l}
fixedRandom : AGENT \rightarrowtail POSITION \\
\leftrightarrow seqAGENT \\
\hline
\forall\, thePositions : AGENT \rightarrowtail POSITION; \\
\quad theSequence : \mathrm{seq}\, AGENT \bullet \\
fixedRandom(thePositions) = theSequence \\
\quad \Leftrightarrow \mathrm{ran}\, theSequence = \mathrm{dom}\, thePositions \land \\
\qquad \#\, theSequence = \#\, thePositions \qquad (1) \\
\forall\, i : 0 \,..\, \#\, theSequence - 2;\ a_1, a_2 : AGENT \bullet \\
\quad (i, a_1) \in theSequence \land (i + 1, a_1) \in theSequence \Rightarrow \\
\quad (\exists\, x_1, x_2 : \mathbb{N} \mid (x_1, a_1), (x_2, a_2) \in RANDOMORDER \\
\qquad \land\, x_1 < x_2 \qquad (2)
\end{array}
$$

1. Every agent in the population appears once and only once in the resulting sequence;

2. The ordering of agents in the sequence is based on the ordering defined in $RANDOMORDERING$.

Random New sweep is simpler to specify. We return a random ordering of agents after each call. We only need to ensure that every agent appears in this sequence exactly once.

$$
\begin{array}{l}
rndNewSweep : AGENT \rightarrowtail POSITION \\
\leftrightarrow seqAGENT \\
\hline
\forall\, thePositions : AGENT \rightarrowtail POSITION; \\
\quad theSequence : \mathrm{seq}\, AGENT \bullet \\
rndNewSweep(thePositions) = theSequence \\
\quad \Leftrightarrow \mathrm{ran}\, theSequence = \mathrm{dom}\, thePositions \land \\
\qquad \#\, theSequence = \#\, thePositions\ (1)
\end{array}
$$

1. Every agent in the population appears once and only once in the resulting sequence;

Uniform Choice allows for an agent to be picked multiple times. The only constraints are that the sequence returned contains only agents in the population and that the size of the sequence equals the number of agents.

$$
\begin{array}{|l}
uniformChoice : AGENT \rightarrowtail POSITION \\
\leftrightarrow \mathrm{seq}\, AGENT \\
\hline
\forall thePositions : AGENT \rightarrowtail POSITION;\ n : \mathbb{N}; \\
theSequence : \mathrm{seq}\, AGENT \mid 0 \leq n < \# theSequence \bullet \\
uniformChoice(thePositions) = theSequence \Leftrightarrow \\
\quad theSequence(n) \in \mathrm{dom}\, thePositions \qquad (1) \\
\quad \land \# theSequence = \# thePositions \qquad (2)
\end{array}
$$

1. Every agent in the sequence is an agent from the simulation population;
2. the size of the sequence equals the total number of individual agents in the population.

Each variation of asynchronous updating can now be covered by the simple matter of swapping in the appropriate ordering function within the specifications.

C.3.2 Growback, Seasonal Growback and Replacement

$Replacement$, $Growback$ and $SeasonalGrowback$ belong to the category of rules we term $independent$. This category includes all rules where the agent involved in the update (or rule execution) does not interact with any other agent - the update result is independent of any outside factor. It follows then that the order in which these rules are executed will have no bearing on their outcome. Given this we need make no changes to any of these rules.

C.3.3 Pollution Diffusion

$PollutionDiffusion$ is defined specifically as a synchronous rule. There is no asynchronous alternative to this rule as imposing AU would redefine the rule entirely. For this reason we do not produce a AU specification of this rule.

C.3.4 Movement

The specification of rules under an AU regime follows a standard pattern. First we impose an ordering on all the agents subject to the rule and then we recursively apply the update to each agent in the defined order. Each individual agent update can affect the global state and these changes must be passed forward to the next sequence of agent updates. This is in contrast to SU where all updates occur simultaneously.

We always define the application of the rule to agents in a sequence recursively. While the rules themselves can be quite simple the Z notation forces us to pass to each update all parts of the global state that can be changed. This can result in large function signatures.

$AsyncMovement_{basic}$

$\Delta SugarScape$

$$\begin{aligned}
&step' = step \wedge loc' = loc\\
&maxSugar' = maxSugar\\
&pollution' = pollution\\
&sex' = sex\\
&vision' = vision\\
&age' = age\\
&maxAge' = maxAge\\
&agentCulture' = agentCulture\\
&loanBook' = loanBook\\
&diseases' = diseases\\
&agentImmunity' = agentImmunity\\
&children' = children\\
&metabolism' = metabolism\\
&population' = population\\
&initialSugar' = initialSugar\\
&(sugar', agentSugar', position') =\\
&\quad applyMove(rndNewSweep(position), vision,\\
&\quad sugar, agentSugar, position)
\end{aligned}$$

Movement is a typical example of this structure. The main specification $AsyncMovement_{basic}$

simply passes the relevant state information alongside the ordering of agents (according to whatever AU variant we are using) to the recursive function $applyMove$. This recursive function applies the move rule to each agent in turn and returns the final updated agent position, agent sugar levels and lattice sugar levels.

$$
\begin{aligned}
&applyMove : \operatorname{seq} AGENT \\
&\times AGENT \nrightarrow \mathbb{N} \\
&\times (POSITION ⤔ \mathbb{N}) \\
&\times AGENT ⤔ \mathbb{N} \\
&\times AGENT ⤔ POSITION \\
&\qquad \leftrightarrow \\
&((POSITION ⤔ \mathbb{N}) \\
&\times AGENT ⤔ \mathbb{N} \\
&\times AGENT ⤔ POSITION)
\end{aligned}
$$

$$
\begin{aligned}
&\forall\, head : AGENT;\; tail : \operatorname{seq} AGENT;\; population : \mathbb{P}\, AGENT; \\
&positions : AGENT ⤔ POSITION;\; sugar : POSITION ⤔ \mathbb{N}; \\
&agentSugar : AGENT ⤔ \mathbb{N};\; vision : AGENT \nrightarrow \mathbb{N};\; \bullet \\
&applyMove(\langle\rangle, vision, sugar, agentSugar, positions) = && (1) \\
&\qquad (sugar, agentSugar, positions) \\
& \\
&applyMove(\langle head \rangle \frown tail, vision, sugar, agentSugar, positions) = && (2) \\
&\exists\, newLoc : POSITION \mid \\
&\qquad newLoc \in neighbourhood(position(head), vision(head)) && (3) \\
&\land \forall\, otherLoc : POSITION \mid \\
&\qquad otherLoc \in neighbourhood(position(head), vision(head)) \\
&\qquad \Rightarrow sugar(otherLoc) \leq sugar(newLoc) \bullet \\
&\qquad applyMove(tail, vision, sugar \oplus \{newLoc \mapsto 0\}, \\
&\qquad\qquad agentSugar \oplus \{head \mapsto agentSugar(head) + sugar(newLoc)\}, \\
&\qquad\qquad positions \oplus \{head \mapsto newLoc\})
\end{aligned}
$$

1. The base case. If there are no agents left to update then we simply return the current state;

2. The recursive case. If we have agents left to process then we move the first agent in the list and apply the rule to the remaining agents;

3. Find the best location for the agent to move to based on sugar levels at each location.

C.3.5 Pollution Diffusion

The movement rule for pollution is almost identical to the simpler basic movement rule. It only differs in that it takes pollution into account when selecting the best new position for an agent to move to.

$$
\begin{array}{|l}
\hline
AsyncMovement_{pollution} \\
\Delta SugarScape \\
\hline
step' = step \\
loc' = loc \\
maxSugar' = maxSugar \\
sex' = sex \\
pollution' = pollution \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
loanBook' = loanBook \\
children' = children \\
agentImmunity' = agentImmunity \\
diseases' = diseases \\
metabolism' = metabolism \\
population' = population \\
(sugar', agentSugar', position') = \qquad (1) \\
\quad applyMove_{pollution}(rndNewSweep(position), vision, sugar, \\
\qquad\qquad agentSugar, position, pollution) \\
pollution' = pollution \oplus \\
\quad \{\forall\, l : POSITION;\ x : AGENT \mid position'(x) = l \bullet \qquad (2) \\
\quad l \mapsto (PRODUCTION * sugar(l) \\
\qquad + CONSUMPTION * metabolism(x))\} \\
\hline
\end{array}
$$

1. Call the recursive $applyMove_{pollution}$ to apply movement rule to each agent in turn;

2. Update location pollution levels based on agent movement.

$$
\begin{array}{|l}
applyMove_{pollution} : \operatorname{seq} AGENT \\
\times AGENT \pinj \mathbb{N} \\
\times (POSITION \pinj \mathbb{N}) \\
\times AGENT \pinj \mathbb{N} \\
\times AGENT \pinj POSITION \\
\times POSITION \pinj \mathbb{N} \\
\quad \leftrightarrow \\
((POSITION \pinj \mathbb{N}) \\
\times AGENT \pinj \mathbb{N} \\
\times AGENT \pinj POSITION) \\
\hline
\forall head : AGENT;\ tail : \operatorname{seq} AGENT;\ population : \mathbb{P}\, AGENT; \\
positions : AGENT \pinj POSITION;\ sugar : POSITION \pinj \mathbb{N}; \\
agentSugar : AGENT \pinj \mathbb{N}; \\
vision : AGENT \pfun \mathbb{N};\ pollution : POSITION \pinj \mathbb{N} \bullet \\
applyMove_{pollution}(\langle\rangle, vision, sugar, agentSugar, positions, pollution) = \\
\quad (sugar, agentSugar, positions) \\
\\
applyMove_{pollution}(\langle head \rangle \frown tail, vision, sugar, \\
\quad agentSugar, positions, pollution) = \\
\exists newLoc : POSITION \mid \\
newLoc \in neighbourhood(position(head), vision(head)) \\
\land \forall otherLoc : POSITION \mid \\
otherLoc \in neighbourhood(position(head), vision(head)) \\
\quad \Rightarrow sugar(otherLoc) \operatorname{div} (1 + pollution(otherLoc)) \leq \\
\qquad sugar(newLoc) \operatorname{div} (1 + pollution(position'(newLoc))) \bullet \\
\quad applyMove_{pollution}(tail, vision, sugar \oplus \{newLoc \mapsto 0\}, \\
\qquad agentSugar \oplus \{head \mapsto agentSugar(head) + sugar(newLoc)\}, \\
\qquad positions \oplus \{head \mapsto newLoc\}, pollution)
\end{array}
$$

C.3.6 Combat

Asynchronous Combat is undertaken with the *applyAllCombat* function which applies the combat rule to each agent in a random order using the *singleFight* function. We note in passing that the synchronous specification seems to us to be simpler than the asynchronous one (even if the implementation is not).

$$
\begin{array}{l}
\textbf{Schema } Combat_{async} \\
\Delta SugarScape \\
\hline
step' = step \\
maxSugar' = maxSugar \\
pollution' = pollution \\
loanBook' = population' \lhd loanBook \rhd \\
\qquad (population' \lhd (\operatorname{ran} loanBook)) \\
\\
\forall\, ag : AGENT;\ l : POSITION \bullet \\
ag \in population' \Rightarrow \\
\qquad (sex'(ag) = sex(ag) \\
\qquad \land\ vision'(ag) = vision(ag) \\
\qquad \land\ age'(ag) = age(ag) \\
\qquad \land\ maxAge'(ag) = maxAge(ag) \\
\qquad \land\ children'(ag) = children(ag) \\
\qquad \land\ agentCulture'(ag) = agentCulture(ag) \\
\qquad \land\ agentImmunity'(ag) = agentImmunity(ag) \\
\qquad \land\ metabolism'(ag) = metabolism(ag) \\
\qquad \land\ diseases'(ag) = diseases(ag)) \\
\qquad \land\ initialSugar'(ag) = initialSugar(ag) \\
(population', position', sugar', agentSugar') = \\
\qquad applyAllCombat(rndNewSweep(position), population, position, \\
\qquad\qquad sugar, agentSugar, vision, agentCulture)
\end{array}
$$

$$
\begin{array}{l}
applyAllCombat : \mathrm{seq}\, AGENT \\
\times \mathbb{P}\, AGENT \\
\times (AGENT \pinj POSITION) \\
\times (POSITION \pinj \mathbb{N}) \\
\times (AGENT \pinj \mathbb{N}) \\
\times (AGENT \pfun \mathbb{N}) \\
\times (AGENT \pfun \mathrm{seq}\, BIT) \\
\rightarrow \\
(\mathbb{P}\, AGENT \\
\times (AGENT \pinj POSITION) \\
\times (POSITION \pinj \mathbb{N}) \\
\times (AGENT \pinj \mathbb{N}))
\end{array}
$$

$$
\begin{array}{l}
\forall\, head : AGENT;\ tail : \mathrm{seq}\, AGENT;\ population : \mathbb{P}\, AGENT; \\
positions : AGENT \pinj POSITION;\ sugar : POSITION \pinj \mathbb{N}; \\
agentSugar : AGENT \pinj \mathbb{N}; \\
vision : AGENT \pfun \mathbb{N};\ culture : AGENT \pfun \mathrm{seq}\, BIT \bullet \\
applyAllCombat(\langle\rangle, population, positions, \\
\quad sugar, agentSugar, vision, culture) = \\
\qquad (population, positions, sugar, agentSugar, vision, culture) \\
\\
applyAllCombat(\langle head\rangle \frown tail, population, positions \\
\quad , sugar, agentSugar, vision, culture) = \\
\mathbf{if}(head \in population)\mathbf{then} \\
\quad applyAllCombat(tail, \\
\qquad singleFight(head, population, positions, \\
\qquad\quad sugar, agentSugar, vision, culture)) \\
\mathbf{else} \\
\quad applyAllCombat(tail, population, positions, \\
\qquad\quad sugar, agentSugar, vision, culture)
\end{array}
$$

$$
\begin{array}{l}
singleFight : AGENT \times \mathbb{P}\, AGENT \\
\times (AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION) \\
\times (POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}) \\
\times (AGENT \nrightarrow \mathbb{N}) \times (AGENT \nrightarrow \mathrm{seq}\, BIT) \\
\rightarrow \\
(\mathbb{P}\, AGENT \\
\times (AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION) \times (POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}) \\
\times (AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}))
\end{array}
$$

$$
\begin{array}{l}
\forall\, agent : AGENT;\ population : \mathbb{P}\, AGENT; \\
positions : AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow POSITION; \\
sugar : POSITION \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N} agentSugar : AGENT \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbb{N}; \\
vision : AGENT \nrightarrow \mathbb{N};\ culture : AGENT \nrightarrow \mathrm{seq}\, BIT \bullet \\
singleFight(agent, population, positions, \\
\qquad sugar, agentSugar, vision, culture) = \\
\textbf{if}\ (availMoves(agent, positions, sugar, \\
\qquad agentSugar, culture, vision(agent)) = \varnothing)\ \textbf{then} \\
\qquad (population, positions, sugar, agentSugar) \\
\textbf{else} \\
\qquad \exists\, loc : POSITION;\ available : \mathbb{P}\, POSITION; \\
\qquad \forall\, otherLoc : POSITION \mid \\
\qquad loc, otherLoc \in availMoves(agent, positions, sugar, \\
\qquad\qquad agentSugar, culture, vision(agent)) \\
\qquad \wedge\ otherLocation \neq location \bullet \\
\qquad reward(loc, sugar, position, agentSugar, COMBATLIMIT) \\
\qquad \geq reward(otherLoc, sugar, position, agentSugar, COMBATLIMIT) \\
\qquad (distance(position(agent), loc) > \\
\qquad\qquad distance(position(otherLoc), position'(agent)) \Rightarrow \\
\qquad reward(loc, sugar, position, agentSugar, COMBATLIMIT) \\
\qquad > reward(otherLoc, sugar, position, agentSugar, COMBATLIMIT)) \bullet \\
\qquad (population \setminus \{positions^{\sim}(loc)\}, (positions \rhd \{loc\}) \oplus \{agent \mapsto loc\}, \\
\qquad sugar \oplus \{loc \mapsto 0\}, agentSugar \oplus \{agent \mapsto agentSugar(agent) + \\
\qquad\qquad reward(position'(agent), sugar, position, \\
\qquad\qquad\qquad agentSugar, COMBATLIMIT)\})
\end{array}
$$

C.3.7 Disease

Disease is a simple rule that follows the standard pattern for AU specification. We place all agents into a sequence, ordered according to the variation of AU we are using, and apply the rule to each agent in turn updating the state as we go along.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transmission}} \\
\hline
\Delta Agents \\
\hline
loanBook' = loanBook \\
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
agentImmunity' = agentImmunity \\
children' = children \\
agentSugar' = agentSugar \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
diseases' = \\
\qquad applyTransmission(rndNewSweep(position), \\
\qquad\qquad diseases, position) \qquad (1) \\
\hline
\end{array}
$$

1. Call recursive $applyTransmission$ on each agent in population in determined order.

$$
\begin{array}{|l}
applyTransmission : \mathrm{seq}\, AGENT \\
\times AGENT \nrightarrow \mathbb{P}\, \mathrm{seq}\, BIT \\
\times AGENT \rightarrowtail POSITION \\
\leftrightarrow \\
AGENT \nrightarrow \mathbb{P}\, \mathrm{seq}\, BIT \\
\hline
\forall\, head : AGENT;\ tail : \mathrm{seq}\, AGENT; \\
diseases : AGENT \nrightarrow \mathbb{P}\, \mathrm{seq}\, BIT; \\
position : AGENT \rightarrowtail POSITION \\
\exists\, newInfections : \mathbb{P}\, \mathrm{seq}\, BIT \mid newInfections = \qquad (1) \\
\quad newDiseases(asSeq(visibleAgents(head, position, 1)), diseases) \bullet \\
applyTransmission(\langle\rangle, diseases, position) = \qquad (2) \\
\quad diseases \\
applyTransmission(\langle head \rangle \frown tail, diseases, position) = \qquad (3) \\
\quad applyTransmission(tail, diseases \oplus \\
\qquad \{head \mapsto (diseases(head) \cup newInfections)\}, position)
\end{array}
$$

1. Construct a set of new infections for an agent using the previously defined $newDiseases$;

2. Base case: Noting to do, return new disease mapping;

3. Recursive case: Add new diseases to the first agent in the list (according to the rule definition) and then recursively apply the rule to the rest of the list.

C.3.8 Culture

$Culture$ is specified in an identical manner to $Disease$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AsyncCulture \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta Agents \\
\hline
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentSugar' = agentSugar \\
children' = children \\
loanBook' = loanBook \\
diseases' = diseases \\
metabolism' = metabolism \\
agentImmunity' = agentImmunity \\
initialSugar' = initialSugar \\
agentCulture' = applyCulture(rndNewSweep(position), \\
\qquad agentCulture, position) \\
\hline
\end{array}$$

1. Call recursive *applyCulture* on each agent in population in determined order.

$$
\begin{array}{l}
applyCulture : \mathrm{seq}\, AGENT \\
\times AGENT \pfun \mathrm{seq}\, BIT \\
\times AGENT \rightarrowtail POSITION \\
\leftrightarrow \\
AGENT \pfun \mathrm{seq}\, BIT \\
\hline
\forall\, head : AGENT;\ tail : \mathrm{seq}\, AGENT;\ culture : AGENT \pfun \mathrm{seq}\, BIT; \\
position : AGENT \rightarrowtail POSITION; \\
\exists\, n : \mathbb{P}\, AGENT \mid n = neighbours(head, position(head), 1) \bullet \\
applyCulture(\langle\rangle, culture, position) = \qquad (1) \\
\quad culture \\
applyCulture(\langle head \rangle \frown tail, culture, position) = \qquad (2) \\
\quad applyCulture(tail, \\
\quad culture \oplus \{head \mapsto flipTags(culture(head), asSeq(n), culture)\} \\
\quad , position)
\end{array}
$$

1. Base case: return new values for culture tags;
2. Recursive case: Flip the tags of the first agent in the list and repeat (recursively) for the remaining agents in the list (sequence).

