Towards Temporal Verification of Swarm Robotic Systems

Clare Dixon[a], Alan Winfield[b], Michael Fisher[a], Chengxiu Zeng[a]

[a]*Department of Computer Science,University of Liverpool, Liverpool, L69 3BX, UK*
[b]*Bristol Robotics Laboratory, University of the West of England, Bristol BS16 1QY, UK*

Abstract

A robot swarm is a collection of simple robots designed to work together to carry out some task. Such swarms rely on: the simplicity of the individual robots; the fault tolerance inherent in having a large population of identical robots; and the self-organised behaviour of the swarm as a whole. Although robot swarms present an attractive solution to demanding real-world applications, designing individual control algorithms that can *guarantee* the required global behaviour is a difficult problem. In this paper we assess and apply the use of *formal verification* techniques for analysing the emergent behaviours of robotic swarms. These techniques, based on the automated analysis of systems using *temporal logics*, allow us to analyse whether all possible behaviours within the robot swarm conform to some required specification. In particular, we apply *model-checking*, an automated and exhaustive algorithmic technique, to check whether temporal properties are satisfied on all the possible behaviours of the system. We target a particular swarm control algorithm that has been tested in real robotic swarms, and show how automated temporal analysis can help to refine and analyse such an algorithm.

Keywords: Swarm Robotics, Formal Verification, Emergent Behaviour, Temporal Logics, Model-Checking.

1. Introduction

The use of autonomous robots has become increasing appealing in areas which are hostile to humans such as underwater environments, contaminated areas, and space, or where direct human control is infeasible due to the complexity or speed of the robot interactions [1, 2, 3]. Rather than deploying one or two, often large and expensive, robots a significant focus is now on the design and development of *swarms* of robots.

A robot swarm is a collection of simple (and usually identical) robots working together to carry out some task [4, 5, 6]. Each robot has a relatively small set of behaviours and is typically able to interact with other (nearby) robots and with its environment. Robot swarms are particularly appealing when compared with fewer, more complex robots, in that it may be possible to design a swarm so that the failure of some of the robots will not jeopardize the overall mission, i.e. the swarm is *fault tolerant*. Such swarms are also advantageous from a financial point of view since each robot is relatively simple and mass production can significantly reduce the fabrication costs.

Despite the advantages of deploying swarms in practice, it is non-trivial for designers to formulate individual robot behaviours so that the emergent behaviour of the swarm as a whole is guaranteed to achieve the task required of the swarm, while the swarm will not exhibit any other, undesirable, behaviours [7]. Specifically, it is often difficult to predict the overall behaviour of the swarm just given the local robot control algorithms. This is, of course, essential if swarm designers are to be able

Email addresses: `cldixon@liverpool.ac.uk` (Clare Dixon), `Alan.Winfield@uwe.ac.uk` (Alan Winfield), `mfisher@liverpool.ac.uk` (Michael Fisher), `zcx@liverpool.ac.uk` (Chengxiu Zeng)

to effectively and confidently develop reliable swarms. So, we require some mechanism for analysing what the swarm can do, given the behaviour of individual robots and a description of their possible interactions both with each other and with their environment. Using such a mechanism, the designer can then assess to what extent the swarm behaves as required and, where necessary, redesign to avoid unwanted outcomes.

Currently, the analysis of swarm behaviour is typically carried out by experimenting with real robot swarms or by simulating the robot swarms and testing various scenarios (e.g. see [8, 9]). In both these cases any errors found will only be relevant to the particular scenarios constructed; neither provides a comprehensive analysis of the swarm behaviour in a wide range of possible circumstances. Specifically, neither approach can detect a problem where undesirable behaviour occurs in some untested situation.

A well-known alternative to simulation and testing is to use *formal verification*, and particularly the technique called *model-checking* [10]. Here a mathematical model of all the possible behaviours of the system is constructed (often a finite state transition system) and then all possible executions through this model are assessed against a required logical formula representing a desired property of the system. In the case of systems such as robot swarms, the mathematical model usually represents an abstraction of the real control system, while the logical formula assessed is usually a *temporal* formula representing the presence of a desirable property or the absence of an undesirable property on all paths; see, for example [11]. We emphasise that model checking is different from simulation as it can check a property holds on *all* paths through the input model. Simulation (or real robot experiments) allows the observation of what occurs on *particular* runs of the system.

