Noname manuscript No.
(will be inserted by the editor)

Applying Synchronous Updating to Agent Based Simulations

A. N. Author

Received: date / Accepted: date

Abstract Although synchronous updating is used extensively in Cellular Automata it is rarely used in Agent Based Models. This is due to the lack of any synchronous updating algorithm that can handle the range and complexity of interactions present in Agent Based Models. In this paper present novel synchronous updating algorithms that can handle the full range of interactions present in Agent Based Models. We show that these algorithms are competitive in space and time with asynchronous algorithms. These algorithms now allow comparisons to be made between synchronous and asynchronous implementations of any ABM.

Keywords Asynchronous Updating · Synchronous Updating · Agent Based Models · Concurrency · Asymptotic Analysis

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 represents the key characteristics or behaviours of the system. An Agent Based Model (ABM) is one where individuals (agents) within the system and their interactions are explicitly represented. Global properties are not explicitly modelled but emerge from the local interactions between agents.

The asynchronous approach to simulating ABMs of the real world is espoused in [20]. This paper distinguishes between Synchronous Updating (SU) and Asynchronous Updating (AU) and defines the differences between them.

The synchronous approach is defined as having a global clock that synchronises the updating of all agent states so that all updates occur in unison. Asynchronous approaches, on the other hand, have no global clock and updates of agent state do not occur simultaneously. In effect AU means that the agent interactions are applied sequentially in some random order. There are various approaches that determine how this random ordering is decided [9]. It has been demonstrated that synchronous and asynchronous implementations of the same simulation can result in widely differing

Address(es) of author(s) should be given

behaviours. Since it is assumed that real world systems are comprised of agents that do not act in unison it is also assumed that the AU is more realistic in that it correlates more closely with the reality being simulated. 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.

Although it is the case that AU and SU can give different results, it has not been proven that AU is more accurate. We show two important differences between AU and SU. Then we produce alternative novel SU algorithms with comparable space-time complexity to AU algorithms. These SU algorithms can handle the full range of interactions that occur in ABM. The practicality of these algorithms is demonstrated by using them to produce a SU implementation of Sugarscape, a well known and complex ABM.

1.1 DES versus ABM

Although we are concerned here with ABM the issues concerning synchronous and asynchronous implementation are also applicable to Discrete Event Simulations (DES). There is debate about the relationship between ABM and DES. [5] states that DES is a subset of ABM while [32] claims that any ABM can be translated into an equivalent DES. In any case the two approaches have much in common [38] and the issue of synchronous or asynchronous implementation of events is relevant to both approaches.

1.2 Outline of Article

In the next section we will compare and contrast AU and SU highlighting the differences between them.

In section three we identify two benefits of SU, the preservation of the principle of bounded rationality and the ability to explicitly express explicit collision resolution rules.

Section four presents our SU algorithms and using asymptotic analysis gives space-time complexity measures for these algorithms (both for sequential and concurrent implementations). We show that the complexity of the SU algorithm is comparable to the complexity of the AU algorithm.

In Section five we demonstrate the practicality of these algorithms by showing the results of implementing the Sugarscape ABM with SU.

Section six reviews related work on introducing SU into ABM and contrasts these approaches to our approach. Finally we finish with a summary of our conclusions and briefly state how our work will progress in the future.

2 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 [15].

Using the synchronous approach, Nowak developed a synchronous simulation of spatial Iterated Prisoners' Dilemma [31] and showed that the simulation generates chaotically changing spatial patterns, in which *cooperators* and *defectors* both persist indefinitely. Huberman and Glance [20] 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*. Thus we have two clearly contradictory results deriving from the application of the same rule and differing only in the updating technique.

The conclusion drawn by [20] is 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. [20]

In line with this, [8] claims that AU is more realistic (i.e. more closely resembles the real world). Similarly other researchers claim that synchronous behaviour is rare in the real world [9] and that AU is more realistic [29]. On the other hand [13] states that AU is suitable only for instantaneous events which do not occur in biological systems.

The precise ordering used by asynchronous updating (dependent on which AU scheme is employed) affects the outcomes as well [35,33]. This is often ignored in ABM [33].

It has become 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 [30,27,2]. Standard ABSSs, such as the canonical Sugarscape [12], assume an asynchronous implementation. The ability to execute a sequence of agent actions in a random order, an essential part of asynchronous simulations, forms part of *Stupid-Model* [34], a suite of models designed to test the suitability of any toolkit for ABM development.

The preference for AU has been accentuated by a lack of synchronous algorithms that can handle the complex interactions that occur in ABMs. This contrasts with Cellular Automata (CA) based simulation where, because the interactions are simpler, synchronous algorithms exist and have proven popular. CA-based Simulations of traffic flow, a real world system, employ synchronous updating [7] as standard. It is also interesting to note that it is not uncommon to see CA based simulations employ both updating methods and comparisons made from the results [3,1]. Even in CA, where these comparisons are more common, the effects of the two updating techniques are not well understood [16].

It is our contention that the case for choosing AU over SU has not been convincingly made and, given that the outcomes of a simulation depend on the approach taken, we need to reexamine this preference for AU in ABM. The new SU algorithms shown here provide modellers with the ability to run AU and SU based ABMs side by side for comparison.