C.3.9 Inheritance

$Inheritance$ also follows the same pattern as $Culture$ and $Disease$.

$AsyncInheritance$

$\Delta Agents$

$$population' = population$$
$$sex' = sex$$
$$position' = position$$
$$vision' = vision$$
$$age' = age$$
$$maxAge' = maxAge$$
$$agentCulture' = agentCulture$$
$$children' = children$$
$$metabolism' = metabolism$$
$$diseases' = diseases$$
$$agentImmunity' = agentImmunity$$
$$initialSugar' = initialSugar$$
$$\begin{aligned}(loanBook', agentSugar') = {} & \\ & applyInheritance(rndNewSweep(position), \\ & \quad children, loanBook, agentSugar)\end{aligned} \qquad (1)$$

1. Use recursive $applyInheritance$ function to calculate inheritance based on each agent in turn.

$$
\begin{array}{l}
applyInheritance : \mathrm{seq}\, AGENT \\
\times AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{P}\, AGENT \\
\times (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\times AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} \\
\times AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} \\
\times AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} \\
\mathrel{+\!\!\!\!\rightarrow} \\
(AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\times AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N}
\end{array}
$$

$$
\begin{array}{ll}
\forall\, head : AGENT;\ tail : \mathrm{seq}\, AGENT; & \\
children : AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{P}\, AGENT; & \\
loans : AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N})); & \\
agentSugar : AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N};\ age, maxAge : AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} & \\
\exists\, newLoans : AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N})); & \\
newAgentSugar : AGENT \mathrel{+\!\!\!\!\rightarrow} \mathbb{N} & \\
\mid newAgentSugar = agentSugar \oplus (\{head \mapsto 0\} \cup & (1) \\
\qquad\qquad \{(a, amt) \mid a \in children(head) \land amt = agentSugar(a) & \\
\qquad\qquad\qquad +agentSugar(head) / \#\, children(head)\}) & \\
\land\ newLoans = (\{head\} \lhd loans) \cup & \\
\qquad oneAgentLoans(a, asSeq(\mathrm{ran}(\{a\} \lhd loans)), & \\
\qquad\qquad\qquad Children(head)) \bullet & (2) \\
applyInheritance(\langle\rangle, children, loans, agentSugar, age, maxAge) = & (3) \\
\qquad (loans, agentSugar) & \\
applyInheritance(\langle head \rangle \frown tail, children, loans, & \\
\qquad agentSugar, age, maxAge) = & \\
\textbf{if}(age(head) = maxAge(head) \lor agentSugar(head) = 0)\ \textbf{then} & (4) \\
\qquad applyInheritance(tail, children, newLoans, & \\
\qquad\qquad newAgentSugar, age, maxAge) & \\
\textbf{else} & \\
\qquad applyInheritance(tail, children, loans, agentSugar, age, maxAge) &
\end{array}
$$

1. Distribute the dying agents sugar equally amongst its children;

2. Distribute any loans where the dying agent is the lender equally amongst its children;

3. Base case of recursion. Nothing to do but return results;

4. Recursive case: If the first agent in the list is dying then handle that agents inheritance and recurse through the rest of the agents otherwise just ignore it and apply the rule to rest of agents.

C.3.10 Mating

The AU specification of $Mating$ is simpler than the SU version as it does not have to construct conflict free sets. It just puts all of the potential pairs in a sequence ordered according to the variant of AU we are using and applies the rule to each in turn.

$$
\begin{array}{|l}
\hline
\textit{AsyncAgentMating} \\
\Xi Lattice \\
\Delta Agents \\
\hline
loanBook' = loanBook \\
\exists potentialMatingPairs : \mathbb{P}(AGENT \times AGENT) \mid \\
potentialMatingPairs = \{(a : AGENT, b : AGENT) \mid \\
\quad sex(a) \neq sex(b) \\
\quad \wedge\ isFertile(age(a), sex(a)) \\
\quad \wedge\ isFertile(age(head), sex(head)) \\
\quad \wedge\ adjacent(position(a), position(head))\} \\
(population', position', vision', agentSugar', agentCulture', \\
\quad metabolism', children', diseases', agentImmunity', \\
\quad age', sex', initialSugar') = \qquad\qquad (1) \\
applyMating(rndNewSweep(potentialMatingPairs), \\
\quad population, position, vision, \\
\quad agentSugar, agentCulture, metabolism, children, \\
\quad diseases, agentImmunity, age, maxAge, sex, initialSugar) \\
\hline
\end{array}
$$

1. Call $applyMating$ function on the agents in sequence.

C.3.11 Credit

$$
\begin{array}{l}
\textit{MakeLoans} \\
\hline
\Delta Agents \\
\Xi Step \\
\hline
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
agentImmunity' = agentImmunity \\
diseases' = diseases \\
children' = children \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
(loanBook', agentSugar') = \qquad (1) \\
\quad applyLoans(rndNewSweep(position), population, \\
\qquad position, agentSugar \\
\qquad , age, sex, loanBook, step) \\
\hline
\end{array}
$$

1. Call $applyLoans$ on each agent in turn.

$$
\begin{array}{|l}
applyLoans : \mathrm{seq}\, AGENT \times \mathbb{P}\, AGENT \\
\times AGENT \rightarrowtail POSITION \times AGENT \twoheadrightarrow \mathbb{N} \\
\times AGENT \twoheadrightarrow \mathbb{N} \times AGENT \twoheadrightarrow SEX \\
\times (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\times \mathbb{N} \\
\rightarrow \\
(AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\times AGENT \twoheadrightarrow \mathbb{N} \\
\hline
\forall\, population : \mathbb{P}\, AGENT; \\
position : AGENT \rightarrowtail POSITION; \\
sex : AGENT \twoheadrightarrow SEX; \\
age : AGENT \twoheadrightarrow \mathbb{N}; \\
agentSugar : AGENT \twoheadrightarrow \mathbb{N}; \\
head, ag : AGENT; \\
loanBook, loans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))); \\
tail : \mathrm{seq}\, AGENT; \\
step : \mathbb{N}; \\
\exists\, newAgentSugar : AGENT \twoheadrightarrow \mathbb{N}; \\
newLoans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))); \\
neighbours : \mathbb{P}\, AGENT \mid \\
\quad neighbours = \{b : AGENT \mid \\
\quad\quad vonNeumanNeighbour(lender, b, position)\} \qquad (1) \\
\quad (newLoans, newAgentSugar) = \\
\quad\quad\quad singleLenderLoans(lender, asSeq(neighbours), \\
\quad\quad\quad\quad\quad agentSugar, age, sex, loanBook, step) \\
applyLoans(\langle\rangle, population, position, agentSugar, \\
\quad\quad age, sex, loanBook, step) = \qquad (2) \\
\quad (loanBook, agentSugar) \\
applyLoans(\langle lender \rangle \frown tail, population, position, \\
\quad\quad\quad\quad agentSugar, age, sex, loanBook, step) = \qquad (3) \\
\quad applyLoans(tail, population, position, newAgentSugar, \\
\quad\quad age, sex, newLoans, step)
\end{array}
$$

1. Construct the set of neighbours of an agent *lender*, the updated loan book and the updated sugar levels gotten by the *lender* giving loans to its neighbours;

2. Base Case: Nothing to do just return existing values;
3. Recursive case: Recursively call $applyLoans$ on the remainder of the agents (excluding the first agent $lender$) and the new loan and sugar levels gotten by $lender$ generating new loans.

$$
\begin{array}{l}
singleLenderLoans : AGENT \times \operatorname{seq} AGENT \\
\times \mathbb{P}\, AGENT \times AGENT \nrightarrow \mathbb{N} \\
\times AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow SEX \\
\times (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \times \mathbb{N} \\
\rightarrow \\
(AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \\
\times AGENT \nrightarrow \mathbb{N}
\end{array}
$$

$$
\begin{array}{l}
\forall\, sex : AGENT \nrightarrow SEX;\ age : AGENT \nrightarrow \mathbb{N}; \\
agentSugar : AGENT \nrightarrow \mathbb{N};\ head, lender : AGENT; \\
loanBook : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))); \\
tail : \operatorname{seq} AGENT;\ step : \mathbb{N}; \\
singleLenderLoans(lender, \langle\rangle, agentSugar, \\
\qquad\qquad age, sex, loanBook, step) = \\
\qquad (loanBook, agentSugar) \\
singleLenderLoans(lender, \langle head\rangle \frown tail, agentSugar, \\
\qquad\qquad age, sex, loans, step) = \\
\textbf{if}\ canLend(age(lender), sex(lender), agentSugar(lender)) \\
\qquad \wedge\ willBorrow(age(head), sex(head), agentSugar(head), \\
\qquad\qquad \{head\} \lhd \operatorname{ran}(loanBook))) \\
\textbf{then} \\
\qquad \exists\, newAgentSugar : AGENT \nrightarrow \mathbb{N}; \\
\qquad newLoans : (AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})));\ amt : \mathbb{N} \mid \\
\qquad amt = min(amtAvail(age(lender), sex(lender), agentSugar(lender)), \\
\qquad\qquad amtReq(agentSugar(head))) \\
\qquad newAgentSugar = agentSugar \oplus \\
\qquad\qquad \{lender \mapsto agentSugar(lender) - amt, \\
\qquad\qquad head \mapsto agentSugar(head) + amt\} \\
\qquad newLoans = loanBook \cup \\
\qquad\qquad \{(lender, (head, (amt, step + DURATION)\} \\
\qquad singleLenderLoans(lender, tail, newAgentSugar, \\
\qquad\qquad age, sex, newLoans, step) \\
\textbf{else} \\
\qquad singleLenderLoans(lender, tail, agentSugar, age, sex, loanBook, step)
\end{array}
$$

$singleLenderLoans$ calculates all loans that a particular agent can give to its neighbours.

1. If there are no loans in the sequence then just return the current loans and sugar levels as is;
2. If the loan sequence is not empty then apply the payment details to the first loan and make the payments on the rest:
   a) If the first loan is capable of being paid by the borrower we simply move the correct amount of sugar from the borrower to the lender;
   b) If the borrower cannot pay off the loan then they pay back half their sugar and the loan is renegotiated for the remainder.

Using these functions we can now specify the $PayLoans$ part of the $Credit$ rule.

$PayLoans$

$\Delta Agents$
$\Xi Step$

$$\begin{aligned}
&population' = population\\
&sex' = sex\\
&position' = position\\
&vision' = vision\\
&age' = age\\
&maxAge' = maxAge\\
&agentCulture' = agentCulture\\
&agentImmunity' = agentImmunity\\
&children' = children\\
&diseases' = diseases\\
&metabolism' = metabolism\\
&initialSugar' = initialSugar\\
&\exists\, dueLoans, newLoans : \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))) \bullet\\
&dueLoans = loanBook \rhd (\text{ran}(loanBook) \rhd\\
&\qquad \{a : (\mathbb{N} \times \mathbb{N}) \mid second(a) = step\}) && (1)\\
&(newLoans, agentSugar') = makePayments(asSeq(dueLoans),\\
&\qquad \varnothing, agentSugar) && (2)\\
&loanBook' = (loanBook \setminus dueLoans) \cup newLoans && (3)
\end{aligned}$$

1. We create the set of due loans;

2. We now create the set of renegotiated loans and update the agentSugar levels using the $makePayments$ function;

3. Finally we update the loan book by removing all loans that were due and adding any new renegotiated loans.

C.4 Added Spice

C.4.1 Introduction

We have defined all the rules so far under the assumption that there is only one resource (known as $sugar$). The final rule, $Trade$, is only defined for simulations with at least two resources. In fact the rules are meant to be general enough that they will work with any number of resources although we know of no sugarscape based simulation that used more than two resources. The second resource is known as $spice$.

We will show how to extend the rules to deal with two resources. In order to avoid unnecessary clutter and make the differences as clear as possible we will show the differences between the one and two resource schemas with **boldface**. Any part of a schema that is not an exact copy of the previously defined version will be in **boldface**.

C.4.2 Basic Types

The basic types are copies of those already defined for $sugar$.

$$MAXSPICEMETABOLISM : \mathbb{N} \qquad (1)$$
$$SPICEGROWTH : \mathbb{N} \qquad (2)$$
$$MAXSPICE : \mathbb{N} \qquad (3)$$
$$INITIALSPICEMIN, INITIALSPICEMAX : \mathbb{N} \qquad (4)$$
$$SPICEPRODUCTION, SPICECONSUMPTION : \mathbb{N} \qquad (5)$$
$$SPICECOMBATLIMIT : \mathbb{N} \qquad (6)$$
$$SPICECHILDAMT : \mathbb{N} \qquad (7)$$

1. Agents metabolise spice during each move at an individually set rate less than $MAXSPICEMETABOLISM$;

2. Spice grows back at a predefined rate;

3. Each location can hold a set maximum amount of spice;
4. Agents created after mating start with an initial spice endowment;
5. Pollution can be caused by production and consumption of spice;
6. $SPICECOMBATLIMIT$ is required to help determine the reward from attacking an agent using the combat rule.
7. We posit that a minimum amount of spice is needed for agent mating to occur.

Note that these constants are replicas of their $sugar$ counterparts.

C.4.3 The SpiceScape

The spice grid contains everything in the $Lattice$ scheme and just adds information on the extra $spice$ resource.

$$\begin{array}{|ll}
\multicolumn{2}{l}{\_\,SpiceLattice\,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
Lattice & \\
\mathbf{spice} : \mathbf{POSITION} \rightarrow\!\!\!\!\!+ \mathbb{N} & \\
\mathbf{maxSpice} : \mathbf{POSITION} \rightarrow\!\!\!\!\!+ \mathbb{N} & \\
\hline
\mathrm{dom}\,\mathbf{spice} = \mathrm{dom}\,\mathbf{maxSpice} = \mathbf{POSITION} & (1) \\
\forall\,\mathbf{x} : \mathbf{POSITION} \bullet \mathbf{spice}(\mathbf{x}) \leq \mathbf{maxSpice}(\mathbf{x}) \leq \mathbf{MAXSPICE} & (2) \\
\hline
\end{array}$$

1. Every location has an associated amount of spice and maximum carrying capacity;
2. Every position's spice levels are within the acceptable levels.

C.4.4 Agents

SpiceAgents

Agents

$\mathbf{agentSpice} : \mathbf{AGENT} \nrightarrow \mathbb{N}$

$\mathbf{initialSpice} : \mathbf{AGENT} \nrightarrow \mathbb{N}$

$\mathbf{spiceMetabolism} : \mathbf{AGENT} \nrightarrow \mathbb{N}$

$\mathbf{spiceLoanBook} : \mathbf{AGENT} \leftrightarrow (\mathbf{AGENT} \times (\mathbb{N} \times \mathbb{N}))$

$\mathrm{dom}\, \mathbf{spiceMetabolism} = \mathrm{dom}\, \mathbf{agentSpice} ==$

$\quad \mathrm{dom}\, \mathbf{initialtSpicepopulation} \qquad (1)$

$\mathrm{dom}\, \mathbf{spiceLoanBook} \subseteq \mathbf{population} \qquad (2)$

$\mathrm{dom}(\mathrm{ran}\, \mathbf{spiceLoanBook}) \subseteq \mathbf{population}$

$\forall\, \mathbf{x} : \mathbf{AGENT} \bullet \mathbf{x} \in \mathbf{population} \Rightarrow$

$\quad \mathbf{spiceMetabolism}(\mathbf{x}) \leq \mathbf{MAXSPICEMETABOLISM} \qquad (3)$

1. Every agent has a spice metabolism and a spice store;
2. The spiceLoanBook has the same invariants as the original loanBook;
3. Every agents metabolism is less than or equal to the defined maximum.

Finally we combine them into an overall schema as before:

SpiceScape

SpiceAgents

SpiceLattice

Step

The initialisation scheme and tick schemas are also largely unchanged.

$$
\begin{array}{l}
\mathit{InitialSpiceScape} \\
\hline
\quad \mathit{SpiceAgents}' \\
\quad \mathit{SpiceLattice}' \\
\quad \mathit{Step}' \\
\hline
\quad \mathit{step}' = 0 \\
\quad \#\,\mathit{population}' = \mathit{INITIALPOPULATIONSIZE} \\
\quad \mathit{loanBook}' = \varnothing \\
\quad \mathbf{spiceLoanBook}' = \varnothing \\
\quad \forall\, a : \mathit{AGENT} \bullet \\
\quad a \in \mathit{population}' \Rightarrow \\
\qquad \mathit{age}'(a) = 0 \\
\qquad \mathit{diseases}'(a) = \varnothing \\
\qquad \mathit{children}'(a) = \varnothing \\
\qquad \mathit{INITIALSUGARMIN} \leq \\
\qquad\quad \mathit{agentSugar}'(a) \leq \mathit{INITIALSUGARMAX} \\
\qquad \mathit{initialSugar}'(a) = \mathit{agentSugar}'(a) \\
\qquad \mathbf{INITIALSPICEMIN} \leq \\
\qquad\quad \mathbf{agentSpice}'(\mathbf{a}) \leq \mathbf{INITIALSPICEMAX} \\
\qquad \mathbf{initialSpice}'(\mathbf{a}) = \mathbf{agentSpice}'(\mathbf{a}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Tick_{spice}\quad}} \\
\Delta SpiceAgents \\
\Delta Step \\
\hline
population' = population \\
position' = position \\
sex' = sex \\
vision' = vision \\
maxAge' = maxAge \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
\mathbf{initialSpice}' = \mathbf{initialSpice} \\
\mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism} \\
agentCulture' = agentCulture \\
children' = children \\
loanBook' = loanBook \\
\mathbf{spiceLoanBook}' = \mathbf{spiceLoanBook} \\
agentImmunity' = agentImmunity \\
diseases' = diseases \\
step' = step + 1 \\
\forall x : AGENT \bullet x \in population \Rightarrow \\
\qquad (age'(x) = age(x) + 1 \\
\qquad \wedge\ agentSugar'(x) = agentSugar(x) - metabolism(x) \\
\qquad \wedge\ \mathbf{agentSpice}'(\mathbf{x}) = \mathbf{agentSpice}(\mathbf{x}) - \mathbf{spiceMetabolism}(\mathbf{x})) \\
\hline
\end{array}
$$

C.4.5 Rules

As well as defining the final rule, $Trade$, we will also expand the other rules to allow them to operate on a simulation with two resources. We define the new rule ($Trade$) first.

C.4.6 Agent Trade T

Agent Trade T

- Agent and neighbour compute their MRSs; if these are equal then end, else continue;

- The direction of exchange is as follows: spice flows from the agent with the higher MRS to the agent with the lower MRS while sugar goes in the opposite direction;
- The geometric mean of the two MRSs is calculated-this will serve as the bargaining price, p;
- The quantities to be exchanged are as follows: if p>1 the p units of spice for 1 unit of sugar; if p < 1 the 1/p units of sugar for 1 unit of spice;
- If this trade will (a) make both agents better off (increases the welfare of both agents), and (b) not cause the agents' MRSs to cross over one another, then the trade is made and return to start, else end.