In this paper we will develop the use of temporal verification for robot swarms in an effort to formally verify whether such swarms do indeed exhibit the required global behaviour. This work is an extension of our preliminary results in [12]. We show how such an approach can provide a useful tool for the swarm designer and, in particular, how the linked use of formal verification, abstraction, and simulation, provides a strong basis for swarm algorithm analysis. The structure of this paper is as follows. We consider related work in Section 2. In Section 3 we give details of the temporal logic in which logical requirements are given and provide an overview of the model-checker we use. In Section 4 we describe our use of formal verification to assess swarm algorithms and introduce one existing algorithm, namely Nembrini's *alpha algorithm* [9], that we will analyse. In Section 5 we describe the abstractions we use in more detail. In Section 6 we give verification results from using the temporal logic model-checker. In Section 7 we describe a simulation for the alpha algorithm and provide comparisons with the model-checking results. In Section 8 we discuss the results. We provide concluding remarks and directions for future work in Section 9.

2. Related Work

In this paper we focus on a particular swarm algorithm aimed at swarm aggregation, namely Nembrini's alpha algorithm. Its behaviour and properties have been analysed using simulations and real robot experiments in [9, 13]. The application of temporal logics to robots swarms has not been widely used. We here highlight previous work in which the formal verification of robot swarms has been considered, typically using model-checking or deductive techniques for temporal logics. We begin by examining our previous work in this area, where we have formally specified the alpha algorithm considered here again using temporal logics [14]. However, in that paper the focus is on specification rather than verification. In [15] this temporal specification of swarm algorithms was used to explore ways to generate implementations from a formal specification.

In [16] we considered the state transition system for a swarm of foraging robots from [17] and represented this using both propositional and first-order temporal logics. A number of properties are then verified using temporal resolution based theorem provers [18, 19] for these logics. Whilst this models the transition system for *each* robot, similar to that in Section 6 of this paper, it focuses on

the state or mode that the robots are in rather than their specific location or movement details. In the propositional setting we could only represent a small number of robots due to the state explosion problem also encountered in this paper (discussed further in Section 8.3). In the first-order case, the robots are represented using variables and, although this allows us to represent an *infinite* number of robots, syntactic restrictions of the monodic first-order temporal logic used in the prover added further representational limitations.

In [20] the probabilistic model of a swarm of foraging robots presented in [17, 21] is analysed using the probabilistic model-checker PRISM [22]. Again the actual location of the robots is ignored and the focus is on the modes of the robots, e.g. resting, searching for food, etc. Different ways of modelling the swarm are proposed, including the product of individual robot transition systems and using a *counting abstraction* to model the whole swarm using a single transition system where the number of robots in each state is recorded. Probabilistic properties relating to, for example, swarm energy are checked for different swarm parameters.

In [23, 24] a model-checking approach is adopted to considering the motion of robot swarms. A hierarchical framework is suggested to abstract away from the many details of the problem including the location of the individual robots. First, a continuous abstraction is used to capture the main features of the swarm's position and size (the example considered uses the centroid and variance of robot positions to achieve this). Next this continuous abstraction is abstracted further, providing a discrete model to which model-checking can then be applied. That approach differs in a number of ways to this paper. In particular, we are interested in the emergent behaviour of the swarm so model individual robots whereas in that paper the descriptions of individual robots are abstracted away from. Further, the paper assumes a centralised communication architecture which is not assumed here.

A related paper is [11] which again considers model-checking robot motion but does not discuss robot swarms. That paper uses a discrete representation of the continuous space of movement producing a finite state transition system. A model-checker is used to produce traces that satisfy particular properties (e.g., visiting regions in a particular order, eventually visiting a region but avoiding other regions on the way). These are then used to produce a continuous movement plan whilst maintaining the required property. The differences between that work and ours are that they do not deal with robot swarms which is our focus here. A related approach is given in [25] which, after assessing a number modelling and verification techniques, focuses on model-checking. There the underlying transition system has states that relate to the robots' behaviour, but only a small number (three) of robots is considered.