3 Benefits of Synchronous Updating

3.1 Bounded Rationality

An essential property of ABSS [11] is the principle of bounded rationality: *an agent can only be aware of the state of other agents within its neighbourhood (locality).* This principle guarantees that an agent cannot be omniscient, that is, it cannot be aware of the global state or have complete knowledge of the universe. If an agent is not in my immediate neighbourhood I cannot know what it is doing now. I may find out at a later time what it was doing now through information diffusion (e.g. gossip) but that is time delayed information. The synchronous approach 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 consideration should be given to using SU.

Take 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 can become aware of this fact under two conditions:

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

Now once A dies the amount of time steps it takes for an 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 one agent to another over one or more subsequent steps.

Consider what happens in an asynchronous simulation. If we are an immediate neighbour of A then 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 before ours 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 about A's death 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 might also be aware of A's change of state instantly (within the same time step). We can construct a sequence $[a_1, a_2, ..., a_n]$ where a_i runs before a_{i+1} within this step and a_i is a neighbour of a_{i+1}. If a_1 dies then every agent in the chain can pass this information on to their neighbour instantly (within the same step). 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 [8]).

There are two related problems here:

1. Agents who are not within the locality of A will be immediately aware of the change of state 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 propagation).

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 the synchronous approach. Of note here are two findings:

1. May [28] 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 has as a cause 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;
2. High population densities enhance the differences in outcomes between asynchronous and synchronous versions of the same ABM [8]. 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 of *White Tailed Deer* herd behaviour [19]. 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 (all things being equal)[1]. 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 can 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 while the half of the herd furthest away are aware.

When considering which updating method to use we need to carefully consider how important bounded rationality (as enforced through locality) is in the system we are simulating and whether inconsistent speed of propagation of information through the simulation can adversely affect the outcomes.

3.2 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

[1] For simplicity we assume all deer are paying attention to their surroundings.

the winner will be the agent that was run first during the step. Since the order that agents run in within a step is determined randomly the winner will always be decided randomly[2].

A SU implementation, 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 similarly to the asynchronous case. But the possibility exists within the synchronous approach 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. This is a rational choice, more accurately models the real world and preserves determinacy. This facility to define collision resolution as part of the behaviour definition is not available with AU.

4 Synchronous Algorithms

4.1 Sugarscape

In order to demonstrate how our algorithms cope with the types of complex interaction that can occur in an ABSS we will use examples from Sugarscape, a canonical ABSS. The Sugarscape ABSS [12] was developed in order 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. All examples of agent interaction given here are taken from Sugarscape. The rules of agent interaction defined in Sugarscape cover a wide range of interaction types and complexity. This makes Sugarscape an ideal benchmark for demonstrating how each type of action can be represented synchronously. To this end we use a formal specification of Sugarscape [21] as our reference.

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 [41]. Thus any findings from investigating Sugarscape can also be more easily checked to see how they apply to DES.

4.2 Categories of Interaction

We have partitioned all behaviours into three different types: *Independent*, *Read-Dependent* and *Write-Dependent*. All interactions fall into one of these categories. We define separate synchronous algorithms for each category.

[2] or by whatever scheduling method is employed by the asynchronous approach. In any case it is determined not by the rule definition but by the choice of updating algorithm.

4.3 Independent Actions

In many cases actions can be expressed naturally in a synchronous manner. Rules that involve no interactions between agents[3] 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 asynchronous interpretation of an independent rule. As each agent performs its action in isolation the order of execution of the agents can have no effect on the outcome.

4.3.1 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 agent's new state (line 2).

Algorithm 1 Independent Action

```
Input: $Agents : list\langle agent \rangle$
1: for all $a \in Agents$ do
2:     $a.state \leftarrow applyRule(a)$
```

4.3.2 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 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. $\Theta(1)$).

4.3.3 Concurrency

The **for**-loop can be implemented as a parallel map and belongs to the class of *embarrassingly parallel* problems. With no communication required between agents the parallel algorithm is $\Theta(1)$ for both the synchronous and asynchronous algorithms.

4.3.4 Deadlock and Livelock

Four conditions are required to hold before deadlock is possible. If any of these conditions do not hold then deadlock is impossible.

Mutual Exclusion At least one resource must be held in a non-shareable mode;

[3] *Growback*, *SeasonalGrowback* and *PollutionFormation*

Hold and Wait or Resource Holding A process can hold one resource while simultaneously requesting additional resources held by another process;

No Preemption A process cannot be forced to release a resource it holds in non-sharable mode;

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, $P_1 \dots P_n$, such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3 and so on until P_N is waiting for a resource held by P_1.

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.

4.4 Read Dependent Actions

Rules where agents are required only to read the state of their neighbours [4] 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 while stated in an asynchronous manner can be given an equally plausible synchronous interpretation.

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 agents new state from the updating of the agents state, as we have previously done for *Independent actions*.

4.4.1 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 this calculated value. 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 $computeNeighbours$ (line 2 of Algorithm 2) takes in an agent a and the set of all agents in the simulation before returning only the neighbouring agents of a. The *applyRule* function (line 3 of Algorithm 2) now must take in the set of neighbours of agent a to help it compute the new state of agent a.

[4] $Culture$, $PollutionDiffusion$, $Inheritance$ and $DiseaseTransmission$

Algorithm 2 Synchronous Read Dependent Action

```
Input: $Agents : list\langle agent\rangle$
1: for all $a \in Agents$ do
2:     $neighbours \leftarrow computeNeighbours(a, Agents)$
3:     $a.newState \leftarrow applyRule(neighbours)$
4: for all $a \in Agents$ do
5:     $a.state \leftarrow a.newState$
```

4.4.2 Asymptotic Analysis

There are two loops in the algorithm. The second loop (lines 4–5) is obviously $\Theta(n)$, 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 ($\Theta(1)$). If we accept those assumptions then the sequence of two loops both of $\Theta(n)$ still gives an overall (upper and lower) bound of $\Theta(n)$.