MRS is calculated simply for an agent as the fraction obtained by dividing its spice level times its sugar metabolism by its spice metabolism times its sugar level, as set out below.

$$MRS : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrowtail \mathbb{A}$$

$$\begin{aligned} &\forall\, sugar, sugarMetabolism, spice, spiceMetabolism : \mathbb{N} \bullet \\ &MRS(sugar, sugarMetabolism, spice, spiceMetabolism) = \\ &\quad (spice * sugarMetabolism)/(spiceMetabolism * sugar) \end{aligned}$$

$tradePairs$ constructs a sequence of all possible trading partners based on the proximity of the agents to each other.

$$\begin{aligned} &tradePairs : \mathrm{seq}\, AGENT \times AGENT \twoheadrightarrow\!\!\!\!\!\!\!\!\;\mapsto POSITION \rightarrowtail \\ &\quad \mathrm{seq}(AGENT \times AGENT) \end{aligned}$$

$$\begin{aligned} &\forall\, tail : \mathrm{seq}\, AGENT;\ positions : AGENT \twoheadrightarrow\!\!\!\!\!\!\!\!\;\mapsto POSITION; \\ &a : AGENT \bullet \\ &tradePairs(\langle\rangle, positions) = \langle\rangle \\ &tradePairs(\langle a \rangle \frown tail, positions) = \\ &\quad trade(tail, positions \frown \\ &\quad\quad asSeq(\{b : agent \mid adjacent(position(a), position(b)) \bullet (a, b)\}) \end{aligned}$$

$Trade$

$\Delta SpiceAgents$

$$
\begin{aligned}
&spiceLoanBook' = spiceLoanBook\\
&loanBook' = loanBook\\
&population' = population\\
&initialSugar' = initialSugar\\
&\mathbf{initialSpice}' = \mathbf{initialSpice}\\
&sex' = sex\\
&metabolism' = metabolism\\
&spiceMetabolism' = spiceMetabolism\\
&position' = position\\
&vision' = vision\\
&age' = age\\
&maxAge' = maxAge\\
&agentCulture' = agentCulture\\
&agentImmunity' = agentImmunity\\
&children' = children\\
&\exists\, allPairs : \mathrm{seq}(AGENT, AGENT) \mid\\
&allPairs = tradePairs(asSeq(population), position)\\
&(agentSugar', agentSpice') = \qquad (1)\\
&\quad executeTrades(allTrades(chooseExclusiveTrades(allPairs),\\
&\qquad (agentSugar, agentSpice), metabolism, spiceMetabolism))
\end{aligned}
$$

$Trade$ is similar to $Mating$ in that trading must be done in exclusive pairs. An agent cannot carry out two simultaneous trades and the rule forces each agent to trade with all its neighbours in some sequence. As with $Mating$ we construct conflict free sets of trading pairs that can proceed simultaneously and then order these sets.

$$
\begin{array}{|l}
chooseExclusiveTrades : AGENT \times AGENT \leftrightarrow \\
\mathrm{seq}\, \mathbb{P}(AGENT, AGENT) \\
\hline
\forall a, b : AGENT;\ tradingPairs : AGENT \leftrightarrow AGENT \bullet \\
chooseExclusiveTrades(\varnothing) = \langle\rangle \\
chooseExclusiveTrades(tradingPairs) = \\
\quad \exists maxSet : AGENT \leftrightarrow AGENT \mid maxSet \subseteq tradingPairs \\
\quad \wedge ((a, b) \in tradingPairs \wedge (a, b) \notin maxSet) \Leftrightarrow \\
\qquad \exists c : AGENT \mid \{(a, c), (c, a), (b, c), (c, b)\} \cap maxSet \neq \varnothing) \\
\quad \langle maxSet \rangle \frown chooseExclusiveTrades(tradingPairs \setminus maxSet)
\end{array}
$$

$$
\begin{array}{|l}
executeTrades : \mathrm{seq}(AGENT \times AGENT) \times \\
\quad AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow \mathbb{N} \\
\leftrightarrow (AGENT \nrightarrow \mathbb{N}, AGENT \nrightarrow \mathbb{N}) \\
\hline
\forall tail : \mathrm{seq}(AGENT \times AGENT);\ head : (AGENT \times AGENT); \\
agentSugar, agentSpice, \\
metabolism, spiceMetabolism : AGENT \nrightarrow \mathbb{N} \bullet \\
executeTrades(\langle\rangle, agentSugar, agentSpice, \\
\qquad metabolism, spiceMetabolism) = \\
\quad (agentSugar, agentSpice) \\
executeTrades(\langle head \rangle \frown tail, agentSugar, sugar) = \\
\exists newAgentSugar, newAgentSpice : AGENT \nrightarrow \mathbb{N} \mid \\
(newAgentSugar, newAgentSpice) = \\
\quad allTrades(head, (agentSugar, agentSpice, \\
\qquad metabolism, spiceMetabolism) \\
\bullet\ executeTrades(tail, newAgentSugar, \\
\quad newAgentSpice, metabolism, spiceMetabolism)
\end{array}
$$

$allTrades$ recursively goes through the sequence of trading partners and gets each trading pair to update the sugar and spice levels based on their trades.

$$
\begin{array}{|l}
allTrades : \mathrm{seq}(AGENT \times AGENT) \times ((AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \times \\
\qquad\qquad (AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N})) \times \\
\qquad\qquad (AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \\
\qquad \leftrightarrow \\
\qquad ((AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N})) \\
\hline
\forall\, head : AGENT \times AGENT;\ tail : \mathrm{seq}(AGENT \times AGENT) \\
sugar, spice, sugarMetabolism, spiceMetabolism : AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N} \bullet \\
allTrades(\langle\rangle, (sugar, spice), sugarMetabolism, spiceMetabolism) = \\
\qquad (sugar, spice) \\
allTrades(\langle head \rangle \frown tail, (sugar, spice), sugarMetabolism, \\
\qquad spiceMetabolism) = \\
\qquad allTrades(tail, pairTrade(head, (sugar, spice), \\
\qquad sugarMetabolism, spiceMetabolism) \\
\qquad\qquad , sugarMetabolism, spiceMetabolism)
\end{array}
$$

Each trading partnership will execute a series of trades until their MRS scores cross over. $pairTrade$ is complicated by the fact that there are multiple options:

1. If their MRS scores are equal then they perform no trades;
2. If their MRS scores are not equal then the direction of trade will depend on which MRS score is higher;
   a) Within a trade a value of p (based on MRS scores) determines the price of the resources.
3. Trades between the pair continue until their new and old MRS scores cross over.

$$pairTrade : (AGENT \times AGENT) \times ((AGENT \pinj \mathbb{N}) \times$$
$$\qquad\qquad (AGENT \pinj \mathbb{N})) \times$$
$$\qquad\qquad (AGENT \pinj \mathbb{N}) \times (AGENT \pinj \mathbb{N})$$
$$\qquad \leftrightarrow ((AGENT \pinj \mathbb{N}) \times (AGENT \pinj \mathbb{N}))$$

$$\forall\, a, b : AGENT;\ sugar, spice : AGENT \pinj \mathbb{N};$$
$$metabolism, spiceMetabolism : AGENT \pinj \mathbb{N} \mid$$
$$\exists\, p, mrsA, mrsB, newMrsA, newMrsB : \mathbb{A};$$
$$newSugar, newSpice : AGENT \pinj \mathbb{N} \mid$$
$$mrsA = MRS(sugar(a), metabolism(a), spice(a), spiceMetabolism(a))$$
$$mrsB = MRS(sugar(b), metabolism(b), spice(b), spiceMetabolism(b))$$
$$p = \sqrt{mrsA * mrsB}$$
$$(mrsA > mrsB \wedge p > 1) \Rightarrow$$
$$\qquad (newSugar = sugar \oplus \{(a, sugar(a) + 1), (b, sugar(b) - 1)\}$$
$$\qquad \wedge\ newSpice = spice \oplus \{a \mapsto spice(a) - p, b \mapsto spice(b) + p\})$$
$$(mrsA > mrsB \wedge p \leq 1) \Rightarrow$$
$$\qquad (newSugar = sugar \oplus \{a \mapsto sugar(a) + (1 \operatorname{div} p),$$
$$\qquad\qquad b \mapsto sugar(b) - (1 \operatorname{div} p)\}$$
$$\qquad newSpice = spice \oplus \{a \mapsto spice(a) - 1, b \mapsto spice(b) + 1\})$$
$$(mrsA \leq mrsB \wedge p > 1) \Rightarrow$$
$$\qquad (newSugar = sugar \oplus \{b \mapsto sugar(b) + 1, a \mapsto sugar(a) - 1\}$$
$$\qquad \wedge\ newSpice = spice \oplus \{b \mapsto spice(b) - p, a \mapsto spice(a) + p\})$$
$$(mrsA \leq mrsB \wedge p \leq 1) \Rightarrow$$
$$\qquad (newSugar = sugar \oplus$$
$$\qquad\qquad \{b \mapsto sugar(b) + (1 \operatorname{div} p), a \mapsto sugar(a) - (1 \operatorname{div} p)\}$$
$$\qquad \wedge\ newSpice = spice \oplus \{b \mapsto spice(b) - 1, a \mapsto spice(a) + 1\})$$
$$newMrsA = MRS(newSugar(a), metabolism(a),$$
$$\qquad newSpice(a), spiceMetabolism(a))$$
$$newMrsB = MRS(newSugar(b), metabolism(b),$$
$$\qquad newSpice(b), spiceMetabolism(b))$$
$$\bullet$$
$$pairTrade((a, b), (sugar, spice), metabolism, spiceMetabolism)$$
$$\qquad = (sugar, spice)$$
$$\Leftrightarrow mrsA = mrsB$$
$$pairTrade((a, b), (sugar, spice), metabolism, spiceMetabolism)$$
$$\qquad = (newSugar, newSpice)$$
$$\Leftrightarrow ((mrsA > mrsB \wedge newMrsA \leq newMrsB)$$
$$\qquad \vee (mrsA < mrsB \wedge newMrsA \geq newMrsB))$$

$$\begin{aligned}
&pairTrade((a,b),(sugar,spice),metabolism,spiceMetabolism) = \\
&\quad pairTrade((a,b),(newSugar,newSpice), \\
&\qquad metabolism,spiceMetabolism) \\
&\quad \Leftrightarrow ((mrsA > mrsB \land newMrsA > newMrsB) \\
&\quad \lor (mrsA < mrsB \land newMrsA < newMrsB))
\end{aligned}$$

C.4.7 Asynchronous Trade

$$\begin{aligned}
&\textit{Trade} \\
&\Delta SpiceAgents \\
\hline
&spiceLoanBook' = spiceLoanBook \\
&loanBook' = loanBook \\
&population' = population \\
&sex' = sex \\
&metabolism' = metabolism \\
&spiceMetabolism' = spiceMetabolism \\
&initialSugar' = initialSugar \\
&\mathbf{initialSpice}' = \mathbf{initialSpice} \\
&position' = position \\
&vision' = vision \\
&age' = age \\
&maxAge' = maxAge \\
&agentCulture' = agentCulture \\
&agentImmunity' = agentImmunity \\
&children' = children \\
&\exists traders : \mathbb{P}(AGENT \times AGENT) \bullet \\
&(agentSugar', agentSpice') = \qquad (1) \\
&\quad allTrades(tradePairs(rndNewSweep(position), positions), \\
&\qquad (agentSugar, agentSpice), \\
&\qquad metabolism, spiceMetabolism)
\end{aligned}$$

1. The new sugar and spice allocations are derived by conducting all possible trades using the recursive helper function $allTrades$.

Growback

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Growback}_{spice}} \\
\hline
\Delta SpiceLattice \qquad (1) \\
\hline
pollution' = pollution \\
maxSugar' = maxSugar \\
\mathbf{maxSpice' = maxSpice} \qquad (2) \\
sugar' = sugar \oplus \{x : POSITION \bullet \\
\quad x \mapsto min(\{sugar(x) + SUGARGROWTH, maxSugar(x)\})\} \\
\mathbf{spice' = spice \oplus \{x : POSITION \bullet} \\
\quad \mathbf{x \mapsto min(\{spice(x) + SPICEGROWTH, maxSpice(x)\})\}} \qquad (3) \\
\hline
\end{array}
$$

1. $Lattice$ is replaced with $SpiceLattice$. In all subsequent schemas $SpiceLattice$ replaces $Lattice$ and $SpiceAgent$ replaces $Agent$;

2. $maxSpice$ remains unchanged;

3. The new spice levels are calculated using the same simple formula used for sugar growback.

Seasonal Growback

$$
\begin{array}{l}
\textit{SeasonalGrowback}_{spice} \\
\hline
\Delta SpiceLattice \\
\Xi Step \\
\hline
pollution' = pollution \\
maxSugar' = maxSugar \\
\mathbf{maxSpice'} = \mathbf{maxSpice} \\
\forall x : POSITION \bullet \\
(step \mathrel{\mathrm{div}} SEASONLENGTH) \mathrel{\mathrm{mod}} 2 = 0 \Rightarrow sugar' = \\
\quad \{x : POSITION \mid first(x) < M \mathrel{\mathrm{div}} 2 \bullet \\
\quad x \mapsto min(\{sugar(x) + SUGARGROWTH, maxSugar(x)\})\} \\
\quad \cup \\
\quad \{x : POSITION \mid first(x) \geq M \mathrel{\mathrm{div}} 2 \bullet \\
\quad x \mapsto min(\{sugar(x) + SUGARGROWTH \mathrel{\mathrm{div}} \\
\qquad WINTERRATE, maxSugar(x)\})\} \\
\forall x : POSITION \bullet \\
(step \mathrel{\mathrm{div}} SEASONLENGTH) \mathrel{\mathrm{mod}} 2 \neq 0 \Rightarrow sugar' = \\
\quad \{x : POSITION \mid first(x) < M \mathrel{\mathrm{div}} 2 \bullet \\
\quad x \mapsto min(\{sugar(x) + SUGARGROWTH \mathrel{\mathrm{div}} \\
\qquad WINTERRATE, maxSugar(x)\})\} \\
\quad \cup \\
\quad \{x : POSITION;\ y : \mathbb{N} \mid first(x) \geq M \mathrel{\mathrm{div}} 2 \bullet \\
\quad x \mapsto min(\{sugar(x) + SUGARGROWTH, maxSugar(x)\})\}
\end{array}
$$

$$
\begin{aligned}
&\forall \mathbf{x} : \mathbf{POSITION} \bullet && (1)\\
&(\mathbf{step}\ \mathrm{div}\ \mathbf{SEASONLENGTH})\ \mathrm{mod}\ \mathbf{2} = \mathbf{0} \Rightarrow \mathbf{spice}' = \\
&\quad \{\mathbf{x} : \mathbf{POSITION} \mid \mathbf{first}(\mathbf{x}) < \mathbf{M}\ \mathrm{div}\ \mathbf{2} \bullet \\
&\quad \mathbf{x} \mapsto \mathbf{min}(\{\mathbf{spice}(\mathbf{x}) + \mathbf{SPICEGROWTH}, \mathbf{maxSpice}(\mathbf{x})\})\} \\
&\quad \cup \\
&\quad \{\mathbf{x} : \mathbf{POSITION} \mid \mathbf{first}(\mathbf{x}) \geq \mathbf{M}\ \mathrm{div}\ \mathbf{2} \bullet \\
&\quad \mathbf{x} \mapsto \mathbf{min}(\{\mathbf{spice}(\mathbf{x}) + \mathbf{SPICEGROWTH}\mathrm{div} \\
&\qquad \mathbf{WINTERRATE}, \mathbf{maxSpice}(\mathbf{x})\})\} \\
&\forall \mathbf{x} : \mathbf{POSITION} \bullet \\
&(\mathbf{step}\ \mathrm{div}\ \mathbf{SEASONLENGTH})\ \mathrm{mod}\ \mathbf{2} \neq \mathbf{0} \Rightarrow \mathbf{spice}' = \\
&\quad \{\mathbf{x} : \mathbf{POSITION} \mid \mathbf{first}(\mathbf{x}) < \mathbf{M}\ \mathrm{div}\ \mathbf{2} \bullet \\
&\quad \mathbf{x} \mapsto \mathbf{min}(\{\mathbf{spice}(\mathbf{x}) + \mathbf{SPICEGROWTH}\mathrm{div} \\
&\qquad \mathbf{WINTERRATE}, \mathbf{maxSpice}(\mathbf{x})\})\} \\
&\quad \cup \\
&\quad \{\mathbf{x} : \mathbf{POSITION} \mid \mathbf{first}(\mathbf{x}) \geq \mathbf{M}\ \mathrm{div}\ \mathbf{2} \bullet \\
&\quad \mathbf{x} \mapsto \mathbf{min}(\{\mathbf{spice}(\mathbf{x}) + \mathbf{SPICEGROWTH}, \mathbf{maxSpice}(\mathbf{x})\})\}
\end{aligned}
$$

1. Seasonal growback adds a rule for spice grow back that is an exact replica of the sugar rule. We note that we only use the one $WINTERRATE$ instead of a separate rate for sugar and spice. In the absence of any explicit direction on this point this solution seems to be the most obvious.

C.4.8 Movement

In order to update Movement we will need to implement a welfare function that can be used to measure the desirability of a location. With two resources the desirability of any location becomes a subjective measure, what one agent may rate highly another may not. Welfare is dependent on the agents current levels of spice and sugar, so an agent with low spice levels may consider a location containing spice more desirable than one containing sugar. Overall the desirability of a location is determined by the agents current resource levels (wealth) and the relative metabolism rates for each resource.

This is in contrast to the previous approach where welfare just equaled the amount of sugar in a location. This welfare measure is precisely defined in the book and we follow this definition.

$$welfare : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrowtail \mathbb{A}$$

$$\begin{aligned}
&\forall\, agentSugar, sugarMetabolism,\\
&\quad agentSpice, spiceMetabolism,\\
&\quad locationSugar, locationSpice : \mathbb{N} \bullet\\
&welfare(agentSugar, sugarMetabolism,\\
&\quad agentSpice, spiceMetabolism,\\
&\quad locationSugar, locationSpice) =\\
&\quad (locationSugar + agentSugar) * (sugarMetabolism \operatorname{div}\\
&\qquad\qquad (sugarMetabolism + spiceMetabolism))\\
&\quad *(locationSpice + agentSpice) * (spiceMetabolism \operatorname{div}\\
&\qquad\qquad (sugarMetabolism + spiceMetabolism))
\end{aligned}$$

Movement can now be restated by replacing the previous measure of a locations desirability (sugar level) with this new measure and the updating of spice levels. In all other respects the schema remains unchanged.

$$
\begin{array}{l}
\textit{Movement}_{basicSpice} \\
\hline
\Delta SpiceScape \\
\hline
step' = step \land population' = population \\
maxSugar' = maxSugar \land pollution' = pollution \\
sex' = sex \land vision' = vision \\
age' = age \land initialSugar' = initialSugar \\
\mathbf{initialSpice'} = \mathbf{initialSpice} \\
metabolism' = metabolism \land maxAge' = maxAge \\
agentCulture' = agentCulture \land loanBook' = loanBook \\
diseases' = diseases \land agentImmunity' = agentImmunity \\
children' = children \\
\mathbf{maxSpice'} = \mathbf{maxSpice} \land \mathbf{spiceLoanBook'} = \mathbf{spiceLoanBook} \\
\mathbf{spiceMetabolism'} = \mathbf{spiceMetabolism} \\
\forall a : AGENT;\ l : POSITION \bullet \\
\quad a \in population \Rightarrow distance(position'(a), position(a)) \leq vision(a) \\
\quad distance(position(a), l) \leq vision(a) \land (l \notin \operatorname{ran} position')) \\
\qquad \Rightarrow \mathbf{welfare}(\mathbf{agentSugar}(\mathbf{a}), \mathbf{metabolism}(\mathbf{a}), \qquad (1) \\
\qquad\quad \mathbf{agentSpice}(\mathbf{a}), \mathbf{spiceMetabolism}(\mathbf{a}), \mathbf{sugar}(\mathbf{l}), \mathbf{spice}(\mathbf{l})) \\
\qquad < \\
\qquad \mathbf{welfare}(\mathbf{agentSugar}(\mathbf{a}), \mathbf{metabolism}(\mathbf{a}), \\
\qquad\quad \mathbf{agentSpice}(\mathbf{a}), \mathbf{spiceMetabolism}(\mathbf{a}), \\
\qquad\qquad \mathbf{sugar}(\mathbf{position'}(\mathbf{a})), \mathbf{spice}(\mathbf{position'}(\mathbf{a}))) \\
agentSugar' = \{a : AGENT \mid a \in population \bullet \\
\quad a \mapsto agentSugar(a) + sugar(position'(a))\} \\
sugar' = sugar \oplus \{loc : POSITION \mid loc \in \operatorname{ran} position' \bullet loc \mapsto 0\} \\
\mathbf{agentSpice'} = \{\mathbf{a} : \mathbf{AGENT} \mid \mathbf{a} \in \mathbf{population} \bullet \qquad (2) \\
\quad \mathbf{a} \mapsto \mathbf{agentSpice}(\mathbf{a}) \\
\qquad + \mathbf{spice}(\mathbf{position'}(\mathbf{a}))\} \\
\mathbf{spice'} = \mathbf{spice} \oplus \{\mathbf{loc} : \mathbf{POSITION} \mid \mathbf{loc} \in \operatorname{ran} \mathbf{position'} \bullet \mathbf{loc} \mapsto \mathbf{0}\} \qquad (3) \\
\hline
\end{array}
$$