Other formalisms have been considered to specify and verify aspects of realistic robot swarms. In [26], Rouff et al. compare a number of formal methods for representing and verifying part of the Autonomous Nano Technology Swarm (ANTS) mission aimed at sending small swarms of spacecraft to study the asteroid belt. As a result of this, in [27] four formal methods were selected, namely Communicating Sequential Processes (CSP), Weighted Synchronous Calculus of Communicating Systems (WSCCS), X-Machines and Unity Logic. These are proposed for use alongside techniques from *agent-oriented software engineering*. While the authors do not apply these techniques, they conclude that there is a need to develop new formal techniques alongside specialised sets of models and software processes based on a number of formal methods and other areas software engineering. Importantly, they do do not tackle specific swarm architectures or carry out full verification experiments.

Finally, in [28] the authors propose a formalism for describing and analysing emergent properties and behaviour from lower level behaviours of a system. Again the focus of the paper is on presenting the formalism rather than applying it to a specific case study.

3. Temporal Logic and Model-Checking

In order to carry out formal verification we aim to apply *model-checking* to a particular robot swarm algorithm. Model-checking [29] is an algorithmic technique that checks whether specified

temporal properties hold over the evolution of dynamic systems as they change over time. Temporal logic is used to describe such temporal properties.

3.1. Temporal Logic

Temporal logics have been widely used to represent and reason about systems that change over time [30, 31, 32]. The particular variety of temporal logic we consider is propositional linear-time temporal logic (PTL) [33], where the underlying model of time is isomorphic to the Natural Numbers, $\mathbb{N}$.

3.1.1. Syntax

The symbols from propositional logic we include are '**false**' (*false*), $\neg$ (*not*), $\vee$ (*or*). The future-time temporal connectives that we use include '$\Diamond$' (*sometime in the future*), '$\Box$' (*always in the future*), and '$\bigcirc$' (*in the next moment in time*). Formally, PTL formulae are constructed from the following elements:

- a set, PROP of propositional symbols;
- propositional connectives, **false**, $\neg$, $\vee$; and
- temporal connectives, $\bigcirc$, $\Diamond$, $\Box$

The set of well-formed PTL formulae (WFF), is inductively defined as the smallest set satisfying the following.

- Any element of PROP and **true** and **false** are in WFF.
- If A and B are in WFF then so are $\neg A \quad A \vee B \quad \Diamond A \quad \Box A \quad \bigcirc A$

Note that the other Boolean operators can be defined using standard equivalences and other operators such as '$\mathcal{U}$' (*until*) can be incorporated. A *literal* is defined as either a proposition symbol or the negation of a proposition symbol.

3.1.2. Semantics

Models for PTL formulae can be characterised as sequences of *states* of the form:

$$\sigma = s_0, s_1, s_2, s_3, \ldots$$

where each state, s_i, is a set of proposition symbols, representing those propositions which are satisfied in the ith moment in time.

The notation $(\sigma, i) \models A$ denotes the truth of formula A in the model σ at state index $i \in \mathbb{N}$ and this is formally defined as follows.

$$\begin{array}{lll}
(\sigma, i) \not\models \textbf{false} & & \\
(\sigma, i) \models p & \text{iff} & p \in s_i \text{ where } p \in \text{PROP} \\
(\sigma, i) \models A \vee B & \text{iff} & (\sigma, i) \models A \text{ or } (\sigma, i) \models B \\
(\sigma, i) \models \neg A & \text{iff} & (\sigma, i) \not\models A \\
(\sigma, i) \models \bigcirc A & \text{iff} & (\sigma, i+1) \models A \\
(\sigma, i) \models \Diamond A & \text{iff} & \exists k \in \mathbb{N}.\ (k \geqslant i) \text{ and } (\sigma, k) \models A \\
(\sigma, i) \models \Box A & \text{iff} & \forall j \in \mathbb{N}.\ \text{if } (j \geqslant i) \text{ then } (\sigma, j) \models A
\end{array}$$

For any formula A, model σ, and state index $i \in \mathbb{N}$, then either $(\sigma, i) \models A$ holds or $(\sigma, i) \models A$ does not hold, denoted by $(\sigma, i) \not\models A$. If there is some σ such that $(\sigma, 0) \models A$, then A is said to be *satisfiable*. If $(\sigma, 0) \models A$ for all models, σ, then A is said to be *valid* and is written $\models A$.