In Sugarscape both assumptions hold for all rules. In any case the running time bounds will be 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 added 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 extra work done by the second loop is 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 $2Cn$ 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 holds for Read and Write Dependent actions.

4.4.3 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 synchronous algorithm can be implemented as two parallel maps occurring in sequence with each concurrent loop having a bound of $\Theta(1)$.

Algorithm 3 Asynchronous Read Dependent Action

```
Input: $Agents : list\langle agent\rangle$
1: for all $a \in Agents$ do
2:     $neighbours \leftarrow computeNeighbours(a, Agents)$
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 paralleled as the synchronous algorithm and requires either locking, with all that entails, or a restructuring of the algorithm to employ geometric tiling. We will show how this can be achieved in the next section.

4.4.4 Deadlock and Livelock

The synchronous approach does not perform updates to state until all agents have finished accessing the state of other agents. Therefore the reading of agent state can proceed without locking as can the updating of state. Without the need to perform any locking we can have no deadlock or livelock.

The issues of Deadlock and Livelock in the asynchronous algorithm will depend on how we implement concurrency.

4.5 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) and the members of a group cause mutual updates of each others state[5]. In these cases the rules can still be defined synchronously. The solution is not to create more time intervals but to break these rules into their smaller (atomic) parts. 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. 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.

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 [12]

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 try to 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

[5] *Credit*, *Combat*, *Trade*, *Mating*, *Replacement* and *Movement*

of a moving agent and its chosen destination location. These groupings can be determined synchronously but some explicit collision resolution (avoidance) rule is required to determine the outcome when two or more agents try to chose the same destination. 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. This gives us three advantages over the asynchronous approach (i) we are not forced to introduce randomness, (ii) we can pick from a number of different conflict resolution strategies and (iii) we make our resolution strategy explicit.

The second component of this rule is the state update phase. Because we have already formed exclusive groupings 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 but, as we have seen, the synchronous approach can exclude inaccurate outcomes allowed by the asynchronous approach.

A 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 independent of the order of agent action execution.

4.5.1 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 $formGroup$ takes in an agent a and the available neighbours of a, and, returns a proposed group containing a and some of its neighbours. If there is no available grouping then it returns a group containing only a. Each group has a weighting attached to it that signifies the ranking of that group. Higher ranking groups get precedence when determining the outcomes of conflict detection and 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. How the weighting is determined has to be explicitly stated by the rule that defines the action.

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 put in rank order based on their associated weighting (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 available agents remaining (**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 *applyGroupRule* 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: $AcceptedGroups \leftarrow \emptyset$                              ▷ Part I: Form Groups
 2: while $AvailableAgents \neq \emptyset$ do
 3:     $ProposedGroups \leftarrow \emptyset$
 4:     for all $a \in AvailableAgents$ do
 5:         $neighbours \leftarrow computeNeighbours(a, Agents)$
 6:         $g \leftarrow formGroup(a, neighbours)$
 7:         $ProposedGroups.add(g)$
 8:     $ProposedGroups.sort()$
 9:     for all $g \in ProposedGroups$ do                        ▷ in ranked order
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:             $AcceptedGroups.add(g)$
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:         $a.newState \leftarrow applyGroupRule(a, group)$
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 non-deterministic. 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: $AcceptedGroups \leftarrow \emptyset$
2: for all $a \in AvailableAgents$ do
3:     $neighbours \leftarrow computeNeighbours(a, Agents)$
4:     $group \leftarrow formGroup(a, neighbours)$
5:     for all $b \in group$ do
6:         $b.state \leftarrow applyGroupRule(b, group)$
```

4.5.2 Asymptotic Analysis

The loop applying the group rule (lines 18–20 in Algorithm 4) is $\Theta(n)$ as long as the function *applyGroupRule* 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 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 **while**-loop 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 assumed 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 ABMs. In an ABM 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.

4.5.3 Concurrency

A naive concurrent implementation of this algorithm will still be dominated by the sort routine. The loops in Part II and III (lines18–23) and the choosing of preferred groups (lines 4–7) can all be implemented using simple parallel maps giving constant running time. The selection of groups from the preferred list (lines 9–17) must be run sequentially and so remains $\Theta(n)$ while the sort routine will still remain close to $\Theta(n \log n)$. A naive concurrent implementation will therefore give us a runtime equal to that of the sequential implementation.

With a little more thought we can use geometric tiling to attain weak scalability and a $\Theta(1)$ concurrent algorithm (see Algorithm 6). To see how this is possible we will assume a 2-D $N \times N$ simulation where the neighbourhood of an agent A is defined as a $C_n \times C_n$ square centred on the agent. An agent A can only interact with other agents located within this neighbourhood. We can partition the simulation space into geometric tiles of size $3C_n \times 3C_n$. We assume for the sake of simplicity (and without loss of generality) that these tiles cover the entire $N \times N$ space (i.e. tile size divides evenly into the simulation space size). Each tile contains nine $C_n \times C_n$ blocks. An agent in one of these blocks can only interact with agents in its adjoining block. By ensuring that we never concurrently compute the updates for adjoining blocks we can updates the tiles concurrently. The $Sync()$ command (line 6) executed after each block is evaluated ensures that no two processes evaluate blocks with overlapping spheres of interest. The entire algorithm has nine sequential steps but each step has only a constant amount of work to do determined by neighbourhood size. Any overhead associated with communication between tiles is, of course, ignored by asymptotic analysis (as it is constant time).