1. This is a copy of the original proposition with the $welfare$ function now replacing the previous sugar level check;

2. Agents consume spice at their new locations;

3. Locations with agents present now have no remaining spice.

C.4.9 Pollution Formation $P_{\Pi,\chi}$

The $Movement_{spicePollution}$ schema has the same alterations as the $Movement_{basicSpice}$

$$
\begin{array}{l}
Movement_{pollutionSpice} \\
\hline
\Delta SpiceScape \\
\hline
step' = step \land population' = population \\
maxSugar' = maxSugar \land sex' = sex \\
pollution' = pollution \land vision' = vision \\
age' = age \land maxAge' = maxAge \\
agentCulture' = agentCulture \land loanBook' = loanBook \\
children' = children \land agentImmunity' = agentImmunity \\
initialSugar' = initialSugar \\
\mathbf{initialSpice}' = \mathbf{initialSpice} \\
diseases' = diseases \\
metabolism' = metabolism \\
\mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism} \\
\mathbf{spiceLoanBook}' = \mathbf{spiceLoanBook} \\
\mathbf{maxSpice}' = \mathbf{maxSpice} \\
\forall a : AGENT;\ l : POSITION \mid a \in \mathrm{dom}(position') \bullet \\
\quad a \in population \Rightarrow distance(position'(a), position(a)) \leq vision(a) \\
\quad (distance(position(a), l) \leq vision(a) \land (l \notin \mathrm{ran}\, position')) \\
\qquad \Rightarrow \mathbf{welfare}(\mathbf{agentSugar}(\mathbf{a}), \mathbf{metabolism}(\mathbf{a}), \\
\qquad\quad \mathbf{agentSpice}(\mathbf{a}), \mathbf{spiceMetabolism}(\mathbf{a}), \mathbf{sugar}(\mathbf{l}), \mathbf{spice}(\mathbf{l})) \\
\qquad\quad \mathrm{div}(\mathbf{1} + \mathbf{pollution}(\mathbf{l})) \\
\qquad < \\
\qquad \mathbf{welfare}(\mathbf{agentSugar}(\mathbf{a}), \mathbf{metabolism}(\mathbf{a}), \\
\qquad\quad \mathbf{agentSpice}(\mathbf{a}), \mathbf{spiceMetabolism}(\mathbf{a}), \mathbf{sugar}(\mathbf{position}'(\mathbf{a})), \\
\qquad\qquad \mathbf{spice}(\mathbf{position}'(\mathbf{a}))) \ \mathrm{div}\ (\mathbf{1} + \mathbf{pollution}(\mathbf{position}'(\mathbf{a})))
\end{array}
$$

$$
\begin{array}{l}
agentSugar' = \{a : AGENT \mid a \in population \bullet \\
\quad a \mapsto agentSugar(a) + sugar(position'(a))\} \\
sugar' = sugar \oplus \{l : POSITION \mid l \in \operatorname{ran} position' \bullet l \mapsto 0\} \\
\mathbf{agentSpice}' = \{\mathbf{a} : \mathbf{AGENT} \mid \mathbf{a} \in \mathbf{population} \bullet \\
\quad \mathbf{a} \mapsto \mathbf{agentSpice}(\mathbf{a}) + \mathbf{spice}(\mathbf{position}'(\mathbf{a}))\} \\
\mathbf{spice}' = \mathbf{spice} \oplus \{\mathbf{l} : \mathbf{POSITION} \mid \mathbf{l} \in \operatorname{ran} \mathbf{position}' \bullet \mathbf{l} \mapsto \mathbf{0}\} \\
pollution' = pollution \oplus \\
\quad \{l : POSITION;\ x : AGENT \mid position'(x) = l \bullet \\
\quad l \mapsto (PRODUCTION * sugar(l) + \\
\qquad CONSUMPTION * metabolism(x))\}
\end{array}
$$

C.4.10 Pollution Diffusion

$PollutionDiffusion_{spice}$

$$
\begin{array}{l}
\Delta SpiceLattice \\
\Xi Step \\
\hline
maxSugar' = maxSugar \\
sugar' = sugar \\
\mathbf{maxSpice}' = \mathbf{maxSpice} \\
\mathbf{spice}' = \mathbf{spice} \\
(step \bmod POLLUTIONRATE \neq 0) \Rightarrow pollution' = pollution \\
(step \bmod POLLUTIONRATE = 0) \Rightarrow pollution' = \\
\quad \{l : POSITION \bullet \\
\qquad l \mapsto (pollution(north(l)) + pollution(south(l)) \\
\qquad +pollution(east(l)) + pollution(west(l))) \operatorname{div} 4\}
\end{array}
$$

C.4.11 Replacement

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Death}_{spice}} \\
\Delta \mathit{SpiceAgents} \\
\hline
\mathit{population}' = \mathit{population} \setminus \{a : \mathit{AGENT} \mid \mathit{age}(a) = \mathit{maxAge}(a) \\
\qquad \lor\ \mathit{agentSugar}(a) = 0 \lor \mathbf{agentSpice}(\mathbf{a}) = \mathbf{0} \bullet a\} \\
\mathit{loanBook}' = \mathit{population}' \lhd \mathit{loanBook} \rhd \\
\qquad \{x : \mathit{AGENT} \times (\mathbb{N} \times \mathbb{N}) \mid \mathit{first}(x) \in \mathit{population}'\} \\
\mathbf{spiceLoanBook}' = \mathbf{population}' \lhd \mathbf{spiceLoanBook} \rhd \\
\qquad \{\mathbf{x} : \mathbf{AGENT} \times (\mathbb{N} \times \mathbb{N}) \mid \mathbf{first}(\mathbf{x}) \in \mathbf{population}'\} \\
\forall\, a : \mathit{AGENT} \bullet \\
\qquad a \in \mathit{population}' \Rightarrow \\
\qquad\qquad (\mathit{sex}(a) = \mathit{sex}'(a) \land \mathit{vision}(a) = \mathit{vision}'(a) \\
\qquad\qquad \land\ \mathit{maxAge}(a) = \mathit{maxAge}'(a) \\
\qquad\qquad \land\ \mathit{agentCulture}(a) = \mathit{agentCulture}'(a) \\
\qquad\qquad \land\ \mathit{position}(a) = \mathit{position}'(a) \land \mathit{age}(a) = \mathit{age}'(a) \\
\qquad\qquad \land\ \mathit{agentSugar}(a) = \mathit{agentSugar}'(a) \\
\qquad\qquad \land\ \mathit{metabolism}'(a) = \mathit{metabolism}(a) \\
\qquad\qquad \land\ \mathit{diseases}'(a) = \mathit{diseases}(a) \\
\qquad\qquad \land\ \mathit{agentImmunity}'(a) = \mathit{agentImmunity}(a) \\
\qquad\qquad \land\ \mathit{children}'(a) = \mathit{children}(a) \\
\qquad\qquad \land\ \mathit{initialSugar}'(a) = \mathit{initialSugar}(a) \\
\qquad\qquad \land\ \mathbf{agentSpice}'(\mathbf{a}) = \mathbf{agentSpice}(\mathbf{a}) \\
\qquad\qquad \land\ \mathbf{initialSpice}'(\mathbf{a}) = \mathbf{initialSpice}(\mathbf{a}) \\
\qquad\qquad \land\ \mathbf{spiceMetabolism}'(\mathbf{a}) = \mathbf{spiceMetabolism}(\mathbf{a})) \\
\hline
\end{array}$$

$$
\begin{array}{l}
\mathit{Replacement}_{spice} \\
\hline
\Delta \mathit{SpiceAgents} \\
\hline
\#\mathit{population}' = \mathit{INITIALPOPULATIONSIZE} \\
\mathit{loanBook}' = \mathit{loanBook} \\
\mathit{spiceLoanBook}' = \mathit{spiceLoanBook} \\
\forall a : \mathit{AGENT} \bullet \\
a \in (\mathit{population}) \Rightarrow \\
\quad (a \in \mathit{population}' \\
\quad \land \mathit{sex}(a) = \mathit{sex}'(a) \land \mathit{vision}(a) = \mathit{vision}'(a) \\
\quad \land \mathit{maxAge}(a) = \mathit{maxAge}'(a) \\
\quad \land \mathit{agentCulture}(a) = \mathit{agentCulture}'(a) \\
\quad \land \mathit{position}(a) = \mathit{position}'(a) \land \mathit{age}(a) = \mathit{age}'(a) \\
\quad \land \mathit{agentSugar}'(a) = \mathit{agentSugar}(a) \\
\quad \land \mathit{metabolism}'(a) = \mathit{metabolism}(a) \\
\quad \land \mathit{initialSugar}'(a) = \mathit{initialSugar}(a) \\
\quad \land \mathbf{initialSpice}'(\mathbf{a}) = \mathbf{initialSpice}(\mathbf{a}) \\
\quad \land \mathbf{agentSpice}'(\mathbf{a}) = \mathbf{agentSpice}(\mathbf{a}) \\
\quad \land \mathbf{spiceMetabolism}'(\mathbf{a}) = \mathbf{spiceMetabolism}(\mathbf{a}) \\
\quad \land \mathit{diseases}'(a) = \mathit{diseases}(a) \\
\quad \land \mathit{agentImmunity}'(a) = \mathit{agentImmunity}(a) \\
\quad \land \mathit{children}'(a) = \mathit{children}(a)) \\
\forall a : \mathit{AGENT} \bullet \\
\quad a \in \mathit{population}' \setminus \mathit{population} \Rightarrow (\mathit{age}'(a) = 0 \\
\quad \land \mathit{INITIALSUGARMIN} \leq \mathit{agentSugar}'(a) \\
\qquad \leq \mathit{INITIALSUGARMAX} \\
\quad \land \mathit{initialSugar}'(a) = \mathit{agentSugar}'(a) \\
\quad \land \mathbf{INITIALSPICEMIN} \leq \mathbf{agentSpice}'(\mathbf{a}) \\
\qquad \leq \mathbf{INITIALSPICEMAX} \\
\quad \land \mathbf{initialSpice}'(\mathbf{a}) = \mathbf{agentSpice}'(\mathbf{a}) \\
\quad \land \mathit{diseases}'(a) = \varnothing \land \mathit{children}'(a) = \varnothing) \\
\hline
\end{array}
$$

C.4.12 Agent Mating

$$
\begin{array}{|l}
\textit{AgentMating} \\
\Xi \textit{Lattice} \\
\Delta \textit{Agents} \\
\hline
loanBook' = loanBook \land \mathbf{spiceLoanBook}' = \mathbf{spiceLoanBook} \\
\exists potentialMatingPairs : \mathbb{P}(AGENT \times AGENT) \mid \qquad (1) \\
potentialMatingPairs = \{(a : AGENT, b : AGENT) \mid sex(a) \neq sex(b) \\
\quad \land isFertile(age(a), sex(a)) \land isFertile(age(head), sex(head)) \\
\quad \land adjacent(position(a), position(head))\} \\
(population', position', vision', agentSugar', agentCulture', metabolism' \\
\quad , children', diseases', agentImmunity', age', sex', initialSugar', \\
\quad \mathbf{spiceMetabolism}', \mathbf{agentSpice}', \mathbf{initialSpice}') = \qquad (2) \\
concurrentMating(getConfictFreePairs(potentialMatingPairs), \\
\quad population, position, vision, \\
\quad agentSugar, agentCulture, metabolism, children, \\
\quad diseases, agentImmunity, age, maxAge, sex, initialSugar, \\
\quad \mathbf{spiceMetabolism}, \mathbf{agentSpice}, \mathbf{initialSpice})
\end{array}
$$

1. Generate the set of all possible mating pairs;

2. Recursively proceed with concurrent mating within the conflict free subsets.

$concurrentMating : \mathrm{seq}\,\mathbb{P}(AGENT \times AGENT) \times \mathbb{P}\,AGENT$
$\times AGENT \rightarrowtail POSITION \times AGENT \nrightarrow \mathbb{N}_1$
$\times AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow \mathrm{seq}\,BIT$
$\times AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow \mathbb{P}\,AGENT$
$\times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\,BIT \times AGENT \nrightarrow \mathrm{seq}\,BIT$
$\times AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow \mathbb{N}_1$
$\times AGENT \nrightarrow SEX \times AGENT \nrightarrow \mathbb{N}$
$\times \mathbf{AGENT} \nrightarrow \mathbb{N} \times \mathbf{AGENT} \nrightarrow \mathbb{N}$
$\times \mathbf{AGENT} \nrightarrow \mathbb{N}$
$\leftrightarrow$
$\mathbb{P}\,AGENT \times AGENT \rightarrowtail POSITION$
$\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow \mathbb{N}$
$\times AGENT \nrightarrow \mathrm{seq}\,BIT \times AGENT \nrightarrow \mathbb{N}$
$\times AGENT \nrightarrow \mathbb{P}\,AGENT \times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\,BIT$
$\times AGENT \nrightarrow \mathrm{seq}\,BIT \times AGENT \nrightarrow \mathbb{N}$
$\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow SEX$
$\times AGENT \nrightarrow \mathbb{N} \times \mathbf{AGENT} \nrightarrow \mathbb{N}$
$\times \mathbf{AGENT} \nrightarrow \mathbb{N} \times \mathbf{AGENT} \nrightarrow \mathbb{N}$

$\forall\, tail : \mathrm{seq}\,\mathbb{P}(AGENT \times AGENT);\ head : \mathbb{P}(AGENT \times AGENT);$
$population : \mathbb{P}\,AGENT;\ position : AGENT \rightarrowtail POSITION;$
$vision : AGENT \nrightarrow \mathbb{N}_1;\ agentSugar, \mathbf{agentSpice} : AGENT \nrightarrow \mathbb{N};$
$agentCulture : AGENT \nrightarrow \mathrm{seq}\,BIT;$
$metabolism, \mathbf{spiceMetabolism} : AGENT \nrightarrow \mathbb{N};$
$children : AGENT \nrightarrow \mathbb{P}\,AGENT;\ diseases : AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\,BIT;$
$agentImmunity : AGENT \nrightarrow \mathrm{seq}\,BIT;\ age : AGENT \nrightarrow \mathbb{N};$
$maxAge : AGENT \nrightarrow \mathbb{N}_1;\ sex : AGENT \nrightarrow SEX;$
$initialSugar, \mathbf{initialSpice} : AGENT \nrightarrow \mathbb{N};$

$\exists\, newpopulation : \mathbb{P}\, AGENT;$
$newposition : AGENT \rightarrowtail POSITION;\; newvision : AGENT \nrightarrow \mathbb{N}_1;$
$newagentSugar, \mathbf{newagentSpice} : AGENT \nrightarrow \mathbb{N};$
$newagentCulture : AGENT \nrightarrow \mathrm{seq}\, BIT;$
$newmetabolism, \mathbf{newspiceMetabolism} : AGENT \nrightarrow \mathbb{N};$
$newchildren : AGENT \nrightarrow \mathbb{P}\, AGENT;$
$newdiseases : AGENT \nrightarrow \mathbb{P}\, \mathrm{seq}\, BIT;$
$newagentImmunity : AGENT \nrightarrow \mathrm{seq}\, BIT;$
$newage : AGENT \nrightarrow \mathbb{N};\; newmaxAge : AGENT \nrightarrow \mathbb{N}_1;$
$newsex : AGENT \nrightarrow SEX;$
$newinitialSugar, \mathbf{newinitialSpice} : AGENT \nrightarrow \mathbb{N};\; |$
$(newpopulation, newposition, newvision, newagentSugar,$
$\quad newagentCulture, newmetabolism, newchildren,$
$\quad newdiseases, newagentImmunity, newage,$
$\quad newmaxAge, newsex, newinitialSugar,$
$\quad \mathbf{newspiceMetabolism}, \mathbf{newagentSpice}, \mathbf{newinitialspice}) =$
$applyMating(asSeq(head), population, position, vision,$
$\quad agentSugar, agentCulture, metabolism, children,$
$\quad diseases, agentImmunity, age, maxAge, sex, initialSugar,$
$\quad \mathbf{spiceMetabolism}, \mathbf{agentSpice}, \mathbf{initialspice}) \bullet$
$concurrentMating(\langle\rangle, population, position, vision,$
$\quad agentSugar, agentCulture, metabolism, children,$
$\quad diseases, agentImmunity, age, maxAge, sex, initialSugar) =$
$(population, position, vision, agentSugar, agentCulture, metabolism,$
$\quad children, diseases, agentImmunity, age, maxAge, sex, initialSugar,$
$\mathbf{spiceMetabolism}, \mathbf{agentSpice}, \mathbf{initialspice})$
$concurrentMating(\langle head\rangle \frown tail, population, position, vision,$
$\quad agentSugar, agentCulture, metabolism, children,$
$\quad diseases, agentImmunity, age, maxAge, sex, initialSugar) =$
$concurrentMating(tail, newpopulation, newposition, newvision,$
$\quad newagentSugar, newagentCulture, newmetabolism, newchildren,$
$\quad newdiseases, newagentImmunity, newage,$
$\quad newmaxAge, newsex, newinitialSugar,$
$\mathbf{newspiceMetabolism}, \mathbf{newagentSpice}, \mathbf{newinitialspice})$

$applyMating : \mathrm{seq}(AGENT \times AGENT) \times \mathbb{P}\, AGENT$
$\times AGENT \rightarrowtail POSITION \times AGENT \nrightarrow \mathbb{N}_1$
$\times AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow \mathrm{seq}\, BIT$
$\times AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow \mathbb{P}\, AGENT$
$\times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\, BIT \times AGENT \nrightarrow \mathrm{seq}\, BIT$
$\times AGENT \nrightarrow \mathbb{N} \times AGENT \nrightarrow \mathbb{N}_1$
$\times AGENT \nrightarrow SEX \times AGENT \nrightarrow \mathbb{N}$
$\times \mathbf{AGENT} \nrightarrow \mathbb{N} \times \mathbf{AGENT} \nrightarrow \mathbb{N}$
$\times \mathbf{AGENT} \nrightarrow \mathbb{N}$
$\leftrightarrow$
$\mathbb{P}\, AGENT \times AGENT \rightarrowtail POSITION$
$\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow \mathbb{N}$
$\times AGENT \nrightarrow \mathrm{seq}\, BIT \times AGENT \nrightarrow \mathbb{N}$
$\times AGENT \nrightarrow \mathbb{P}\, AGENT \times AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\, BIT$
$\times AGENT \nrightarrow \mathrm{seq}\, BIT \times AGENT \nrightarrow \mathbb{N}$
$\times AGENT \nrightarrow \mathbb{N}_1 \times AGENT \nrightarrow SEX$
$\times AGENT \nrightarrow \mathbb{N} \times \mathbf{AGENT} \nrightarrow \mathbb{N}$
$\times \mathbf{AGENT} \nrightarrow \mathbb{N} \times \mathbf{AGENT} \nrightarrow \mathbb{N}$