3.2. Model-Checking

Model-checking [10] is a popular technique for verifying the temporal properties of systems. It is an algorithmic technique for exhaustively analysing the logical correctness of a finitely-represented system. It checks a logical requirement against all possible behaviours of the system in question. Input to the model-checker is a model of the system and a property to be checked. The model is a finite structure, for example a finite-state transition system representing all the paths through the system to be verified. The property is a formula to be checked on that model usually expressed in some form of temporal logic, for example, the linear-time temporal logic PTL, or the branching-time temporal logic CTL [34]. Model-checking has come to prominence in recent years as it provides fast, automated, and relatively easy to use verification techniques. A number of practical model-checking tools have been developed for example SPIN [35], `NuSMV` [36], Java PathFinder [37], and UPPAAL [38]. In this paper we will use `NuSMV` [36] which allows properties expressed in both CTL and PTL to be checked.

Essentially, we construct a set of finite-state transition systems, corresponding to each of the robots in the swarm, and then model-check a PTL formula against the concurrent composition of these transition systems. The key element of model-checkers is that, if there are execution paths of the system that *do not* satisfy the required temporal formula, then at least one such "failing" path will be returned as a counter-example. If no such counter-examples are produced then all paths through the system indeed satisfy the prescribed temporal formula.

4. Analysing Swarm Algorithms

Analysing the behaviours of large and complex robotic control systems is notoriously difficult, and is particularly problematic for formal verification techniques. However, when we move on to consider robot swarms, two aspects lead us to re-consider the use of formal verification:

1. individual robotic behaviours are *much* simpler within a swarm than within larger individual robots; and
2. the emergent behaviour of cooperating swarms is hard to simulate and test using standard techniques.

So, our aim is to develop, deploy and extend formal verification techniques for use in swarm robotics and so show that formal verification is viable in this, more restricted, context. Yet, even within swarm robotics, the application of formal verification remains very difficult. The continuous aspects of both the robotic control system and the robots' movements and environment do not sit well with the discrete and finite nature of model-checking. Fortunately, in developing simple robotic control algorithms, engineers typically use finite-state machines as part of their behavioural design. Thus, we can base our verification on such finite-state machines provided by roboticists and so verify the algorithms, as designed. There remains the problem of the use of continuous functions/variables in such state machines. As in the verification of *hybrid systems*, we must provide abstractions to simplify such continuous values so that model-checking can be carried out [39]. However, we note that even with such abstractions, representing the location and movement for a number of robots will generate a huge state space so initially we must focus on small grid sizes and numbers of robots.

Thus, in summary, our approach is as follows.

1. Take the design of a swarm control algorithm for an individual robot, as represented in the form of a finite-state machine.
2. Describe an abstraction that tackles the continuous nature of the domain, the potentially unbounded number of robots, and the nature of concurrent activity and communication within the swarm.

3. Carry out (automatic) model-checking to assess the temporal behaviour of the model from (2). If model-checking succeeds, then return to (2) refining the abstraction to make it increasingly realistic. If model-checking fails, returning a scenario in which the temporal requirement is not achieved, then analyse (by hand) how the algorithm in (1) *should* cope with this scenario. Either there is a problem with the original algorithm, so this must be revised, or the algorithm is correct for this scenario and so the abstraction in (2) must be revisited and expanded to capture this behaviour.

This process is continued until no errors are found in (3) and the abstraction in (2) is sufficiently close to the physical scenario to be convincing, and here is one area where we can employ simulation techniques. While this cycle is clearly not (and cannot be) fully automatic, the results from model-checking help direct us in refining the algorithm and/or abstractions used in the design. This approach follows the spirit of the "counter-example guided abstraction refinement" method initiated in [40] and applied to hybrid systems in [41].