This tiled approach alters the SU outcomes somewhat. While each tile is updated synchronously there will be an ordering imposed on interactions that occur between agents in adjoining tiles. It is locally synchronous (within tiles) but globally asynchronous (between tiles). Any simulation employing tiling must be prepared to accept this limitation.

Tiling will also work for the asynchronous algorithm (Algorithm 7). This algorithm is almost identical to the tiled synchronous algorithm with the only difference being that we update each block asynchronously (line 5) instead of synchronously.

Both the synchronous and asynchronous tiled algorithms have the same time complexity (that is, $\Theta(1)$).

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 $i \leftarrow 1 \dots 3$ do                          ▷ Sequential for loop
3:         for $k \leftarrow 1 \dots 3$ do                      ▷ Sequential for loop
4:             $AvailableAgents \leftarrow getAgentsInBlock(tile, i, k)$
5:             $ComputeSynchronousUpdate(AvailableAgents)$
6:             $Sync()$
```

Algorithm 7 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 $i \leftarrow 1 \dots 3$ do                          ▷ Sequential for loop
3:         for $k \leftarrow 1 \dots 3$ do                      ▷ Sequential for loop
4:             $AvailableAgents \leftarrow getAgentsInBlock(tile, i, k)$
5:             $ComputeAsynchronousUpdate(AvailableAgents)$
6:             $Sync()$
```

4.5.4 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 in either synchronous or asynchronous algorithms.

4.6 Synchronous DES

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 (1) rules are replaced by events and (2) 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 now, where now is defined as the time of the first event in the sorted *EventQueue* (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 the 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 ($now + \Delta T$).

A major advantage of implementing an ABM as a DES is increased efficiency as identified in [41]. When not every agent is required to apply a rule during every step then the amount of work required during each step is reduced.

Algorithm 8 Discrete Event Simulation Algorithm

```
Input: $EventQueue : SortedQueue\langle event\rangle$
Input: $AcceptedEventGroups, proposedEventGroups : List\langle(event, group)\rangle$
 1: $AcceptedEventGroups \leftarrow \emptyset$
 2: $currentEvents \leftarrow \emptyset$
 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:     $nextEvent = EventQueue.peek()$
 9: while $currentEvents \neq \emptyset$ do
10:     $ProposedEventGroups \leftarrow \emptyset$
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:         $applyEventRule(e, group)$
```

5 Implementing Synchronous Sugarscape

Sugarscape [12] is, as we noted in section 4.1, a heterogeneous ABM (more specifically an ABSS) consisting of 13 rules that range in complexity from simple read dependent rule (e.g. *Growback*) to the more complex read dependent (e.g. $PollutionDiffusion$) and write dependent (e.g. *Combat*). It consists of an $N \times N$ lattice (or grid) of locations upon which agents reside. In every location a food resource (known as sugar) grows up to some specified maximum amount. Each location can hold at most one agent at a time. Agents are mobile and can move across the lattice of locations in the four cardinal directions (north, south, east and west). When an agent arrives at a location it immediately consumes all the sugar at that location. Agents metabolise sugar during each step of the simulation and die if their sugar levels reach zero. Some of the rules of Sugarscape determine the rate of sugar growth (*Growback*, *SeasonalGrowback*) or pollution ($PollutionFormation$, $PollutionDiffusion$) at each location. The remaining rules determine how the agents move and interact with each other ($Movement$, *Combat*, *Disease*, *Culture*, *Reproduction*, etc.).

While the original version of Sugarscape assumes AU, the large range of rules employed in this ABSS makes it an excellent testbed for SU. The version of Sugarscape developed here is, to the best of our knowledge, the only SU implementation in existence and the most complex ABM implemented using SU. The simplest rule combination of Sugarscape ($Movement$ and *Growback*) is equal to the most complex rules used by the alternative SU algorithms in the next section and it is not clear how or if they could cope with any of the other *Write-dependent* rules of Sugarscape.

This implementation demonstrates that our algorithms can handle the full range of possible agent interactions in any ABM. It also enables us to make comparisons between the original AU implementation of Sugarscape and our SU implementation. There are issues of reproducibility associated with ABM [18,37,10] (sometimes known as The Replication Problem) in general and Sugarscape in particular [4]. In order to help overcome these issues we use a formal specification of Sugarscape [21, 22] developed in an effort to overcome the replication problem.

5.1 The Implementation

An Object-Oriented framework was developed in C++ that encoded the three different types of agent interaction. To implement any particular interaction type all that is required is that we inherit from the appropriate interaction type (IndependentAction, ReadDependentAction or WriteDependentAction) and implement two functions: *formGroup* and *executeAction*. *FormGroup* returns the set of agents that the current agent wants to interact with, e.g. a new location to move to or agents it wants to mate with. *ExecuteAction* apples the rule to everyone in a chosen group. All other issues, such as contention or concurrency, are handled transparently by the framework.

At present we have only fully developed the single resource version of the 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*).