$\forall\, population : \mathbb{P}\, AGENT;\ position : AGENT \rightarrowtail POSITION;$
$sex : AGENT \nrightarrow SEX;\ vision : AGENT \nrightarrow \mathbb{N}_1;$
$age : AGENT \nrightarrow \mathbb{N};\ initialSugar, \mathbf{initialSpice} : AGENT \nrightarrow \mathbb{N};$
$maxAge : AGENT \nrightarrow \mathbb{N}_1;$
$metabolism, \mathbf{spiceMetabolism} : AGENT \nrightarrow \mathbb{N};$
$agentSugar, \mathbf{agentSpice} : AGENT \nrightarrow \mathbb{N};$
$agentCulture : AGENT \nrightarrow \mathrm{seq}\, BIT;$
$children : AGENT \nrightarrow \mathbb{P}\, AGENT;$
$agentImmunity : AGENT \nrightarrow \mathrm{seq}\, BIT;$
$diseases : AGENT \nrightarrow \mathbb{P}\,\mathrm{seq}\, BIT;\ head : AGENT \times AGENT;$
$tail : \mathrm{seq}(AGENT \times AGENT);\ \bullet$

$\exists offspring, a, b : AGENT;\ newsex : AGENT \pfun SEX;$
$newvision : AGENT \pfun \mathbb{N}_1;$
$newmetabolism, newagentSugar, newinitialSugar : AGENT \pfun \mathbb{N};$
$\mathbf{newspiceMetabolism}, \mathbf{newagentSpice}, \mathbf{newinitialSpice} : \mathbf{AGENT} \pfun \mathbb{N};$
$newmaxAge : AGENT \pfun \mathbb{N}_1;$
$newagentCulture : AGENT \pfun \operatorname{seq} BIT;$
$newchildren : AGENT \pfun \mathbb{P}\, AGENT;$
$newagentImmunity : AGENT \pfun \operatorname{seq} BIT;$
$inheritedImmunity : \operatorname{seq} BIT;$
$inheritedCulture : \operatorname{seq} BIT;\ |\ offspring \notin population$
$a = first(head) \land b = second(head)$
$newchildren : children \cup \{offspring \mapsto \varnothing\, a \mapsto children(a) \cup \{offspring\},$
$\quad b \mapsto children(b) \cup \{offspring\}\}$
$newsex = sex \cup \{offspring \mapsto male\}$
$\quad \lor newsex = sex \cup \{offspring \mapsto female\}$
$newvision = vision \cup \{offspring \mapsto vision(a)\}$
$\quad \lor newvision = vision \cup \{offspring \mapsto vision(b)\}$
$newmaxAge = maxAge \cup \{offspring \mapsto maxAge(a)\}$
$\quad \lor newmaxAge = maxAge \cup \{offspring \mapsto maxAge(b)\}$
$newmetabolism = metabolism \cup \{offspring \mapsto metabolism(a)\}$
$\quad \lor newmetabolism = metabolism \cup \{offspring \mapsto metabolism(b)\}$
$\mathbf{newspiceMetabolism} = \mathbf{spiceMetabolism} \cup \{\mathbf{offspring} \mapsto \mathbf{spiceMetabolism}(\mathbf{a})\}$
$\quad \lor \mathbf{newspiceMetabolism} = \mathbf{spiceMetabolism} \cup$
$\quad\quad \{\mathbf{offspring} \mapsto \mathbf{spiceMetabolism}(\mathbf{b})\}$
$newinitialSugar = initialSugar \oplus$
$\quad \{offspring \mapsto initialSugar(a)/2 + initialSugar(b)/2\}$
$newagentSugar = agentSugar \oplus$
$\quad \{offspring \mapsto initialSugar(offspring),$
$\quad\quad a \mapsto initialSugar(a)/2, b \mapsto initialSugar(b)/2\}$
$\mathbf{newinitialSpice} = \mathbf{initialSpice} \oplus$
$\quad \{\mathbf{offspring} \mapsto \mathbf{initialSpice}(\mathbf{a})/\mathbf{2} + \mathbf{initialSpice}(\mathbf{b})/\mathbf{2}\}$
$\mathbf{newagentSpice} = \mathbf{agentSpice}$
$\quad \oplus \{\mathbf{offspring} \mapsto \mathbf{initialSpice}(\mathbf{offspring}),$
$\quad\quad \mathbf{a} \mapsto \mathbf{initialSpice}(\mathbf{a})/\mathbf{2}, \mathbf{b} \mapsto \mathbf{initialSpice}(\mathbf{b})/\mathbf{2}\}$

$$\begin{aligned}
&\quad \land \forall n : 1 \mathrel{..} IMMUNITYLENGTH \bullet \\
&\qquad (inheritedImmunity(n) = agentImmunity(a)(n) \\
&\qquad \lor inheritedImmunity(n) = agentImmunity(b)(n)) \\
&newagentImmunity : agentImmunity \cup \{offspring \mapsto inheritedImmunity\} \\
&\quad \land \forall n : 1 \mathrel{..} CULTURECOUNT \bullet \\
&\qquad (inheritedCulture(n) = agentCulture(a)(n) \\
&\qquad \lor inheritedCulture(n) = agentCulture(b)(n)) \\
&newagentCulture : agentCulture \cup \{offspring \mapsto inheritedCulture\} \\
&applyMating(\langle\rangle, population, position, vision, agentSugar, agentCulture, \\
&\quad metabolism, children, diseases, agentImmunity, age, maxAge, \\
&\quad sex, initialSugar, \mathbf{spiceMetabolism}, \mathbf{agentSpice}, \mathbf{initialSpice}) = \\
&(population, position, vision, agentSugar, agentCulture, metabolism, \\
&\quad children, diseases, agentImmunity, age, \\
&\quad maxAge, sex, initialSugar, \mathbf{spiceMetabolism}, \mathbf{agentSpice}, \mathbf{initialSpice}) \\
&applyMating(\langle head\rangle \frown tail, population, position, vision, agentSugar, \\
&\quad agentCulture, metabolism, children, diseases, agentImmunity, age, \\
&\quad maxAge, sex, initialSugar, \mathbf{spiceMetabolism}, \mathbf{agentSpice}, \mathbf{initialSpice}) = \\
&\mathbf{if}((\exists\, loc : POSITION \mid (adjacent(loc, position(ag))) \\
&\qquad \lor adjacent(loc, position(head)) \land loc \notin \operatorname{dom} position) \\
&\qquad \land (\mathbf{agentSpice}(\mathbf{head}) > \mathbf{initialSpice}(\mathbf{head})) \\
&\qquad\quad mathbf{\land}\ (agentSpice(ag) > initialSpice(ag)) \\
&\qquad \land (agentSugar(head) > initialSugar(head)) \\
&\qquad\quad \land (agentSugar(ag) > initialSugar(ag))) \\
&\mathbf{then} && (2a) \\
&\quad applyMating(tail, population \cup \{offspring\}, position \cup \\
&\qquad \{offspring \mapsto loc\}, \\
&\quad newvision, newagentSugar, newagentCulture, \\
&\quad newmetabolism, newchildren, diseases \cup \{offspring \mapsto \varnothing\}, \\
&\quad newagentImmunity, age \cup \{offspring \mapsto 0\}, newmaxAge, newsex, \\
&\quad initialSugar, \mathbf{newspiceMetabolism}, \mathbf{newagentSpice}, \mathbf{newinitialSpice}) \\
&\mathbf{else} && (2b) \\
&\quad applyMating(tail, population, position, vision, agentSugar, \\
&\quad agentCulture, metabolism, children, diseases, agentImmunity, age, \\
&\quad maxAge, sex, initialSugar, \mathbf{spiceMetabolism}, \mathbf{agentSpice}, \mathbf{initialSpice})
\end{aligned}$$

C.4.13 Culture

$$
\begin{array}{l}
\textit{Culture}_{spice} \\
\hline
\Delta \textit{SpiceAgents} \\
\hline
\textit{population}' = \textit{population} \\
\textit{sex}' = \textit{sex} \\
\textit{position}' = \textit{position} \\
\textit{vision}' = \textit{vision} \\
\textit{age}' = \textit{age} \\
\textit{maxAge}' = \textit{maxAge} \\
\textit{agentSugar}' = \textit{agentSugar} \\
\textit{children}' = \textit{children} \\
\textit{loanBook}' = \textit{loanBook} \\
\textit{diseases}' = \textit{diseases} \\
\textit{initialSugar}' = \textit{initialSugar} \\
\textit{metabolism}' = \textit{metabolism} \\
\textit{agentImmunity}' = \textit{agentImmunity} \\
\textbf{agentSpice}' = \textbf{agentSpice} \\
\textbf{spiceMetabolism}' = \textbf{spiceMetabolism} \\
\textbf{spiceLoanBook}' = \textbf{spiceLoanBook} \\
\textbf{initialSpice}' = \textbf{initialSpice} \\
\forall\, a : \textit{AGENT} \bullet a \in \textit{population} \Rightarrow \\
\quad \textit{agentCulture}'(a) = \textit{flipTags}(\textit{agentCulture}(a), \\
\qquad \textit{asSeq}(\{b : \textit{AGENT} \mid \textit{adjacent}(\textit{position}(a), \textit{position}(b))\}), \\
\qquad \textit{agentCulture}) \\
\hline
\end{array}
$$

C.4.14 Disease

$$
\begin{array}{l}
\textit{ImmuneResponse}_{spice} \\
\hline
\Xi \textit{SpiceLattice} \\
\Delta \textit{SpiceAgents} \\
\Xi \textit{Step} \\
\hline
\textit{loanBook}' = \textit{loanBook} \\
\mathbf{spiceLoanBook}' = \mathbf{spiceLoanBook} \\
\textit{population}' = \textit{population} \\
\textit{sex}' = \textit{sex} \\
\textit{position}' = \textit{position} \\
\textit{vision}' = \textit{vision} \\
\textit{age}' = \textit{age} \\
\textit{maxAge}' = \textit{maxAge} \\
\textit{agentCulture}' = \textit{agentCulture} \\
\textit{diseases}' = \textit{diseases} \\
\textit{children}' = \textit{children} \\
\textit{agentSugar}' = \textit{agentSugar} \\
\textit{initialSugar}' = \textit{initialSugar} \\
\mathbf{initialSpice}' = \mathbf{initialSpice} \\
\textit{metabolism}' = \textit{metabolism} \\
\mathbf{agentSpice}' = \mathbf{agentSpice} \\
\mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism} \\
\textit{agentImmunity}' = \{a : \textit{AGENT} \mid a \in \textit{population} \bullet \\
\quad a \mapsto \textit{applyDiseases}(\textit{agentImmunity}(a), \textit{asSeq}(\textit{diseases}(a)))\} \\
\forall x : \textit{AGENT} \bullet x \in \textit{population} \Rightarrow \textit{agentSugar}'(x) = \textit{agentSugar}(x) - \\
\quad \#\{d : \mathrm{seq}\, \textit{BIT} \mid d \in \textit{diseases}(a) \land \neg\, \textit{subseq}(d, \textit{agentImmunity}(a)\} \\
\forall x : \textit{AGENT} \bullet x \in \textit{population} \Rightarrow \textit{agentSpice}'(x) = \textit{agentSpice}(x) - \\
\quad \#\{d : \mathrm{seq}\, \textit{BIT} \mid d \in \textit{diseases}(a) \land \neg\, \textit{subseq}(d, \textit{agentImmunity}(a)\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transmission}_{spice}} \\
\hline
\Delta \textit{SpiceAgents} \\
\hline
\textit{loanBook}' = \textit{loanBook} \\
\textit{initialSugar}' = \textit{initialSugar} \\
\mathbf{spiceLoanBook}' = \mathbf{spiceLoanBook} \\
\textit{population}' = \textit{population} \\
\textit{sex}' = \textit{sex} \\
\textit{position}' = \textit{position} \\
\textit{vision}' = \textit{vision} \\
\textit{age}' = \textit{age} \\
\textit{maxAge}' = \textit{maxAge} \\
\textit{agentCulture}' = \textit{agentCulture} \\
\textit{agentImmunity}' = \textit{agentImmunity} \\
\textit{children}' = \textit{children} \\
\textit{agentSugar}' = \textit{agentSugar} \\
\mathbf{agentSpice}' = \mathbf{agentSpice} \\
\mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism} \\
\mathbf{initialSpice}' = \mathbf{initialSpice} \\
\textit{metabolism}' = \textit{metabolism} \\
\forall a : \textit{AGENT} \bullet a \in \textit{population} \Rightarrow \\
\textit{diseases}'(a) = \textit{diseases}(a) \cup \\
\quad \textit{newDiseases}(\textit{asSeq}(\textit{visibleAgents}(a, \textit{position}, 1)), \textit{diseases}) \\
\hline
\end{array}
$$

C.4.15 Inheritance

$$
\begin{array}{l}
\textit{Inheritance}_{spice} \\
\hline
\Delta \textit{SpiceAgents} \\
\hline
population' = population \land sex' = sex \\
position' = position \land vision' = vision \\
age' = age \land maxAge' = maxAge \\
agentCulture' = agentCulture \land children' = children \\
metabolism' = metabolism \land \mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism} \\
diseases' = diseases \land agentImmunity' = agentImmunity \\
initialSugar' = initialSugar \\
\mathbf{initialSpice}' = \mathbf{initialSpice} \\
\exists\, dying : \mathbb{P}\, AGENT; \\
inheritFromFemale, inheritFromMale : AGENT \mathrel{+\!\!\!\!\rightarrow} (\mathbb{N} \times \mathbb{N}) \bullet \qquad (1) \\
\operatorname{dom} inheritFromFemale = \operatorname{dom} inheritFromMale = population \setminus dying \\
\quad dying = \{x : AGENT \mid x \in population \land (age(x) = maxAge(x) \lor \\
\qquad agentSugar(x) = 0 \lor \mathbf{agentSpice}(\mathbf{x}) = \mathbf{0})\} \qquad (2) \\
\forall\, x : AGENT;\ n, m : \mathbb{N} \mid x \in population \setminus dying \bullet \\
\quad getMother(x, children, sex) \notin dying \Rightarrow \\
\qquad inheritFromFemale(x) = (0, 0) \\
\quad getFather(x, children, sex) \notin dying \Rightarrow \\
\qquad inheritFromMale(x) = (0, 0) \\
\quad \exists\, m : AGENT \mid m = getMother(x, children, sex) \land m \in dying \Rightarrow \\
\qquad inheritFromFemale(x) = \\
\qquad\quad (agentSugar(m) \operatorname{div} \#(population \cap children(m) \setminus dying)) \\
\qquad\quad (\mathbf{agentSpice}(\mathbf{m}) \operatorname{div} \#(\mathbf{population} \cap \mathbf{children}(\mathbf{m}) \setminus \mathbf{dying})) \quad (3) \\
\quad \exists\, d : AGENT \mid d = getFather(x, children, sex) \land d \in dying \Rightarrow \\
\qquad inheritFromMale(x) = \\
\qquad\quad agentSugar(d) \operatorname{div} \#(population \cap children(d) \setminus dying)) \\
\qquad\quad (\mathbf{agentSpice}(\mathbf{m}) \operatorname{div} \#(\mathbf{population} \cap \mathbf{children}(\mathbf{m}) \setminus \mathbf{dying}))
\end{array}
$$

$$
\begin{aligned}
&x \in dying\\
&\qquad \Rightarrow (agentSugar'(x) = 0 \wedge \mathbf{agentSpice'(x) = 0})\\
&x \notin dying\\
&\qquad \Rightarrow (agentSugar'(x) = agentSugar(x)\\
&\qquad\qquad + first(inheritFromMale(x)) + first(inheritFromFemale(x))\\
&\qquad\qquad \wedge\ \mathbf{agentSpice'(x) = agentSpice(x)}\\
&\qquad\qquad \mathbf{+second(inheritFromFemale(x)) + second(inheritFromMale(x)))}\\
&loanBook' = disperseLoans(loanBook, asSeq(dying), children)\\
&\mathbf{spiceLoanBook'} =\\
&\qquad \mathbf{disperseLoans(spiceLoanBook, asSeq(dying), children)} \qquad (4)
\end{aligned}
$$

1. Agents can now inherit two amounts, a $sugar$ inheritance and a $spice$ inheritance;
2. Death now occurs if either resource reaches zero;
3. The individual $spice$ inheritance is calculated in the same way as the $sugar$ inheritance;
4. Spice loans are dispersed amongst children.

C.4.16 Combat

Combat is defined only in terms of sugar. We can either accept this and assume combat is based only on sugar levels or we can extend combat by defining new versions of wealth and reward. We note that no simulations combining combat with more than one resource are presented in the book.

We can extend the combat rule with a few simple assumptions. First the wealth of an agent is used to determine if we can attack that agent or if an agent can retaliate against us. In the single resource scenario we simply used the sugar that an agent carried. With two resources we need to combine both sugar and spice. The simplest approach is to add these two amounts together and in the absence of any guidelines this seems the sensible option.