4.1. The 'Alpha' Algorithm

As a case study we consider algorithms for robot swarms which make use of local wireless connectivity information alone to achieve swarm aggregation. Specifically, we examine the simplest (alpha) algorithm described in [9, 13]. Here each robot has range-limited wireless communication which, for simplicity, we model as covering a finite distance in all directions from the robot's location. Beyond this boundary, robots are out of detection range.

The basic alpha algorithm is very simple:

- The default behaviour of a robot is forward motion.
- While moving each robot periodically sends an "Are you there?" message. It will receive "Yes, I am here" messages only from those robots that are in range, namely its neighbours.
- If the number of a robot's neighbours should fall below the threshold α then it assumes it is moving *out* of the swarm and will execute a $180°$ turn.
- When the number of neighbours rises above α (when the swarm is regained) the robot then executes a random turn. This is to avoid the swarm simply collapsing in on itself.

Thus, each robot has three basic behaviours: move forward (default); avoidance (triggered by the collision sensor); and coherence (triggered by the number of connected neighbours falling below α), as shown in the Finite State Machine of Figure 1.

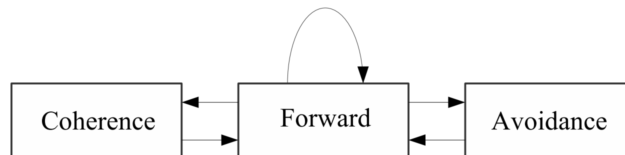

Figure 1: Alpha algorithm robot controller Finite State Machine

5. Abstraction

In the following we explain in more detail the abstractions that are used to build an appropriate model to be checked. We note, first of all, that the robot control engineers have already made significant simplifications, particularly concerning continuous dynamics. Thus, in [9, 13] a continuous

abstraction is used to capture the main features of the swarm's position and size. An example considered is using the centroid and variance of robot positions. This continuous abstraction must then be discretised, before model-checking can be applied.

Spatial Aspects. We consider a number of identical robots moving about a square grid and assume that the grid is divided into squares with at most one robot in each square. We assume a step size of one grid square and that a robot can detect other robots for purposes of avoidance in the adjacent squares. The robot has a direction it is moving in. Rather than taking a bearing, we simply describe this as one of *North*, *South*, *East* or *West*. To make the problem finite we limit the grid size to be an $n \times n$ square which wraps around, i.e. assume that the horizontal x-axis and vertical y-axis is marked from 0 to $n - 1$. In the grid we assume that moving North moves upwards, South moves downwards, East moves to the right and West to the left. If a robot is located in square $(n - 1, m)$ and moves East in the next moment it will be in square $(0, m)$. Similarly if a robot is located in square $(m, 0)$ and moves South in the next moment it will be in square $(m, n - 1)$ etc. Initially the robots may have any direction but are placed on the grid where they are connected to the swarm but in different grid squares. Initially, any robot may move first.

Connectivity. Regarding connectivity, this is calculated from the robots' relative positions. Initially we assume that each robot can detect other robots in the eight squares surrounding it. Hence, in a 5×5 grid if a robot is in square (1,1) (denoted in Figures 2 and 3 by R) it can detect robots in squares (0,0), (0,1), (0,2), (1,0), (1,2), (2,0), (2,1), and (2,2). This is shown in Figure 2. Thus it has a wireless range

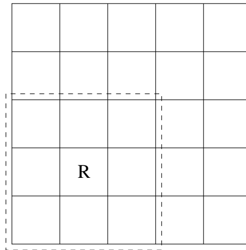

Figure 2: Wireless range of 1 in a 5×5 grid

of one square in all directions. Note that as we have assumed that the grid *wraps round* the wireless detection range acts similarly. For example if the grid size is 5×5, a robot located in grid square (2,0) will be connected with robots in the grid squares (1,4), (2,4), (3,4) as well as with robots in the five squares adjacent to it. The wireless range for such a robot is depicted in Figure 3. Initially we set the

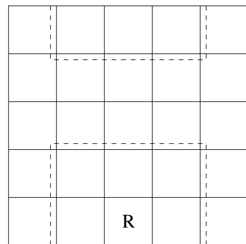

Figure 3: Wireless range of 1 in a 5×5 grid showing wrap around

value of $\alpha = 1$ i.e. a robot is connected if it can detect at least one other robot.