5.2 Results

Due to ambiguities in the original Sugarscape definition 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 we replicated it and where there is ambiguity we have made a best guess of the ordinal intentions. However, even given these ambiguities we can still see if we can replicate the general emergent properties and trends in the simulation[6]..

One of the first emergent properties identified and measured was the carrying capacity[7] of the lattice. The graph showing the carrying capacity [12] has a distinctive shape and when we repeated the measurement we obtained a graph (see figure 1) that closely matched the original.

Similarly when we measured the effects of varying agent metabolism (sugar consumption rate) and vision[8] our results (see figure 2) also mirror those of the original.

[6] Trends in population growth, spread of culture over time, etc.

[7] The number of agents the lattice can support.

[8] Vision determines how far in the cardinal directions agents can see and move in a single step.

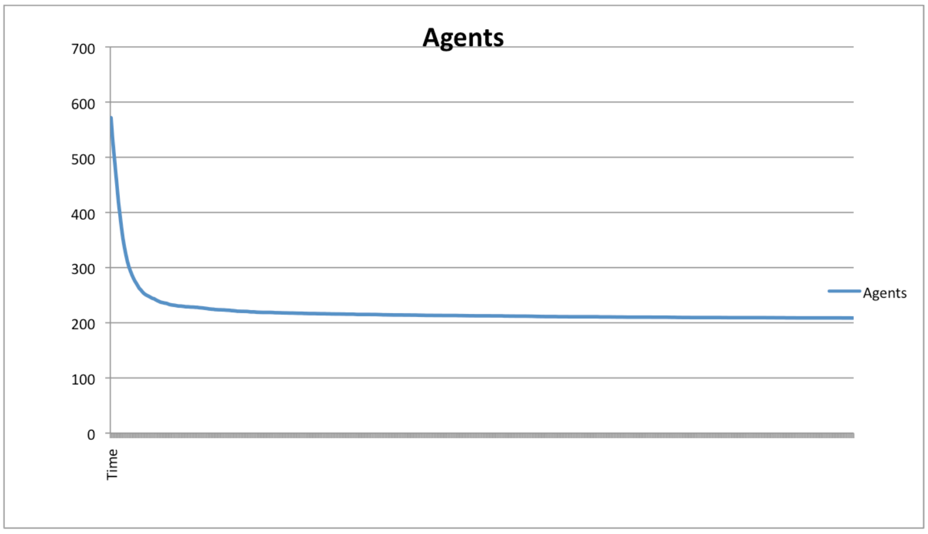

Fig. 1 Lattice Carrying Capacity

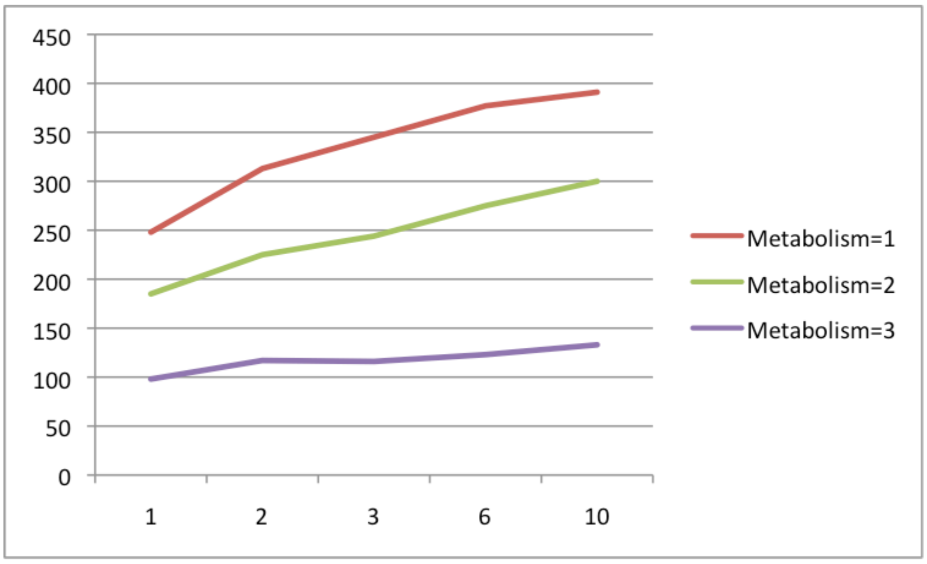

Fig. 2 Carrying Capacity with varying Metabolism and Vision

The differences in the specific values attained for carrying capacity can be accounted for in the differences in the simulation set-up.

Our initial investigations have found no great differences between the AU and SU implementations. That is not to say that there will not be differences between the approaches. We would expect, based on previous work [20] that differences will be present in the simulation, the question is how big will those differences be. An example of the differences that can occur can be seen with the *Culture* rule.