$availMoves$ requires only minor changes to return the set of all safe moves that an agent can make.

$$
\begin{array}{l}
availMoves_{spice} : AGENT \times (AGENT \rightarrowtail\!\!\!\!\rightarrow POSITION) \\
\quad \times (POSITION \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \\
\quad \times (POSITION \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\rightarrow \mathrm{seq}\, BIT) \times \mathbb{N} \\
\quad \nrightarrow \mathbb{P}\, POSITION
\end{array}
$$

$$
\begin{array}{l}
\forall\, x, agent : AGENT;\ positions : AGENT \rightarrowtail\!\!\!\!\rightarrow POSITION;\ vision : \mathbb{N}; \\
sugar, spice : POSITION \rightarrowtail\!\!\!\!\rightarrow \mathbb{N};\ agentSpice, agentSugar : AGENT \rightarrowtail\!\!\!\!\rightarrow \mathbb{N}; \\
culture : AGENT \rightarrowtail\!\!\!\!\rightarrow \mathrm{seq}\, BIT \bullet \\
\\
availMoves_{spice}(agent, positions, sugar, agentSugar, spice, \\
\quad agentSpice, culture, vision) = \\
\{ l : POSITION;\ x : AGENT \mid l \in distance(l, positions(agent)) \leq vision \\
\wedge\ positions(x) = l \Rightarrow (agentSugar(x) + \mathbf{agentSpice}(\mathbf{x}) < \\
\qquad\qquad agentSugar(agent) + \mathbf{agentSpice}(\mathbf{ag}) \\
\quad \wedge\ tribe(culture(x)) \neq tribe(culture(agent))) \\
\wedge\ ((distance(positions(x), l) \leq vision) \\
\quad \wedge\ tribe(culture(x)) \neq tribe(culture(agent))) \Rightarrow \\
\qquad agentSugar(x) + \mathbf{agentSpice}(\mathbf{x}) < agentSugar(agent) \\
\qquad\quad + \mathbf{agentSpice}(\mathbf{agent}) \\
\qquad + reward(l, sugar, positions, agentSugar, COMBATLIMIT)) \\
\qquad + \mathbf{reward}(\mathbf{l}, \mathbf{spice}, \mathbf{positions}, \mathbf{agentSpice}, \mathbf{SPICECOMBATLIMIT}) \bullet l \}
\end{array}
\tag{3}
$$

$$\mathit{Combat}_{spice}$$

$$\Delta \mathit{SugarScape}$$

$$
\begin{aligned}
&\mathit{step}' = \mathit{step}\\
&\mathit{maxSugar}' = \mathit{maxSugar}\\
&\mathbf{maxSpice}' = \mathbf{maxSpice}\\
&\mathit{pollution}' = \mathit{pollution}\\
&\mathit{loanBook}' = \mathit{population}' \lhd \mathit{loanBook} \rhd\\
&\qquad (\mathit{population}' \lhd (\operatorname{ran} \mathit{loanBook}))\\
&\mathbf{spiceLoanBook}' =\\
&\qquad \mathbf{population}' \lhd \mathbf{spiceLoanBook} \rhd\\
&\qquad\qquad (\mathbf{population}' \lhd (\operatorname{ran} \mathbf{spiceLoanBook}))\\
&\forall\, ag : \mathit{AGENT};\ l : \mathit{POSITION} \bullet\\
&\mathit{ag} \in \mathit{population}' \Rightarrow \qquad\qquad\qquad\qquad\qquad\qquad (3)\\
&\qquad (\mathit{sex}'(ag) = \mathit{sex}(ag)\\
&\qquad \wedge\ \mathit{vision}'(ag) = \mathit{vision}(ag)\\
&\qquad \wedge\ \mathit{age}'(ag) = \mathit{age}(ag)\\
&\qquad \wedge\ \mathit{maxAge}'(ag) = \mathit{maxAge}(ag)\\
&\qquad \wedge\ \mathit{children}'(ag) = \mathit{children}(ag)\\
&\qquad \wedge\ \mathit{agentCulture}'(ag) = \mathit{agentCulture}(ag)\\
&\qquad \wedge\ \mathit{agentImmunity}'(ag) = \mathit{agentImmunity}(ag)\\
&\qquad \wedge\ \mathit{metabolism}'(ag) = \mathit{metabolism}(ag)\\
&\qquad \wedge\ \mathit{initialSugar}'(ag) = \mathit{initialSugar}(ag)\\
&\qquad \wedge\ \mathbf{initialSpice}'(\mathbf{ag}) = \mathbf{initialSpice}(\mathbf{ag})\\
&\qquad \wedge\ \mathbf{spiceMetabolism}'(\mathbf{ag}) = \mathbf{spiceMetabolism}(\mathbf{ag})\\
&\qquad \wedge\ \mathit{diseases}'(ag) = \mathit{diseases}(ag))\\
&(\mathit{population}', \mathit{position}', \mathit{sugar}', \mathit{agentSugar}', \mathit{agentSpice}') =\\
&\qquad \mathit{applyAllCombat}_{spice}(\mathit{asSeq}(\mathit{population}), \mathit{population}, \mathit{position},\\
&\qquad\qquad \mathit{sugar}, \mathit{agentSugar}, \mathit{agentSpice}, \mathit{vision}, \mathit{agentCulture})
\end{aligned}
$$

$$
\begin{array}{l}
applyAllCombat_{spice} : \mathrm{seq}\, AGENT \times \mathbb{P}\, AGENT \\
\quad \times (AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow\!\!\!\!\mapsto POSITION) \\
\quad \times (POSITION \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}) \\
\quad \times (AGENT \nrightarrow \mathbb{N}) \times (AGENT \nrightarrow \mathrm{seq}\, BIT) \\
\rightarrow \\
(\mathbb{P}\, AGENT \times (AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow POSITION) \times (POSITION \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}) \\
\quad \times (AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}))
\end{array}
$$

$$
\begin{array}{l}
\forall\, head : AGENT;\ tail : \mathrm{seq}\, AGENT;\ pop : \mathbb{P}\, AGENT; \\
positions : AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow POSITION;\ sugar : POSITION \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}; \\
agSugar, agSpice : AGENT \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbb{N}; \\
vision : AGENT \nrightarrow \mathbb{N};\ culture : AGENT \nrightarrow \mathrm{seq}\, BIT \bullet \\
applyAllCombat(\langle\rangle, pop, positions, sugar, \\
\qquad agSugar, agSpice, vision, culture) = \\
\quad (pop, positions, sugar, agSugar, agSpice) \\
\\
applyAllCombat(\langle head\rangle \frown tail, pop, positions, sugar, \\
\qquad agSugar, agSpice, vision, culture) = \\
\mathbf{if}(head \in pop)\mathbf{then} \\
\quad applyAllCombat_{spice}(tail, \\
\qquad singleFight_{spice}(head, pop, positions, sugar, \\
\qquad\quad agSugar, agSpice, vision, culture)) \\
\mathbf{else} \\
\quad applyAllCombat_{spice}(tail, pop, positions, sugar, \\
\qquad\quad agSugar, agSpice, vision, culture)
\end{array}
$$

$$
\begin{array}{l}
singleFight_{spice} : AGENT \times \mathbb{P}\, AGENT \times (AGENT \rightarrowtail\!\!\!\twoheadrightarrow POSITION) \\
\qquad \times (POSITION \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}) \\
\qquad \times (AGENT \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}) \times (AGENT \nrightarrow \mathbb{N}) \times (AGENT \nrightarrow \mathrm{seq}\, BIT) \\
\rightarrow \\
(\mathbb{P}\, AGENT \times (AGENT \rightarrowtail\!\!\!\twoheadrightarrow POSITION) \times (POSITION \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}) \\
\qquad \times (AGENT \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}) \times (AGENT \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}))
\end{array}
$$

$$
\begin{array}{l}
\forall\, ag : AGENT;\; population : \mathbb{P}\, AGENT; \\
positions : AGENT \rightarrowtail\!\!\!\twoheadrightarrow POSITION; \\
sugar : POSITION \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N};\; agSugar, agSpice : AGENT \rightarrowtail\!\!\!\twoheadrightarrow \mathbb{N}; \\
vision : AGENT \nrightarrow \mathbb{N};\; culture : AGENT \nrightarrow \mathrm{seq}\, BIT \bullet \\
singleFight_{spice}(ag, population, positions, sugar, \\
\qquad agSugar, agSpice, vision, culture) = \\
\textbf{if}\, (availMoves(ag, positions, sugar, agSugar, \\
\qquad\qquad agSpice, culture, vision(ag)) = \varnothing)\; \textbf{then} \\
\qquad (population, positions, sugar, agSugar, agSpice) \\
\textbf{else} \\
\qquad \exists\, loc : POSITION;\; available : \mathbb{P}\, POSITION; \\
\qquad \forall\, otherLoc : POSITION \mid \\
\qquad loc, otherLoc \in availMoves_{spice}(ag, positions, \\
\qquad\qquad sugar, agSugar, agSpice, culture, vision(ag)) \\
\qquad \wedge\; otherLocation \neq location \\
\qquad reward(loc, sugar, position, agSugar, COMBATLIMIT) \\
\qquad + \mathbf{reward}(\mathbf{loc}, \mathbf{spice}, \mathbf{position}, \mathbf{agSpice}, \mathbf{SPICECOMBATLIMIT}) \\
\qquad \geq reward(otherLoc, sugar, position, agSugar, COMBATLIMIT) \\
\qquad + \mathbf{reward}(\mathbf{otherLoc}, \mathbf{spice}, \mathbf{position}, \mathbf{agSpice}, \mathbf{SPICECOMBATLIMIT}) \\
\qquad (distance(position(ag), loc) > distance(position(otherLoc), position'(ag)) \\
\qquad \Rightarrow reward(loc, sugar, position, agSugar, COMBATLIMIT) \\
\qquad + \mathbf{reward}(\mathbf{loc}, \mathbf{sugar}, \mathbf{position}, \mathbf{agSpice}, \mathbf{SPICECOMBATLIMIT}) \\
\qquad > reward(otherLoc, sugar, position, agSugar, COMBATLIMIT) \\
\qquad + \mathbf{reward}(\mathbf{otherLoc}, \mathbf{spice}, \mathbf{position}, \mathbf{agSpice}, \mathbf{SPICECOMBATLIMIT})) \bullet \\
\qquad (population \setminus \{positions^{\sim}(loc)\}, (positions \rhd \{loc\}) \oplus \\
\qquad\qquad \{ag \mapsto loc\}, sugar \oplus \{loc \mapsto 0\}, \\
\qquad agSugar \oplus \{ag \mapsto agSugar(ag) + \\
\qquad\qquad reward(position'(ag), sugar, position, agSugar, COMBATLIMIT)\}, \\
\qquad \mathbf{agSpice} \oplus \{\mathbf{ag} \mapsto \mathbf{agSpice}(\mathbf{ag}) + \\
\qquad\qquad \mathbf{reward}(\mathbf{position}'(\mathbf{ag}), \mathbf{spice}, \mathbf{position}, \mathbf{agSpice}, \mathbf{SPICECOMBATLIMIT})\})
\end{array}
$$

$$SynchronousCombat_{spice}$$

$$\Delta SugarScape$$

$$step' = step$$
$$maxSugar' = maxSugar$$
$$\mathbf{maxSpice}' = \mathbf{maxSpice}$$
$$pollution' = pollution$$
$$loanBook' = population' \lhd loanBook \rhd (population' \lhd (\operatorname{ran} loanBook))$$
$$\mathbf{spiceLoanBook}' =$$
$$\quad \mathbf{population}' \lhd \mathbf{spiceLoanBook} \rhd (\mathbf{population}' \lhd (\operatorname{ran} \mathbf{spiceLoanBook}))$$
$$population' \subseteq population$$
$$sugar' = sugar \oplus \{p : POSITION \mid p \in \operatorname{ran} position' \bullet p \mapsto 0\}$$
$$\forall ag : AGENT;\ l : POSITION \bullet$$
$$ag \in population' \Rightarrow$$
$$\quad (sex'(ag) = sex(ag)$$
$$\quad \land\ vision'(ag) = vision(ag)$$
$$\quad \land\ age'(ag) = age(ag)$$
$$\quad \land\ maxAge'(ag) = maxAge(ag)$$
$$\quad \land\ children'(ag) = children(ag)$$
$$\quad \land\ agentCulture'(ag) = agentCulture(ag)$$
$$\quad \land\ agentImmunity'(ag) = agentImmunity(ag)$$
$$\quad \land\ metabolism'(ag) = metabolism(ag)$$
$$\quad \land\ initialSugar'(ag) = initialSugar(ag)$$
$$\quad \land\ \mathbf{initialSpice}'(\mathbf{ag}) = \mathbf{initialSpice}(\mathbf{ag})$$
$$\land\ \mathbf{spiceMetabolism}(\mathbf{ag}) = \mathbf{spiceMetabolism}(\mathbf{ag})$$
$$\quad \land\ diseases'(ag) = diseases(ag)$$
$$\quad \land\ agentSugar'(ag) = agentSugar(ag)$$
$$\qquad + reward(position'(ag), sugar, position,$$
$$\qquad\quad agentSugar, COMBATLIMIT)$$

$$
\begin{array}{l}
\qquad \land\ agentSpice'(ag) = agentSpice(ag) \\
\qquad\qquad +\mathbf{reward}(\mathbf{position}'(\mathbf{ag}), \mathbf{spice}, \mathbf{position}, \mathbf{agentSpice}, \\
\qquad\qquad\qquad\qquad \mathbf{SPICECOMBATLIMIT}) \\
\qquad \land\ position'(ag) \in \\
\qquad\qquad availMoves_{spice}(ag, position, sugar, agentSugar, \\
\qquad\qquad\qquad agentSpice, agentCulture, vision(ag))) \\
\qquad) \\
ag \in population \setminus population' \Rightarrow \\
\qquad \exists x : AGENT \bullet position'(x) = position(ag) \\
\qquad\qquad \land\ tribe(culture(x)) \neq tribe(culture(ag)) \\
(l \in availMoves_{spice}(ag, position, sugar, agentSugar, \\
\qquad\qquad agentSpice, agentCulture, vision(ag)) \\
\qquad \land\ reward(l, sugar, position, agentSugar, COMBATLIMIT) \\
\qquad +\mathbf{reward}(\mathbf{l}, \mathbf{spice}, \mathbf{position}, \mathbf{agentSpice}, \mathbf{SPICECOMBATLIMIT}) \\
\qquad \geq reward(position'(ag), sugar, position, \\
\qquad\qquad agentSugar, COMBATLIMIT) \\
\qquad +\mathbf{reward}(\mathbf{position}'(\mathbf{ag}), \mathbf{spice}, \mathbf{position}, \mathbf{agentSpice}, \\
\qquad\qquad\qquad \mathbf{SPICECOMBATLIMIT}) \\
\qquad \land\ distance(position(ag), l) < distance(position(ag), position'(ag))) \\
\qquad \Rightarrow \exists x : AGENT \bullet position'^{\sim}(l) = x \land position(x) \neq l
\end{array}
$$

C.4.17 Credit

Credit is defined with one resource. It is incorporated into a dual resource simulation but no extra information is given as to what changes were made, if any. The most logical approach is to assume that $spice$ loans are administered the the exact same way as $sugar$. We create a separate system of loans for $spice$ that is dealt with in the exact same manner as $sugar$ loans. Our specification is now split into two parts, the parts dealing with $sugar$, already specified, and the parts dealing with $spice$, which are copies of their counterparts. The amount of a resource available to be borrowed now depends on which resource we are talking about.

$$amtAvail_{new} : \mathbb{N} \times SEX \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\forall\, age, resource, baseAmt : \mathbb{N} \bullet$$

$$amtAvail_{new}(age, male, resource, baseAmt) =$$
$$\quad resource \operatorname{div} 2 \Leftrightarrow age > MALEFERTILITYEND$$
$$amtAvail_{new}(age, male, resource, baseAmt) = resource - baseAmt \Leftrightarrow$$
$$\qquad (age \leq MALEFERTILITYEND \land$$
$$\qquad isFertile(age, male) \land resource > baseAmt)$$
$$amtAvail_{new}(age, male, resource, baseAmt) = 0 \Leftrightarrow$$
$$\qquad (age \leq MALEFERTILITYEND \land$$
$$\qquad \neg\, (isFertile(age, male) \lor resource > baseAmt))$$
$$amtAvail_{new}(age, female, resource, baseAmt) =$$
$$\quad resource \operatorname{div} 2 \Leftrightarrow age > FEMALEFERTILITYEND$$
$$amtAvail_{new}(age, female, resource, baseAmt) = resource - baseAmt \Leftrightarrow$$
$$\qquad (age \leq FEMALEFERTILITYEND \land$$
$$\qquad isFertile(age, female) \land resource > baseAmt)$$
$$amtAvail_{new}(age, female, resource, baseAmt) = 0 \Leftrightarrow$$
$$\qquad (age \leq FEMALEFERTILITYEND$$
$$\qquad \land \neg\, (isFertile(age, female) \lor resource > baseAmt))$$

$$
\begin{array}{l}
\textit{PaySugarLoans} \\
\hline
\Delta Agents \\
\Xi Step \\
\hline
population' = population \\
sex' = sex \\
position' = position \\
vision' = vision \\
age' = age \\
maxAge' = maxAge \\
agentCulture' = agentCulture \\
agentImmunity' = agentImmunity \\
children' = children \\
diseases' = diseases \\
metabolism' = metabolism \\
initialSugar' = initialSugar \\
\mathbf{initialSpice}' = \mathbf{initialSpice} \\
\mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism} \\
\mathbf{spiceLoanBook}' = \mathbf{spiceLoanBook} \\
\mathbf{agentSpice}' = \mathbf{agentSpice} \\
\exists\, dueLoans, newLoans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \bullet \\
dueLoans = loanBook \rhd \\
\qquad (\mathrm{ran}(loanBook) \rhd \{a : (\mathbb{N} \times \mathbb{N}) \mid second(a) = step\}) \\
(loanBook', agentSugar') = \\
\qquad payExclusiveLoans(chooseConflictFreeSets(dueLoans), \\
\qquad\qquad agentSugar, loanBook) \\
\hline
\end{array}
$$

$PaySpiceLoans$

$\Delta Agents$

$\Xi Step$

$population' = population$

$sex' = sex$

$position' = position$

$vision' = vision$

$age' = age$

$maxAge' = maxAge$

$agentCulture' = agentCulture$

$agentImmunity' = agentImmunity$

$children' = children$

$diseases' = diseases$

$metabolism' = metabolism$

$initialSugar' = initialSugar$

$\mathbf{initialSpice}' = \mathbf{initialSpice}$

$\mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism}$

$loanBook' = loanBook$

$agentSugar' = agentSugar$

$\exists\, dueLoans, newLoans : (AGENT \leftrightarrow (AGENT \times (\mathbb{N} \times \mathbb{N}))) \bullet$

$dueLoans = loanBook \rhd$

$\quad (\mathrm{ran}(spiceLoanBook) \rhd \{a : (\mathbb{N} \times \mathbb{N}) \mid second(a) = step\})$

$(spiceLoanBook', agentSpice') =$

$\quad payExclusiveLoans(chooseConflictFreeSets(dueLoans),$

$\qquad agentSpice, spiceLoanBook)$

$MakeLoans_{Sugar}$

$\Delta Agents$
$\Xi Step$

$population' = population \land sex' = sex$
$position' = position \land vision' = vision$
$age' = age \land maxAge' = maxAge$
$agentCulture' = agentCulture \land agentImmunity' = agentImmunity$
$diseases' = diseases \land children' = children$
$metabolism' = metabolism$
$initialSugar' = initialSugar$
$\mathbf{initialSpice}' = \mathbf{initialSpice}$
$\mathbf{spiceMetabolism}' = \mathbf{spiceMetabolism}$
$\mathbf{agentSpice}' = \mathbf{agentSpice}$
$\mathbf{spiceLoanBook}' = \mathbf{spiceLoanBook}$
$\exists\, newLoans : \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})));$
$\forall\, ag, lender, borrower : AGENT;\ amt, due : \mathbb{N} \bullet$
$loanBook' = loanBook \cup newLoans$
$ag \in \mathrm{dom}\, newLoans \Rightarrow$
$\quad agentSugar'(ag) = agentSugar(ag) - totalLoaned(ag, newLoans)$
$ag \in \mathrm{dom}(\mathrm{ran}\, newLoans) \Rightarrow$
$\quad agentSugar'(ag) = agentSugar(ag) + totalOwed(ag, newLoans)$
$ag \notin \mathrm{dom}(newLoans) \cup \mathrm{dom}(\mathrm{ran}\, newLoans) \Rightarrow$
$\quad agentSugar'(ag) = agentSugar(ag)$
$willBorrow(age(ag), sex(ag), agentSugar'(ag),$
$\quad \mathrm{ran}(loanBook' \cap \{a : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N}))$
$\qquad\qquad \mid borrower(a) = borrower(loan)\})) \Rightarrow$
$\qquad \neg\ \exists\, ag2 : AGENT \bullet canLend(age(ag2),$
$\qquad\quad sex(ag2), agentSugar'(ag2))$
$\qquad\quad \land\ adjacent(position(ag2), position(ag))$

$$
\begin{array}{l}
totalLoaned(ag, newLoans) \leq \\
\quad amtAvail_{new}(age(ag), sex(ag), agentSugar(ag), CHILDAMT) \\
totalOwed(ag, newLoans) \leq CHILDAMT - agentSugar(ag) \\
(lender, (borrower, (amt, due))) \in newLoans \Rightarrow \\
\quad (canLend(age(lender), sex(lender), agentSugar(lender)) \\
\quad \land willBorrow(age(borrower), sex(borrower), agentSugar(borrower), \\
\quad\quad \{borrower\} \lhd (\mathrm{ran}\, loanBook)) \\
\quad \land amt \leq \\
\quad\quad min(\{amtAvail_{new}(age(lender), sex(lender), agentSugar(lender), \\
\quad\quad\quad CHILDAMT), CHILDAMT - agentSugar(borrower))\}) \\
\quad \land due = step + DURATION \\
\quad \land adjacent(position(lender), position(borrower)))
\end{array}
$$

$$
\begin{array}{l}
\underline{\quad MakeSpiceLoans \quad} \\
\Xi SpiceLattice \\
\Delta SpiceAgents \\
\Xi Step \\
\hline
population' = population \land sex' = sex \\
position' = position \land vision' = vision \\
age' = age \land maxAge' = maxAge \\
agentCulture' = agentCulture \\
agentImmunity' = agentImmunity \\
diseases' = diseases \land children' = children \\
metabolism' = metabolism \\
spiceMetabolism' = spiceMetabolism \\
agentSugar' = agentSugar \\
loanBook' = loanBook \\
initialSugar' = initialSugar \\
\mathbf{initialSpice'} = \mathbf{initialSpice} \\
\exists newLoans : \mathbb{P}(AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})));\\
\forall ag, lender, borrower : AGENT;\ amt, due : \mathbb{N} \bullet \\
spiceLoanBook' = spiceLoanBook \cup newLoans \\
ag \in \mathrm{dom}\, newLoans \Rightarrow \\
\qquad agentSpice'(ag) = agentSpice(ag) - totalLoaned(ag, newLoans) \\
ag \in \mathrm{dom}(\mathrm{ran}\, newLoans) \Rightarrow \\
\qquad agentSpice'(ag) = agentSpice(ag) + totalOwed(ag, newLoans) \\
ag \notin \mathrm{dom}(newLoans) \cup \mathrm{dom}(\mathrm{ran}\, newLoans) \Rightarrow \\
\qquad agentSpice'(ag) = agentSpice(ag) \\
willBorrow(age(ag), sex(ag), agentSpice'(ag), \\
\qquad \mathrm{ran}(loanBook' \cap \{a : AGENT \times (AGENT \times (\mathbb{N} \times \mathbb{N})) \\
\qquad\qquad\qquad \mid borrower(a) = borrower(loan)\})) \Rightarrow \\
\qquad\qquad \neg\ \exists ag2 : AGENT \bullet canLend(age(ag2), \\
\qquad\qquad\qquad sex(ag2), agentSpice'(ag2)) \\
\qquad\qquad\qquad \land\ adjacent(position(ag2), position(ag))
\end{array}
$$

$$\begin{array}{l}
totalLoaned(ag, newLoans) \leq \\
\quad amtAvail_{new}(age(ag), sex(ag), agentSpice(ag), \\
\qquad SPICECHILDAMT) \\
totalOwed(ag, newLoans) \leq SPICECHILDAMT - agentSpice(ag) \\
(lender, (borrower, (amt, due))) \in newLoans \Rightarrow \\
\quad (canLend(age(lender), sex(lender), agentSpice(lender)) \\
\quad \land willBorrow(age(borrower), sex(borrower), agentSpice(borrower), \\
\qquad \{borrower\} \lhd (\operatorname{ran} spiceLoanBook)) \\
\quad \land amt \leq min(\{amtAvail_{new}(age(lender), sex(lender), \\
\qquad\qquad agentSpice(lender), SPICECHILDAMT) \\
\qquad\quad SPICECHILDAMT - agentSpice(borrower))\}) \\
\quad \land due = step + DURATION \\
\quad \land adjacent(position(lender), position(borrower)))
\end{array}$$