The proposition con_i denotes that robot i is connected to at least α other robots. In our verification, we aim to show that for all robots i, $\Box\Diamond con_i$ follows from our specification, i.e. each robot stays connected infinitely often. Thus, the swarm need not *always* be connected but, if it becomes

disconnected, should eventually reconnect. In particular, no specific robot will remain disconnected forever. Further, we note that checking $\Box\Diamond con_i$ for robot swarms of size four or larger with $\alpha = 1$ would be satisfied on runs where the swarm breaks into two sub-parts with the robots in each sub-part remaining connected infinitely often. However, as in this paper we only deal with small swarm sizes this will not be a problem.

Motion Modes. Each robot can be in one of two motion modes: *forward* or *coherence*. The connectivity of each robot can also be in one of two modes: *connected* or *not connected*. The combination of motion and connectivity give us four possible alternatives.

- In the forward mode, when connected, move forward and the motion mode remains 'forward'.
- In the forward mode, but not connected, turn $180°$ and change the motion mode to 'coherence'.
- In the coherence mode, but not connected, move forward and the motion mode remains as 'coherence'.
- In the coherence mode, when connected, perform a 90° turn (i.e. either 90° left or 90° right) and change the motion mode to 'forward'.

Avoidance is dealt with as follows. If a robot is moving in some direction and the square ahead is occupied:

- move to the right or left;
- if these are both occupied move backwards; else
- stay in the current position.

The original direction the robot was moving in is maintained.

Concurrency. An important variety of abstraction concerns the representation of concurrent activity within the swarm of robots. Thus, in modelling this aspect we must ask questions about whether all robots run simultaneously, whether some robots run faster than others, etc. Thus, there is a wide variety of different abstractions we might use, which in turn correspond to different mechanisms for concurrently composing the robot transition systems/models.

- *synchrony* — all robots execute at the same time and with the same clock.
- *(strict) turn taking* — execution of the robots is essentially interleaved, but the robots must execute in a certain order, e.g. r_1, r_2, r_3, r_1, r_2, r_3, etc.
- *(non-strict) turn taking* — execution of the robots is again interleaved but for m robots in every cycle of m steps each robot moves once, so we can now have a situation where a robot executes two steps consecutively, e.g. r_1, r_2, r_3, r_3, r_2, r_1, etc.
- *(fair) asynchrony* — robots execute at the same time, yet some robots are faster than others. However the *fair* aspect ensures that a robot can only take a finite number of steps before all other robots have finished their step, i.e. each robot must get a chance to move infinitely often.

It is important to note that the particular view of concurrency taken can significantly affect the verification results.

6. Verification Using Model-Checking

We model Nembrini's alpha algorithm and aim to verify the property $\Box\Diamond con_i$ for each of the i robots. The model is a finite-state transition system written in NuSMV's input language representing the algorithm in Section 5. Due to the large state spaces involved we assume initially that the grid is 5×5 and there are two robots and subsequently increase these values. We appreciate both the grid size and number of robots is small but this will increase and allowing the grid to wrap around means that the robots can, for example, move North forever. As well as considering different numbers of robots and grid sizes we will change the abstraction we use relating to concurrency and wireless range.