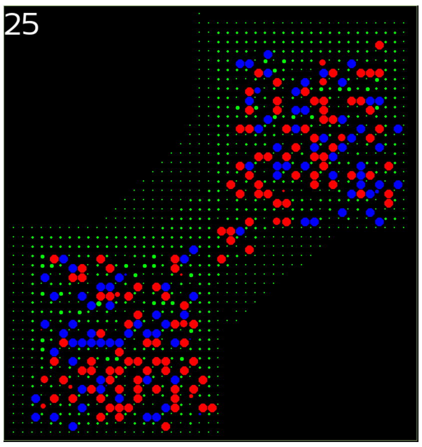

Fig. 3 Initial Culture Distribution under G_1, M, K

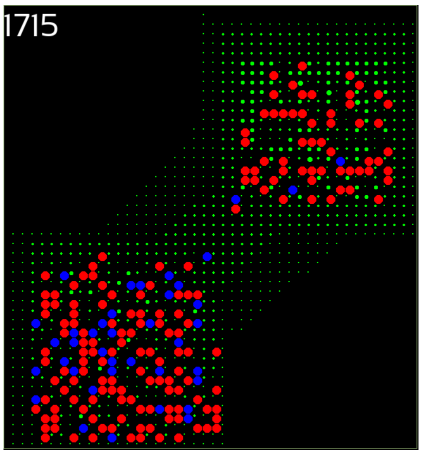

Fig. 4 Cultural Convergence begins

An agents culture is determined by an attribute containing a bit string that contains an odd number of bits. If within an agent's bit string there are more 1's than 0's then we say they are members of the red group and if they have more 0's than 1's they belong to the blue group. To simulate agents influencing each others culture we make agents that are neighbours swap a randomly selected bit with each other. In the original experiment after approximately 2,700 steps the agents converge to one or other colour across the entire lattice. The SU implementation (see figures 3 and 4), on the other hand, takes a lot longer to converge (approximately 6,000 steps). This is not surprising given what we already know about SU/AU differences but more work is required to see how big these differences are[9].

6 Related Work

It has been understood for some time that there are differences between the outcomes of AU and SU interpretations of ABM [33,8,20]. Some work has been undertaken in the area of CA to try understand these effects [33], there has been little similar work undertaken in ABM [16]. We now briefly survey other attempts made to apply SU to ABM. All of these SU 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.

6.1 Influences and Reaction

In [14] a model of situated multi agent systems, *Influences and Reaction*, is derived from *Situation Calculus* [26]. It extends Situation Calculus to enable it to deal with simultaneous interactions. This is achieved by splitting state into two components.

[9] For example, the population size can affect convergence times and we have no precise figures on the original agent count.

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.

6.2 Synchronous IBM

An Individual Based Model (IBM) [8] in the field of ecology was developed using both SU and AU so as to compare the results. The model is a simple Individual Based Model 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 AU and SU 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 were actually 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 fully synchronous.

6.3 Transactional Cellular Automata

Following on from [14], the *Influences and Reaction* model was used as a basis for a framework that converts ABM into equivalent synchronous CA [39]. The resulting CA is called a *Transactional CA*. There were two main reasons for attempting this:

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

In the Transactional CA model updates occur using a three step approach [39]:

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 ($N > 1$) 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 [23] proposed for computer games which instead proposes a multi-pass exchange of messages between objects, thus allowing it to handle complex interactions correctly.

6.4 Pedestrian Traffic

In [7] a stochastic Cellular Automata 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." [7]

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 SU is the most widely accepted approach in Cellular Automata based traffic models.

6.5 Turmites

In [13] a synchronous implementation of the multi-turmite model is presented. It is used to demonstrate that different updating schemes have large effects on simulation outcomes. The model itself is very simple with only one type of agent and only one type of behaviour [25]:

– The *vant*[10] 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.

They found out that *Deadlock* is possible between agents when using their algorithm. In contrast the algorithms we propose guarantee deadlock and livelock freedom while also allowing us to incorporate different collision resolution strategies into our rules.

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 sequence of moves employed) but a synchronous rule could use a different strategy (perhaps closest agent to the destination location ‘wins’). The implicit assumption that AU will be used leads many modellers to ignore the collision resolution issue without justifying that decision.

6.6 Synchronous Objects

The Synchronous Objects [23, 24] model of programming was developed for Computer Games and not ABM based simulations. However, there are many similarities between ABM and computer games. The model used in computer games is based on many interacting heterogeneous agents with overall system properties emerging from their interactions (similar to the ABM approach). The Synchronous Objects model was inspired by the BSP [40] and Active Objects [6, 17] approaches to concurrency and adapted for development of simulations (albeit Games Simulations). Its purpose

[10] vant stands for Virtual Ant.

was to make concurrent programming of game simulations easier by removing issues of *Deadlock*, *Livelock* and *Data races*. 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 SU along with message passing similar to that proposed in the *Transactional CA* [39] 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 approach requires a change to the message passing paradigm of programming. This is something modellers may not be familiar or comfortable with.

6.7 Combined Synchronous and Asynchronous

In [36] a model that combined agents using SU alongside agents that used AU 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. The authors state that synchronous interactions are not always easy to implement and the interactions in this simulation are 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 the interactions handled by our algorithms. It is not clear how this approach could handle these more complex interactions.

6.8 Summary

There have been a number of attempts at incorporating SU into ABMs. None of these has been completely successful. It is not clear how (or if) these approaches could handle interactions from Sugarscape such as *Combat*. The example ABMs that have been implemented using these approaches have contained only very simple interactions. The most complex interaction implemented is *Movement*, one of the simplest interactions available within Sugarscape. The space-time complexity of these approaches has not been derived and also require further analysis.

7 Conclusions

While AU and SU implementations of ABMs can produce different outcomes it is still an open question as to which is more realistic. Even if it is the case that real world systems are, in some sense, asynchronous it is not proven that AU properly implements an asynchronous world. AU proposes a sequential model of the world even though the world is concurrent. It is posited that by sequentially updating the state of each individual agent while holding all other agents' state constant we obtain the behaviour of an asynchronous system. For this argument to hold each action would have to be instantaneous in time [13], something that is seldom true in real world systems. As interactions are rarely instantaneous the onus is on anyone using AU to demonstrate that the interactions are instantaneous.