C.4.18 Rule Application Sequence

$$\begin{array}{l}
Tick_{spice} \\
[\fatsemi Growback_{spice} \mid \fatsemi SeasonalGrowback_{spice}] \\
[\fatsemi Movement_{basicSpice} \mid \fatsemi (Movement_{pollutionSpice} \fatsemi PollutionDiffusion_{spice}) \mid \\
\fatsemi Combat_{spice}] \\
\{\fatsemi Inheritance_{spice}\}\{\fatsemi Death_{spice}[\fatsemi Replacement_{spice} \mid \fatsemi AgentMating_{spice}]\} \\
\{\fatsemi Culture\} \\
\{\fatsemi PaySugarLoans \fatsemi PaySpiceLoans \fatsemi MakeSugarLoans \fatsemi MakeSpiceLoans\} \\
\{\fatsemi Transmission_{spice} \fatsemi ImmuneResponse_{spice}\}\{\fatsemi Trade\}
\end{array}$$

C.5 Conclusions

We have shown that it is possible to apply formal methods fruitfully in the area of ABSS and produced a full formal specification of the Sugarscape family of simulations. It is, to the best of our knowledge, the first formal specification of the entire Sugarscape simulation family. The purpose of the specification is to provide a clear, unambiguous and precise definition of Sugarscape. The specification has identified many ambiguities and/or missing bits of information in the original rule definitions. Where there is an obvious way of removing these ambiguities we have done so. If there is more than one possible solution we have identified them and chosen the most likely one.

The issues with the model definition that we have encountered can broadly be grouped into three main types: Lack of Clarity, Missing Information and Sequential biases.

Lack of Clarity The rules, although simply stated in the appendix, lack clarity in their definition. Only one version of each rule is presented even when many variations are referred to in the text. The variations presented cannot always be used together, for example the Movement rule defined in the appendix is not the variant required if the pollution rule is also used. Our specification brings them all together in one place for ease of reference.

Missing Information Missing or incomplete information is the biggest cause for concern. In many cases we can work out the most likely answer based on context but in some cases there is not one definitive correct answer. If there was more than one arguably correct solution we chose the simplest. How we fill in these blanks can have a big effect on how the simulation proceeds. These effects may be important if we are trying to compare different implementations of Sugarscape. If we take the disease transmission rule, for example, questions that are unanswered include:

1. Once an agent gains immunity from a particular disease, do we remove that particular disease from the set of diseases that the agent is carrying, or is the agent still a carrier?
2. When we transmit a disease do we only transmit diseases that we carry and have no immunity for, or, can any disease we carry be transmitted?
3. The Mating rule omits important information about parents contributing half of their resources to their offspring. This has a huge effect on how mating works in a simulation.

By replacing each ambiguous interpretation with one simple and precise interpretation we allow different developers to replicate their results and benchmark them against each other. All hidden assumptions that could serve to advantage one implementation over another are excised.

Sequential Biases Sugarscape is based on the assumption that it will be implemented sequentially. While this may have been a good assumption at the time it was written it is not now necessarily the case. Improvements in processing speed have recently been attained mainly through the introduction of concurrency. Simulations are now almost always run on multicore or even multiprocessor machines.

The Z specification is free from assumptions about implementation. It achieves this without having to specify or constrain in any way what conflict resolution or avoidance strategies are employed. This leaves developers the freedom to try out different approaches as suits their implementation platform.

Because the specification is high level and only defines the before and after state of each rule it makes few assumptions as to how any rule will be implemented. All inherent biases towards a sequential implementation are removed. Implementers have complete freedom as to what programming model they employ (Object-oriented, imperative, functional, or any concurrent approach). Any simulations that adhere to the standard can be properly compared in terms of performance or patterns of behaviour. This will put on a firmer foundation any claims made by researchers about their implementations.

C.5.1 Further Work

Further work remains to be done in getting agreement from the ABM community on the decisions made in producing this interpretation of Sugarscape. Any incorrect assumptions made in removing ambiguities need to be identified and agreed upon. This provides a route to address the issues of replication of experimental results in ABSS.

Sugarscape can now be used as a benchmark (or rather set of benchmarks) for ABM implementers. This is particularly useful for those proposing new approaches to concurrency that promise performance improvements. Current trends, for example, include the use of GPUs [Deissenberg et al., 2008, Richmond et al., 2009], containing hundreds to thousands of individual processors. These approaches tend not to be tested on the more complex rules in Sugarscape (such as Combat, Inheritance and Trade) as they are not easily parallelized. By providing a precise and full set of these rules it is now possible for researchers to properly compare how different models cope with more complex and more realistic ABMs.

Z itself is rather verbose and can be hard to parse when reading. The lack of modularity made function definition signatures overly long. The available tools such as CZT [**?**] make the process of writing the specification easier but I have altered the specifications to remove bracketing where I thought it made the specification easier to read even if this was flagged as an error in the type checker. These issues could be overcome through the use of a variant of Z such as Object-Z or Alloy. The issue of whether ABM modellers would be willing to use formal specifications remains unknown.

There are differences between the outcomes of the synchronous and asynchronous approaches. Sugarscape assumes an asynchronous approach and this affects the style of specification that we use. We have shown in the case of combat the differences in a synchronous and asynchronous specification. While we regard the synchronous specification as somewhat simpler to produce but others may disagree. We tackle the question as to which approach is the more correct elsewhere.

Bibliography

[Akhter and Roberts, 2006] Akhter, S. and Roberts, J. (2006). *Multi-Core Programming, Increased Performance through Software Multi-threading*. Intel Press, first edition.

[Altaweel et al., 2010] Altaweel, M. R., Alessa, L. N., Kliskey, A., and Bone, C. (2010). A framework to structure agent-based modeling data for social-ecological systems. *Structure and Dynamics*, 4(1).

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA. ACM.

[Association, 2015] Association, E. S. (2015). Industry facts. http://www.theesa.com/about-esa/industry-facts/.

[Axelrod and Hamilton, 1981] Axelrod, R. and Hamilton, W. D. (1981). The evolution of cooperation.

[Axtell and Axtell, 2000] Axtell, R. and Axtell, R. L. (2000). Why agents? on the varied motivations for agent computing in the social sciences. In *Working Paper 17, Center on Social and Economic Dynamics, Brookings Institution*, page 17.

[Axtell et al., 2002] Axtell, R. L., Epstein, J. M., Dean, J. S., Gumerman, G. J., Swedlund, A. C., Harburger, J., Chakravarty, S., Hammond, R., Parker, J., and Parker, M. (2002). Population growth and collapse in a multiagent model of the kayenta anasazi in long house valley. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7275–7279.

[Ayguadé et al., 2009] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418.

[Bach* et al., 2003] Bach*, L. A., Sumpter, D. J., Alsner, J., and Loeschcke, V. (2003). Spatial evolutionary games of interaction among generic cancer cells. *Journal of Theoretical Medicine*, 5(1):47–58.

[Banks et al., 2005] Banks, J., ..., II, J. S. C., ..., Nelson, B. L., and [et al.], . (2005). *Discrete-event system simulation*. Prentice-Hall international series in industrial and systems engineering. Prentice Hall, Upper Saddle River, NJ.

[Ben, 2006] Ben, M. (2006). *Principles of Concurrent and Distributed Programming, Second Edition*. Addison-Wesley, second edition.

[Berryman, 2008] Berryman, M. (2008). Review of software platforms for agent based models. Technical report, DTIC Document.

[Bertolli et al., 2014] Bertolli, C., Antao, S. F., Eichenberger, A. E., O'Brien, K., Sura, Z., Jacob, A. C., Chen, T., and Sallenave, O. (2014). Coordinating GPU Threads for OpenMP 4.0 in LLVM. In *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, LLVM-HPC '14, pages 12–21, Piscataway, NJ, USA. IEEE Press.

[Bezbradica et al., 2014] Bezbradica, M., Ruskin, H. J., and Crane, M. (2014). Comparative analysis of asynchronous cellular automata in stochastic pharmaceutical modelling. *Journal of Computational Science*, 5(5):834–840.

[Bigbee et al., 2007] Bigbee, A., Cioffi-Revilla, C., and Luke, S. (2007). Replication of sugarscape using mason. In Terano, T., Kita, H., Deguchi, H., and Kijima, K., editors, *Agent-Based Approaches in Economic and Social Complex Systems IV: Post-Proceedings of The AESCS International Workshop 2005*, pages 183–190. Springer Japan, Tokyo.

[Black and Scholes, 1973] Black, F. and Scholes, M. (1973). The pricing of options and corporate liabilities. *Journal of political economy*, 81(3):637.

[Blake et al., 2009] Blake, G., Dreslinski, R., and Mudge, T. (2009). A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26 –37.

[Borkar et al., 2006] Borkar, S., Mulder, H., Dubey, P., Pawlowski, S., Kahn, K., Rattner, J., and Kuck, D. (2006). Platform 2015: Intel processor and platform evolution for the next decade. Intel White Paper.

[Brailsford, 2014] Brailsford, S. (2014). DES is alive and kicking. *J Simulation*, 8(1):1–8.

[Breshears, 2009] Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc.

[Briot et al., 1998] Briot, J.-P., Guerraoui, R., and Lohr, K.-P. (1998). Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329.

[Brooks, 1987] Brooks, Jr., F. P. (1987). No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19.

[Burstedde et al., 2001] Burstedde, C., Klauck, K., Schadschneider, A., and Zittartz, J. (2001). Simulation of pedestrian dynamics using a two-dimensional cellular automaton. *Physica A: Statistical Mechanics and its Applications*, 295(3):507–525.

[Campillo et al., 2012] Campillo, X. R., Cela, J. M., and Cardona, F. X. H. (2012). Simulating archaeologists? Using agent-based modelling to improve battlefield excavations. *Journal of Archaeological Science*, 39(2):347 – 356.

[Caron-Lormier et al., 2008] Caron-Lormier, G., Humphry, R. W., Bohan, D. A., Hawes, C., and Thorbek, P. (2008). Asynchronous and synchronous updating in individual-based models. *Ecological Modelling*, 212(34):522 – 527.

[Chapman et al., 2007] Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press.

[CHŃG, 2008] CHŃG, E. (2008). Using games engines for archaeological visualisation: Recreating lost worlds. In *11th International Conference on Computer Games*, CGAMES '07.

[Coakley et al., 2006] Coakley, S., Smallwood, R., and Holcombe, M. (2006). From molecules to insect communities-how formal agent based computational modelling is uncovering new biological facts. *Scientiae Mathematicae Japonicae*, 64(2):185–198.

[Cordasco et al., 2013] Cordasco, G., De Chiara, R., Raia, F., Scarano, V., Spagnuolo, C., and Vicidomini, L. (2013). Designing computational steering facilities for distributed agent based simulations. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 385–390, New York, NY, USA. ACM.

[Cornforth et al., 2005] Cornforth, D., Green, D. G., and Newth, D. (2005). Ordered asynchronous processes in multi-agent systems. *Physica D: Nonlinear Phenomena*, 204(12):70 – 82.

[Crooks et al., 2008] Crooks, A., Castle, C., and Batty, M. (2008). Key challenges in agent-based modelling for geo-spatial simulation. *Computers, Environment and Urban Systems*, 32(6):417 – 430. GeoComputation: Modeling with spatial agents.

[Cui and Shi, 2011] Cui, X. and Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, 11(1):125–130.

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55.

[Davidsson, 2000] Davidsson, P. (2000). Multi agent based simulation: Beyond social simulation. In *3. Workshop on Multi Agent Based Simulation (MABS) 2000, LNAI*, pages 97–107. Springer.

[Davidsson, 2002] Davidsson, P. (2002). Agent based social simulation: a computer science view. *Journal of Artificial Societies and Social Simulation*, 5(1).

[Dawkins, 1976] Dawkins, R. (1976). *The Selfish Gene*. Oxford University Press, Oxford, UK.

[Dean et al., 2000] Dean, J. S., Gumerman, G. J., Epstein, J. M., Axtell, R. L., Swedlund, A. C., Parker, M. T., and McCarroll, S. (2000). Understanding anasazi culture change through agent-based modeling. *Dynamics in human and primate societies: Agent-based modeling of social and spatial processes*, pages 179–205.

[Deissenberg et al., 2008] Deissenberg, C., van der Hoog, S., and Dawid, H. (2008). EURACE: A massively parallel agent-based model of the European economy. *Applied Mathematics and Computation*, 204(2):541 – 552. Special Issue on New Approaches in Dynamic Optimization to Assessment of Economic and Environmental Systems.

[Dictionary.com, 2015] Dictionary.com (2015). Dictionary.com unabridged.

[Dijkstra, nd] Dijkstra, E. W. (n.d.). Over de sequentialiteit van procesbeschrijvingen. circulated privately.

[Downey, 2005] Downey, A. (2005). *The Little Book of Semaphores*. Green Tea Press.

[Drummond, 2009] Drummond, C. (2009). Replicability is not reproducibility: Nor is it good science.

[D'Souza et al., 2009] D'Souza, R. M., Lysenko, M., Marino, S., and Kirschner, D. (2009). Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 21. Society for Computer Simulation International.

[D'Souza et al., 2007] D'Souza, R. M., Lysenko, M., and Rahmani, K. (2007). SugarScape on steroids: simulating over a million agents at interactive rates. *Proceedings of Agent2007*.

[Edmonds and Hales, 2003] Edmonds, B. and Hales, D. (2003). Replication, replication and replication: Some hard lessons from model alignment. *Journal of Artificial Societies and Social Simulation*, 6(4).

[Epstein, 1999] Epstein, J. M. (1999). Agent-based computational models and generative social science. *Complex.*, 4(5):41–60.

[Epstein, 2006] Epstein, J. M. (2006). Remarks on the foundations of agent-based generative social science. In Tesfatsion, L. and Judd, K. L., editors, *Handbook of Computational Economics*, volume 2, chapter 34, pages 1585–1604. Elsevier, 1 edition.

[Epstein, 2008] Epstein, J. M. (2008). Why model? *Journal of Artificial Societies and Social Simulation*, 11(4):12.

[Epstein and Axtell, 1996] Epstein, J. M. and Axtell, R. (1996). *Growing Artificial Societies: Social Science from the Bottom Up*. The Brookings Institution, Washington, DC, USA.

[Epstein et al., 2002a] Epstein, J. M., Cummings, D. A. T., Chakravarty, S., Singa, R. M., and Burke, D. S. (2002a). Toward a containment strategy for smallpox bioterror: An individual-based computational approach. In *Complexity*, page 4157. Press.

[Epstein et al., 2002b] Epstein, J. M., Steinbruner, J. D., and Parker, M. T. (2002b). Modeling civil violence: An agent-based computational approach. In *Proceedings of the National Academy of Science of the U.S.A. 99, Suppl. 3: 72437250. Available online*.

[Fatès, 2013] Fatès, N. (2013). A guided tour of asynchronous cellular automata. In Kari, J., Kutrib, M., and Malcher, A., editors, *Automata*, volume 8155 of *Lecture Notes in Computer Science*, pages 15–30. Springer.

[Fatès and Chevrier, 2010] Fatès, N. and Chevrier, V. (2010). How important are updating schemes in multi-agent systems? an illustration on a multi-turmite model. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 533–540. International Foundation for Autonomous Agents and Multiagent Systems.

[Ferber and Müller, 1996] Ferber, J. and Müller, J.-P. (1996). Influences and reaction: a model of situated multiagent systems. In *Proceedings of Second International Conference on Multi-Agent Systems (ICMAS-96)*, pages 72–79.

[Galan et al., 2009] Galan, J. M., Izquierdo, L. R., Izquierdo, S. S., Santos, J. I., Olmo, R. D., Lopez-Paredes, A., and Edmonds, B. (2009). Errors and artefacts in agent-based modelling.

[Gardner, 1970] Gardner, M. (1970). Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, pages 120–123.

[Gilbert, 2004] Gilbert, N. (2004). Agent-based social simulation: dealing with complexity. *The Complex Systems Network of Excellence*, 9(25):1–14.

[Gilbert, 2014] Gilbert, N. (2014). private communication.

[Gilbert and Terna, 2000] Gilbert, N. and Terna, P. (2000). How to build and use agent-based models in social science. *Mind & Society*, 1(1):57–72.

[Girault and Raviart, 1986] Girault, V. and Raviart, P.-A. (1986). *Finite element methods for Navier-Stokes equations : theory and algorithms*. Springer series in computational mathematics. Springer-Verlag, Berlin, New York. Extended version of : Finite element approximation of the Navier-Stokes equations.

[Gomila, 2015] Gomila, L. (2015). Simple and fast multimedia library.

[Graham et al., 2003] Graham, R., McCabe, H., and Sheridan, S. (2003). Pathfinding in computer games. *ITB Journal*, 8:57–81.

[Greene and Knuth, 2007] Greene, D. and Knuth, D. (2007). *Mathematics for the Analysis of Algorithms*. Modern Birkhäuser Classics. Birkhäuser Boston.

[Grilo and Correia, 2011] Grilo, C. and Correia, L. (2011). Effects of asynchronism on evolutionary games. *Journal of Theoretical Biology*, 269(1):109 – 122.