6.1. The NuSMV Input File

We provide a sample input file for the two robots synchronous case with a 5×5 grid where $\alpha = 1$ in Appendix A. The input file represents the movement of the robots as described in Section 5. Essentially the transition system representing the movement of a single robot is instantiated a number of times and the model-checker constructs the cross product of these on which to check the property. In the input file the keyword 'MODULE' is used to specify the movement of *each* robot. Variables storing the robot's location, direction and motion mode are defined. The next value of variables is defined as a series of cases dependent on the current value of some conditions. For example the statement below gives the next value for the robot direction. It states that if the current mode of the robot is forward, it is not connected and its direction is North then in the next moment its direction will be South.

```
fnocon & direction = n : s;
```

Each robot is instantiated a number of times depending on the number of the robots required. Fair asynchrony is modelled using the NuSMV keyword 'process' indicating that one of the instantiated modules is chosen to execute non-deterministically. The 'FAIRNESS running' statement is used to ensure each instantiated robot will be chosen to execute infinitely often. To model strict or non-strict turn taking the movement of the robots is modelled as synchronous but the rules defining movement are guarded by whether it is the current robot's turn to move. If it is not that robot's turn to move, the location, direction and mode of the robot remains the same. Several global variables are defined such as 'range' to denote the wireless range and 'lenmax' to denote the grid size.

The property we wish to verify is denoted as follows

```
LTLSPEC G F connected1  & G F connected2;
```

where 'LTLSPEC' states this is a property from propositional linear time temporal logic, ' G' represents the temporal operator always ($\Box$), 'F' represents the temporal operator sometime ($\Diamond$) and '&' is the propositional logic operator conjunction ($\wedge$). The variables connected1 and connected2 represent the propositions con_i where $i = 1, 2$ and are calculated from the robots' relative positions. Note that a response of '*true*' from the model-checker means that every path from every initial state of the model satisfies this property. A response of '*false*' means not all paths satisfy the property and NuSMV outputs a failing trace.

The following results are from running NuSMV on a PC with a 2.13 GHz Intel Core 2 Duo E6400 processor, 3GB main memory, and 5GB virtual memory running Fedora release 9 with a 32-bit Linux kernel.

6.2. Results: Changing the Concurrency Mode

Here we provide results from running the model-checker where the concurrency mode is fair asynchrony, non-strict turn taking, strict turn taking and then synchrony. In each case we consider two robots with grid sizes 5×5, 6×6, 7×7 and 8×8 and three robots with grid sizes 5×5 and 6×6.

Fair asynchrony means that each robot can take an arbitrary number of steps before another has a turn to move but each robot must move infinitely often, i.e. we disallow situations where, after some point, one robot never gets a chance to move again. Intuitively we might expect this to be the best choice to model robot swarms since, because of physical differences between robots they will not move at exactly the same rate. With (non-strict) turn taking, exactly one robot can move at any moment and for m robots in every cycle of m steps each robot moves once. Thus for two robots where t_i denotes robot i takes a turn to move $t_1, t_2, t_1, t_2, t_2, t_1, \ldots$ represents a valid sequence of turns but $t_1, t_2, t_1, t_1, \ldots$ does not. With strict turn taking, exactly one robot can move at any moment and they follow a strict order. For example, for two robots we might have $t_1, t_2, t_1, t_2, t_1, t_2, \ldots$. With synchrony robots all move at the same time.

For fair asynchrony, non-strict turn taking and strict turn taking, for the size of grid and number of robots we have experimented with, the model-checker returns '*false*'. This means that `NuSMV` can show that the property *does not* hold on all paths from every initial state for fair asynchrony, non-strict turn taking and strict turn taking for every case that was tried. The two robot cases take less than two minutes whereas the three robot cases take more than an hour. Looking at the failing traces obtained help us to identify the cause of the failure. These are output automatically by the model-checker and may not represent the shortest failing trace. In the following failing traces r_1 denotes that robot 1, and r_2 denotes robot 2. We use the following abbreviations: *FC* (forward and connected); *FNC* (forward and not connected); *CC* (coherence and connected); *CNC* (coherence and not connected); *Loc* (location); and *Dir* (Direction). Additionally (x, y) denotes the x and y positions of the robot. If this is underlined it denotes that this robot moves next.