We have also shown that AU breaks the principle of locality (used to enforce bounded rationality). That is, it allows agents to be immediately aware of events not within their neighbourhood and to be aware of these events even before other agents who are within the neighbourhood of the events are aware of them. In fact, we have shown that delayed information is only properly (and consistently) produced by SU. SU also allows us to employ different collision detection and resolution solutions.

Our 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 implementations of the system.

Both of these seem to be borne out in the literature [8, 28].

We categorised agent interactions 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. This synchronous approach reduces nondeterminism, increases clarity and allows different conflict resolution rules to be applied.

We then 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 equal, using geometric tiling, in the concurrent case. Both the synchronous and asynchronous algorithms offer good *weak scalability*. Our 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. We demonstrated this implementing a single resource version of the formally defined [21, 22] Sugarscape using our SU algorithms. This is the only SU implementation of Sugarscape that we are aware of and the most complex ABM implemented with SU.

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. Our SU algorithms mean that it is now possible for these types of comparison to be made with ABMs as well.

7.1 Further Work

Initial benchmarking on multicore processors indicates that these algorithms scale within the bounds set out by the asymptotic analysis. Work on converting the algorithms for execution on Graphics Processing Units (GPU) is ongoing.

We intend to extend the framework to allow for two resources enabling us to implement the *Trade* rule. We will also refactor the code to allow AU and SU implementations of the same simulation to run side by side. This will allow precise comparisons to be made as each simulation being compared can have identical parameter settings.

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.

References

1. Bach*, L.A., Sumpter, D.J., Alsner, J., Loeschcke, V.: Spatial evolutionary games of interaction among generic cancer cells. Journal of Theoretical Medicine **5**(1), 47–58 (2003)
2. Berryman, M.: Review of software platforms for agent based models. Tech. rep., DTIC Document (2008)
3. Bezbradica, M., Ruskin, H.J., Crane, M.: Comparative analysis of asynchronous cellular automata in stochastic pharmaceutical modelling. Journal of Computational Science **5**(5), 834–840 (2014)
4. Bigbee, A., Cioffi-Revilla, C., Luke, S.: Replication of Sugarscape using MASON. Springer (2007)
5. Brailsford, S.: DES is alive and kicking. J Simulation **8**(1), 1–8 (2014). URL http://dx.doi.org/10.1057/jos.2013.13
6. Briot, J.P., Guerraoui, R., Lohr, K.P.: Concurrency and distribution in object-oriented programming. ACM Comput. Surv. **30**(3), 291–329 (1998). DOI http://doi.acm.org.remote.library.dcu.ie/10.1145/292469.292470
7. Burstedde, C., Klauck, K., Schadschneider, A., Zittartz, J.: Simulation of pedestrian dynamics using a two-dimensional cellular automaton. Physica A: Statistical Mechanics and its Applications **295**(3), 507–525 (2001). URL http://EconPapers.repec.org/RePEc:eee:phsmap:v:295:y:2001:i:3:p:507-525
8. Caron-Lormier, G., Humphry, R.W., Bohan, D.A., Hawes, C., Thorbek, P.: Asynchronous and synchronous updating in individual-based models. Ecological Modelling **212**(3 4), 522 – 527 (2008). DOI http://dx.doi.org/10.1016/j.ecolmodel.2007.10.049. URL http://www.sciencedirect.com/science/article/pii/S0304380007006011
9. Cornforth, D., Green, D.G., Newth, D.: Ordered asynchronous processes in multi-agent systems. Physica D: Nonlinear Phenomena **204**(1 2), 70 – 82 (2005). DOI http://dx.doi.org/10.1016/j.physd.2005.04.005. URL http://www.sciencedirect.com/science/article/pii/S0167278905001338
10. Edmonds, B., Hales, D.: Replication, replication and replication: Some hard lessons from model alignment. Journal of Artificial Societies and Social Simulation **6**(4) (2003)
11. Epstein, J.M.: Agent-based computational models and generative social science. Complex. **4**(5), 41–60 (1999). DOI 10.1002/(SICI)1099-0526(199905/06)4:5<41::AID-CPLX9>3.0.CO;2-F. URL http://dx.doi.org/10.1002/(SICI)1099-0526(199905/06)4:5<41::AID-CPLX9>3.0.CO;2-F
12. Epstein, J.M., Axtell, R.: Growing Artificial Societies: Social Science from the Bottom Up. The Brookings Institution, Washington, DC, USA (1996)
13. Fatès, N., Chevrier, V.: 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, pp. 533–540. International Foundation for Autonomous Agents and Multiagent Systems (2010)
14. Ferber, J., Müller, J.P.: Influences and reaction: a model of situated multiagent systems. In: Proceedings of Second International Conference on Multi-Agent Systems (ICMAS-96), pp. 72–79 (1996)
15. Gardner, M.: Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". Scientific American pp. 120–123 (1970)
16. Grilo, C., Correia, L.: Effects of asynchronism on evolutionary games. Journal of Theoretical Biology **269**(1), 109 – 122 (2011). DOI http://dx.doi.org/10.1016/j.jtbi.2010.10.022. URL http://www.sciencedirect.com/science/article/pii/S0022519310005564
17. Hernandez, J., de Miguel, P., Barrena, M., Martinez, J., Polo, A., Nieto, M.: Parallel and distributed programming with an actor-based language. In: Parallel and Distributed Processing, 1994. Proceedings. Second Euromicro Workshop on, pp. 420 –427 (1994)