[Grimm et al., 2006] Grimm, V., Berger, U., Bastiansen, F., Eliassen, S., Ginot, V., Giske, J., Goss-Custard, J., Grand, T., Heinz, S. K., Huse, G., Huth, A., Jepsen, J. U., Jrgensen, C., Mooij, W. M., Mller, B., Peer, G., Piou, C., Railsback, S. F., Robbins, A. M., Robbins, M. M., Rossmanith, E., Rger, N., Strand, E., Souissi, S., Stillman, R. A., Vab, R., Visser, U., and DeAngelis, D. L. (2006). A standard protocol for describing individual-based and agent-based models. *Ecological Modelling*, 198(12):115 – 126.

[Grimm et al., 2010] Grimm, V., Berger, U., DeAngelis, D. L., Polhill, J. G., Giske, J., and Railsback, S. F. (2010). The {ODD} protocol: A review and first update. *Ecological Modelling*, 221(23):2760 – 2768.

[Gumerman et al., 2003] Gumerman, G. J., Swedlund, A. C., Dean, J. S., and Epstein, J. M. (2003). The evolution of social behavior in the prehistoric american southwest. *Artificial Life*, 9(4):435–444.

[Heath et al., 2009] Heath, B. L., Hill, R. R., and Ciarallo, F. W. (2009). A survey of agent-based modeling practices (january 1998 to july 2008). *J. Artificial Societies and Social Simulation*, 12(4).

[Hejlsberg et al., 2003] Hejlsberg, A., Wiltamuth, S., and Golde, P. (2003). *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Helbing, 2011] Helbing, D. (2011). *How to Do Agent-Based Simulations in the Future: From Modeling Social Mechanisms to Emergent Phenomena and Interactive Systems Design*, chapter Agent Based Modelling, pages 25–70. Springer.

[Herlihy and Shavit, 2012] Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Hernandez et al., 1994] Hernandez, J., de Miguel, P., Barrena, M., Martinez, J., Polo, A., and Nieto, M. (1994). Parallel and distributed programming with an actor-based language. In *Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on*, pages 420 –427.

[Hinkelmann et al., 2011] Hinkelmann, F., Murrugarra, D., Jarrah, A., and Laubenbacher, R. (2011). A mathematical framework for agent based models ofcomplex biological networks. *Bulletin of Mathematical Biology*, 73(7):1583–1602.

[Hinsen, 2011] Hinsen, K. (2011). A data and code model for reproducible research and executable papers. *Procedia Computer Science*, 4(0):579 – 588. Proceedings of the International Conference on Computational Science, {ICCS} 2011.

[Hinsen, 2014] Hinsen, K. (2014). Computational science: shifting the focus from tools to models.

[Hinsen, 2015] Hinsen, K. (2015). Activepapers: a platform for publishing and archiving computer-aided research. *F1000Research*, 3.

[Hirth and McCullough, 1977] Hirth, D. H. and McCullough, D. R. (1977). Evolution of alarm signals in ungulates with special reference to white-tailed deer. *The American Naturalist*, 111(977):pp. 31–42.

[Hoare, 1961] Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321.

[Holcombe, 1988] Holcombe, M. (1988). X-machines as a basis for dynamic system specification. *Software Engineering Journal*, 3(2):69–76.

[Huberman and Glance, 1993] Huberman, B. A. and Glance, N. S. (1993). Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences*, 90(16):7716–7718.

[Hummel et al., 2012] Hummel, A., Kern, H., Khne, S., and Dhler, A. (2012). An agent-based simulation of viral marketing effects in social networks. *26th European Simulation and Modelling Conference - ESM'2012*, pages 212–219.

[Huss-Lederman et al., 1996] Huss-Lederman, S., Jacobson, E. M., Johnson, J. R., Tsao, A., and Turnbull, T. (1996). *Strassen's Algorithm for Matrix Multiplication: Modeling, Analysis, and Implementation*. Citeseer.

[Inchiosa and Parker, 2002] Inchiosa, M. E. and Parker, M. T. (2002). Overcoming design and development challenges in agent-based modeling using ascape. *Proceedings of the National Academy of Sciences*, 99(suppl 3):7304–7308.

[J. Held and Koehl, 2006] J. Held, J. B. and Koehl, S. (2006). From a few cores to many: A tera scale computing research review. Intel White Paper.

[Jackson, 2006] Jackson, D. (2006). *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[Karbowski, 2008] Karbowski, A. (2008). Amdahl's and gustafson-barsis laws revisited. *CoRR*, abs/0809.1177.

[Kehoe, 2015a] Kehoe, J. (2015a). The specification of sugarscape. `http://arxiv.org/abs/1505.06012`.

[Kehoe, 2015b] Kehoe, J. (2015b). Synchronous sugarscape: A reference implementation. `https://github.com/josephkehoe/Sugarscape`. Accessed: 2015-12-30.

[Kehoe, 2016a] Kehoe, J. (2016a). Creating reproducible agent based models using formal methods. In *17th International Workshop on Multi-Agent-Based Simulation*. Springer.

[Kehoe, 2016b] Kehoe, J. (2016b). Robust reproducibility of agent based models. In *The European Simulation and Modelling Conference*. Inderscience.

[Kehoe and Morris, 2011] Kehoe, J. and Morris, J. (2011). A concurrency model for game scripting. In *12th International Conference on Intelligent Games and Simulation*, pages 10–16. EUROSIS.

[Kehoe and Morris, 2013] Kehoe, J. and Morris, J. (2013). Scalable parallelism in games. Sixth York Doctoral Symposium.

[Kitchin, 2015] Kitchin, J. R. (2015). Examples of effective data sharing in scientific publishing. *ACS Catalysis*, 5(6).

[Klaiber and Levy, 1994] Klaiber, A. and Levy, H. (1994). A comparison of message passing and shared memory architectures for data parallel programs. In *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 94 –105.

[Kleiner, 2005] Kleiner, A. (2005). Game AI: The shrinking gap between computer games and ai systems. *Ambient Intelligence: The Evolution of Technology, Communication and Cognition Towards the Future of Human-Computer Interaction*, 6:143–155.

[Knuth, 1976] Knuth, D. E. (1976). Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24.

[Lake, 2014] Lake, M. (2014). Trends in archaeological simulation. *Journal of Archaeological Method and Theory*, 21(2):258–287.

[Langton, 1986] Langton, C. G. (1986). Studying artificial life with cellular automata. *Phys. D*, 2(1-3):120–149.

[Lee, 2000] Lee, E. (2000). What's ahead for embedded software? *Computer*, 33(9):18–26.

[Lee et al., 2004] Lee, J., Adachi, S., Peper, F., and Morita, K. (2004). Asynchronous game of life. *Physica D: Nonlinear Phenomena*, 194(34):369 – 384.

[Lespérance et al., 1996] Lespérance, Y., Levesque, H. J., Lin, F., Marcu, D., Reiter, R., and Scherl, R. B. (1996). Foundations of a logical approach to agent programming. In *Intelligent Agents II Agent Theories, Architectures, and Languages*, pages 331–346. Springer.

[Li and O'Donoghue, 2013] Li, J. and O'Donoghue, C. (2013). A survey of dynamic microsimulation models: uses, model structure and methodology. *International Journal of Microsimulation*, 6(2):3–55.

[Little, 1961] Little, J. D. C. (1961). A proof for the queuing formula: L= λ w. *Operations Research*, 9(3):pp. 383–387.

[Lotka, 1909] Lotka, A. J. (1909). Contribution to the theory of periodic reactions. *The Journal of Physical Chemistry*, 14(3):271–274.

[Luke et al., 2005] Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527.

[Lysenko and D'Souza, 2008] Lysenko, M. and D'Souza, R. (2008). A framework for megascale agent based model simulations on graphics processing units. *J. Artificial Societies and Social Simulation*, 11(4).

[Macal and North, 2009] Macal, C. and North, M. (2009). Agent-based modeling and simulation. In *Simulation Conference (WSC), Proceedings of the 2009 Winter*, pages 86–98.

[Majeski, 1984] Majeski, S. J. (1984). Arms races as iterated prisoner's dilemma games. *Mathematical Social Sciences*, 7(3):253 – 266.

[Malik et al., 2013] Malik, A., Sharma, A., and Saroha, V. (2013). Greedy algorithm. *International Journal of Scientific and Research Publications (IJSRP)*, 3(8):1–5.

[Matsumoto and Nishimura, 1998] Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30.

[May, 1973] May, R. M. R. M. (1973). *Stability and complexity in model ecosystems*. Monographs in population biology. Princeton, N.J. Princeton University Press.

[Mazzaferro and Morciano, 2008] Mazzaferro, C. and Morciano, M. (2008). CAPP_DYN: A dynamic microsimulation model for the italian social security system. Department of Economics 0595, University of Modena and Reggio E., Faculty of Economics Marco Biagi.

[McCool et al., 2012] McCool, M., Reinders, J., and Robison, A. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[Murray, 2002] Murray, J. D. (2002). *Mathematical biology. I. , An introduction*. Interdisciplinary applied mathematics. Springer, New York.

[NAG, 2006] NAG, C. (2006). g05–random number generators.

[Nareyek, 2004] Nareyek, A. (2004). AI in computer games. *Queue*, pages 59–65.

[Nau et al., 1999] Nau, D., Cao, Y., Lotem, A., and Munoz-Avila, H. (1999). SHOP: Simple hierarchical ordered planner. Technical Report CS-TR-3981, UMIACS-TR-9904, Department of Computer Science and Institute for Systems Research, University of Maryland.

[Nehaniv, 2003] Nehaniv, C. L. (2003). Asynchronous automata networks can emulate any synchronous automata network. *Journal of Algebra*, pages 1–21.

[Newth and Cornforth, 2009] Newth, D. and Cornforth, D. (2009). Asynchronous spatial evolutionary games. *Biosystems*, 95(2):120 – 129.

[North et al., 2013a] North, M., Collier, N., Ozik, J., Tatara, E., Macal, C., Bragen, M., and Sydelko, P. (2013a). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1).

[North et al., 2013b] North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., and Sydelko, P. (2013b). Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1):1–26.

[North et al., 2006] North, M. J., Collier, N. T., and Vos, J. R. (2006). Experiences creating three implementations of the repast agent modeling toolkit. *ACM Trans. Model. Comput. Simul.*, 16(1):1–25.

[Nowak and May, 1992] Nowak, M. A. and May, R. M. (1992). Evolutionary games and spatial chaos. *Nature*, 359(6398):826–829.

[Nvidia, 2011] Nvidia (2011). *Fermi Compute Architecture Whitepaper*.

[Onggo, 2010] Onggo, B. S. (2010). Running agent-based models on a discrete-event simulator. In *Proceedings of the 24th European Simulation and Modelling Conference*, pages 25–27. Citeseer.

[OpenMP Architecture Review Board, 2013] OpenMP Architecture Review Board (2013). OpenMP application program interface version 4.0.

[Orkin, 2005] Orkin, J. (2005). Agent architecture considerations for real-time planning in games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*.

[Orkin, 2006] Orkin, J. (2006). Three states and a plan: The AI of F.E.A.R. *Proceedings of the Game Developer's Conference (GDC)*.

[Park and Fishwick, 2010] Park, H. and Fishwick, P. A. (2010). A gpu-based application framework supporting fast discrete-event simulation. *Simulation*, 86(10):613–628.

[Parker and Epstein, 2011] Parker, J. and Epstein, J. M. (2011). A distributed platform for global-scale agent-based models of disease transmission. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 22(1):2.

[Parker, 2001] Parker, M. T. (2001). What is ascape and why should you care? *Journal of Artificial Societies and Social Simulations*, 4(1).

[Parunak et al., 1998] Parunak, H. V. D., Savit, R., and Riolo, R. L. (1998). Agent-based modeling vs. equation-based modeling: A case study and users' guide. In *Proceedings of the First International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pages 10–25, London, UK, UK. Springer-Verlag.

[Peng, 2011] Peng, R. D. (2011). Reproducible research in computational science. *Science (New York, Ny)*, 334(6060):1226–1227.

[Perumalla, 2006] Perumalla, K. S. (2006). Discrete-event execution alternatives on general purpose graphical processing units (gpgpus). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, PADS '06, pages 74–81, Washington, DC, USA. IEEE Computer Society.

[Perumalla and Aaby, 2008] Perumalla, K. S. and Aaby, B. G. (2008). Data parallel execution challenges and runtime performance of agent simulations on gpus. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, pages 116–123, San Diego, CA, USA. Society for Computer Simulation International.

[Pollacia, 1989] Pollacia, L. F. (1989). A survey of discrete event simulation and state-of-the-art discrete event languages. *SIGSIM Simul. Dig.*, 20(3):8–25.

[Poundstone, 1992] Poundstone, W. (1992). *Prisoner's Dilemma: John Von Neumann, Game Theory and the Puzzle of the Bomb*. Doubleday, New York, NY, USA, 1st edition.

[Radax and Rengs, 2010] Radax, W. and Rengs, B. (2010). Timing matters: Lessons From The CA Literature On Updating. *Third Word Congress of Social Simulation*, abs/1008.0941.

[Railsback et al., 2005] Railsback, S., Lytinen, S., and Grimm, V. (2005). Stupid-model and extensions: A template and teaching tool for agent-based modeling platforms.

[Railsback et al., 2006a] Railsback, S. F., Lytinen, S. L., and Jackson, S. K. (2006a). Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609–623.

[Railsback et al., 2006b] Railsback, S. F., Lytinen, S. L., and Jackson, S. K. (2006b). Agent-based simulation platforms: Review and development recommendations. *Simulation*, 82(9):609–623.

[Rank, 2010] Rank, S. (2010). Docking agent-based simulation of collective emotion to equation-based models and interactive agents. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 6:1–6:8, San Diego, CA, USA. Society for Computer Simulation International.

[Reynolds, 1987] Reynolds, C. (1987). Flocks, herds, and schools: A distributed behavioural model. In *Computer Graphics - SIGGRAPH*, volume 21, pages 25–34.

[Richmond et al., 2009] Richmond, P., Coakley, S., and Romano, D. M. (2009). A high performance agent based modelling framework on graphics card hardware with CUDA. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '09, pages 1125–1126, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

[Richmond et al., 2010] Richmond, P., Walker, D., Coakley, S., and Romano, D. (2010). High performance cellular level agent-based simulation with flame for the gpu. *Briefings in Bioinformatics*, 11(3):334–347.

[Rudowsky, 2004] Rudowsky, I. (2004). Intelligent agents. *The Communications of the Association for Information Systems*, 14(1):48.

[Russell and Cohn, 2012] Russell, J. and Cohn, R. (2012). *Hamming Distance*. Book on Demand.

[Ruxton and Saravia, 1998] Ruxton, G. D. and Saravia, L. A. (1998). The need for biological realism in the updating of cellular automata models. *Ecological Modelling*, 107(23):105 – 112.

[Sahay et al., 2014] Sahay, N., Ierapetritou, M., and Wassick, J. (2014). Synchronous and asynchronous decision making strategies in supply chains. *Computers and Chemical Engineering*, 71:116 – 129.

[Sansores and Pavn, 2005] Sansores, C. and Pavn, J. (2005). Agent-based simulation replication: A model driven architecture approach. In Gelbukh, A. F., de Albornoz, A., and Terashima-Marn, H., editors, *MICAI*, volume 3789 of *Lecture Notes in Computer Science*, pages 244–253. Springer.

[Sarkar, 2000] Sarkar, P. (2000). A brief history of cellular automata. *ACM Comput. Surv.*, 32(1):80–107.

[Sato et al., 2014] Sato, M., Tsutsui, S., Fujimoto, N., Sato, Y., and Namiki, M. (2014). First results of performance comparisons on many-core processors in solving qap with aco: Kepler gpu versus xeon phi. In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation Companion*, GECCO Comp '14, pages 1477–1478, New York, NY, USA. ACM.

[Schelling, 1971] Schelling, T. (1971). Dynamic models of segregation. *Journal of Mathematical Sociology*, (1).

[Schönfisch and de Roos, 1999] Schönfisch, B. and de Roos, A. (1999). Synchronous and asynchronous updating in cellular automata. *Biosystems*, 51(3):123 – 143.

[Seok and Kim, 2012] Seok, M. G. and Kim, T. G. (2012). Parallel discrete event simulation for devs cellular models using a gpu. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 11:1–11:7, San Diego, CA, USA. Society for Computer Simulation International.

[Shopf et al., 2008] Shopf, J., Barczak, J., Oat, C., and Tatarchuk, N. (2008). March of the froblins: Simulation and rendering massive crowds of intelligent and detailed creatures on gpu. In *ACM SIGGRAPH 2008 Games*, SIGGRAPH '08, pages 52–101, New York, NY, USA. ACM.

[Siebers et al., 2010] Siebers, P.-O., Macal, C. M., Garnett, J., Buxton, D., and Pidd, M. (2010). Discrete-event simulation is dead, long live agent-based simulation! *J. Simulation*, 4(3):204–210.

[Silva et al., 2010] Silva, A. R. D., Lages, W. S., and Chaimowicz, L. (2010). Boids that see: Using self-occlusion for simulating large groups on gpus. *Comput. Entertain.*, 7(4):51:1–51:20.

[Smith, 2012] Smith, G. (2012). *The Object-Z specification language*, volume 1. Springer Science & Business Media.

[Šperka and Spišák, 2012] Šperka, R. and Spišák, M. (2012). Tobin tax introduction and risk analysis in the Java simulation. In *Proceedings of the 30th International Conference Mathematic Methods in Economics*.

[Spicher et al., 2010] Spicher, A., Fatès, N., and Simonin, O. (2010). Translating discrete multi-agents systems into cellular automata: Application to diffusion-limited aggregation. In Filipe, J., Fred, A., and Sharp, B., editors, *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 270–282. Springer Berlin Heidelberg.

[Spivey, 1989] Spivey, J. M. (1989). *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Sutter and Larus, 2005] Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3(7):54–62.

[Sweeney, 2006] Sweeney, T. (2006). The next mainstream programming language: a game developer's perspective. *SIGPLAN Not.*, 41(1):269–269.

[Tang et al., 2009] Tang, H., Tan, C., Tan, K., and Tay, A. (2009). Neural network versus behavior based approach in simulated car racing game. In *Evolving and Self-Developing Intelligent Systems, 2009. ESDIS'09. IEEE Workshop on*, pages 58–65. IEEE.

[Tisue, 2004] Tisue, S. (2004). Netlogo: Design and implementation of a multi-agent modeling environment. In *In Proceedings of Agent 2004*, pages 7–9.

[Tisue and Wilensky, 2004] Tisue, S. and Wilensky, U. (2004). Netlogo: A simple environment for modeling complexity. In *in International Conference on Complex Systems*, pages 16–21.

[Topping et al., 2010] Topping, C. J., Hye, T. T., and Olesen, C. R. (2010). Opening the black boxdevelopment, testing and documentation of a mechanistically rich agent-based model. *Ecological Modelling*, 221(2):245 – 255.

[Troitzsch, 2009] Troitzsch, K. G. (2009). Perspectives and challenges of agent-based simulation as a tool for economics and other social sciences. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 35–42. International Foundation for Autonomous Agents and Multiagent Systems.

[Valiant, 1990] Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111.

[Valve and Gearbox, 1998] Valve and Gearbox (1998). Cryengine. http://half-life2.com/.

[van Dam et al., 2007] van Dam, K. H., Verwater-Lukszo, Z., Ferreira, L., and Sirikijpanichkul, A. (2007). Planning the location of intermodal freight hubs: an agent based approach. In *ICNSC*, pages 187–192. IEEE.

[van Heesch, 2015] van Heesch, D. (2015). Doxygen.

[von Neumann, 1951] von Neumann, J. (1951). The general and logical theory of automata. In Jeffress, L. A., editor, *Cerebral Mechanisms in Behaviour*. Wiley.

[Zaft and Zeigler, 2002] Zaft, G. C. and Zeigler, B. P. (2002). Discrete event simulation and social science: The xeriscape artificial society. *Post Conference Proceedings of the Eighth World Multi-Conference on Systemics, Cybernetics and Informatics/International Conference on Information Systems, Analysis and Synthesis (SCI 2002/ISAS 2002)*, 6.