18. Hinsen, K.: A data and code model for reproducible research and executable papers. Procedia Computer Science **4**(0), 579 – 588 (2011). DOI http://dx.doi.org/10.1016/j.procs.2011.04.061. URL http://www.sciencedirect.com/science/article/pii/S1877050911001190. Proceedings of the International Conference on Computational Science, {ICCS} 2011
19. Hirth, D.H., McCullough, D.R.: Evolution of alarm signals in ungulates with special reference to white-tailed deer. The American Naturalist **111**(977), pp. 31–42 (1977). URL http://www.jstor.org/stable/2459977
20. Huberman, B.A., Glance, N.S.: Evolutionary games and computer simulations. Proceedings of the National Academy of Sciences **90**(16), 7716–7718 (1993)
21. Kehoe, J.: The specification of sugarscape. CoRR **abs/1505.06012** (2015). URL http://arxiv.org/abs/1505.06012
22. Kehoe, J.: Creating reproducible agent based models using formal methods. In: 17th International Workshop on Multi-Agent-Based Simulation. Springer (2016)
23. Kehoe, J., Morris, J.: A concurrency model for game scripting. In: 12th International Conference on Intelligent Games and Simulation, pp. 10–16. EUROSIS (2011)
24. Kehoe, J., Morris, J.: Scalable parallelism in games. Sixth York Doctoral Symposium (2013)
25. Langton, C.G.: Studying artificial life with cellular automata. Phys. D **2**(1-3), 120–149 (1986). URL http://dl.acm.org/citation.cfm?id=25201.25210
26. Lespérance, Y., Levesque, H.J., Lin, F., Marcu, D., Reiter, R., Scherl, R.B.: Foundations of a logical approach to agent programming. In: Intelligent Agents II Agent Theories, Architectures, and Languages, pp. 331–346. Springer (1996)
27. Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., Balan, G.: Mason: A multiagent simulation environment. Simulation **81**(7), 517–527 (2005). DOI 10.1177/0037549705058073. URL http://dx.doi.org.remote.library.dcu.ie/10.1177/0037549705058073
28. May, R.M.R.M.: Stability and complexity in model ecosystems. Monographs in population biology. Princeton, N.J. Princeton University Press (1973). URL http://opac.inria.fr/record=b1083163
29. Newth, D., Cornforth, D.: Asynchronous spatial evolutionary games. Biosystems **95**(2), 120 – 129 (2009). DOI http://dx.doi.org/10.1016/j.biosystems.2008.09.003. URL http://www.sciencedirect.com/science/article/pii/S0303264708001949
30. North, M.J., Collier, N.T., Ozik, J., Tatara, E.R., Macal, C.M., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with repast simphony. Complex Adaptive Systems Modeling **1**(1), 1–26 (2013)
31. Nowak, M.A., May, R.M.: Evolutionary games and spatial chaos. Nature **359**(6398), 826–829 (1992). DOI 10.1038/359826a0. URL http://dx.doi.org/10.1038/359826a0
32. Onggo, B.S.: Running agent-based models on a discrete-event simulator. In: Proceedings of the 24th European Simulation and Modelling Conference, pp. 25–27. Citeseer (2010)
33. Radax, W., Rengs, B.: Timing matters: Lessons From The CA Literature On Updating. Third Word Congress of Social Simulation (2010)
34. Railsback, S., Lytinen, S., Grimm, V.: Stupidmodel and extensions: A template and teaching tool for agent-based modeling platforms (2005)
35. Ruxton, G.D., Saravia, L.A.: The need for biological realism in the updating of cellular automata models. Ecological Modelling **107**(2 3), 105 – 112 (1998). DOI http://dx.doi.org/10.1016/S0304-3800(97)00179-8. URL http://www.sciencedirect.com/science/article/pii/S0304380097001798
36. Sahay, N., Ierapetritou, M., Wassick, J.: Synchronous and asynchronous decision making strategies in supply chains. Computers and Chemical Engineering **71**, 116 – 129 (2014). DOI http://dx.doi.org/10.1016/j.compchemeng.2014.07.005. URL http://www.sciencedirect.com/science/article/pii/S0098135414002099
37. Sansores, C., Pav n, J.: Agent-based simulation replication: A model driven architecture approach. In: A.F. Gelbukh, A. de Albornoz, H. Terashima-Mar n (eds.) MICAI, *Lecture Notes in Computer Science*, vol. 3789, pp. 244–253. Springer (2005). URL http://dblp.uni-trier.de/db/conf/micai/micai2005.html#SansoresP05
38. Siebers, P.O., Macal, C.M., Garnett, J., Buxton, D., Pidd, M.: Discrete-event simulation is dead, long live agent-based simulation! J. Simulation **4**(3), 204–210 (2010)
39. Spicher, A., Fatès, N., Simonin, O.: Translating discrete multi-agents systems into cellular automata: Application to diffusion-limited aggregation. In: J. Filipe, A. Fred, B. Sharp (eds.) Agents and Artificial Intelligence, *Communications in Computer and Information Science*, vol. 67, pp. 270–282. Springer Berlin Heidelberg (2010)

40. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990). DOI http://doi.acm.org.remote.library.dcu.ie/10.1145/79173.79181
41. Zaft, G.C., Zeigler, B.P.: 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** (2002)