In Figure 4 we provide a failing trace for the fair asynchronous case for a 6×6 grid for two robots produced by `NuSMV`. From state 1.2 onwards the pair of robots is unconnected. State 1.19 is the same as 1.7 showing that the pair of robots can loop round the states 1.8 to 1.19 forever remaining unconnected. Also in this cycle each robot gets at least one chance to move, satisfying our fairness constraint. The trace shows that, after some initial moves, both robot 1 moves West forever and

State	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15	1.16	1.17	1.18	1.19
Loc r_1	(0,0)	$\underline{(0,0)}$	$\underline{(5,0)}$	(4,0)	$\underline{(4,0)}$	$\underline{(3,0)}$	$\underline{(2,0)}$	(2,0)	(2,0)	$\underline{(2,0)}$	$\underline{(1,0)}$	$\underline{(0,0)}$	$\underline{(5,0)}$	$\underline{(4,0)}$	$\underline{(3,0)}$	(2,0)	(2,0)	(2,0)	$\underline{(2,0)}$
Dir r_1	E	E	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
Mode r_1	FC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC
Loc r_2	$\underline{(0,1)}$	(0,2)	(0,2)	$\underline{(0,2)}$	(0,1)	(0,1)	(0,1)	$\underline{(0,0)}$	$\underline{(0,5)}$	(0,4)	(0,4)	(0,4)	(0,4)	(0,4)	(0,4)	$\underline{(0,4)}$	$\underline{(0,3)}$	$\underline{(0,2)}$	(0,1)
Dir r_2	N	N	N	N	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
Mode r_2	FC	FNC	FNC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC

Figure 4: Failing trace for fair asynchrony with a 6×6 grid and two robots.

robot 2 moves South forever. Allowing each robot to take several steps before the other has a turn to move means that they never become connected again. Although the movement of the real world robots is inherently asynchronous, as each robot will run at slightly different rates, the modelling of this using `NuSMV`'s fair asynchrony does not appear to capture this sensibly. This is because using fair asynchrony with the model-checker one robot can take hundreds of steps before any other gets a chance to move. This doesn't match the real world expectation that all robots are moving at the same time and at almost the same speed. Because of this we next consider non-strict and then strict turn taking.

In Figure 5 we provide a failing trace for the non-strict turn taking case for the 6×6 grid output for two robots produced by `NuSMV`. We use the same abbreviations as previously. From state 1.2 onwards the pair of robots is unconnected. State 1.17 is the same as 1.5 showing that the pair of robots can loop round the states 1.6 to 1.17 forever, remaining unconnected. Looking at the failing trace in Figure 5, robot 1 makes a move resulting in the robots losing connectedness which they never regain. This may be due to a number of reasons. One reason is that robot 1 has to wait two steps before it can take any action relating to this loss of connectedness (robot 2 next moves twice). This might be avoided by using a strict turn taking abstraction which we consider next.

State	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15	1.16	1.17
Loc r_1	(0,0)	(0,5)	(0,5)	(0,5)	(0,0)	(0,0)	(0,1)	(0,1)	(0,2)	(0,3)	(0,3)	(0,3)	(0,4)	(0,4)	(0,5)	(0,5)	(0,0)
Dir r_1	S	S	S	S	N	N	N	N	N	N	N	N	N	N	N	N	N
Mode r_1	FC	FNC	FNC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC
Loc r_2	(0,1)	(0,1)	(5,1)	(4,1)	(4,1)	(3,1)	(3,1)	(2,1)	(2,1)	(2,1)	(1,1)	(0,1)	(0,1)	(5,1)	(5,1)	(4,1)	(4,1)
Dir r_2	E	E	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W
Mode r_2	FC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC

Figure 5: Failing trace for non-strict turn-taking with a 6×6 grid and two robots.

In Figure 6 we provide a failing trace for the 5×5 grid output for two robots by NuSMV where the abbreviations used are as previously. From state 1.3 onwards the pair of robots is unconnected. State 1.15 is the same as 1.5 showing that the pair of robots can loop round the states 1.6 to 1.15 forever remaining unconnected. In this trace robot r_1 does some avoidance moving from state 1.1 to 1.2 as robot r_2 is in its way. Considering the failing trace, the robots lose connectedness and both end up

State	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15
Loc r_1	(0,0)	(4,0)	(4,0)	(4,4)	(4,4)	(4,3)	(4,3)	(4,2)	(4,2)	(4,1)	(4,1)	(4,0)	(4,0)	(4,4)	(4,4)
Dir r_1	N	N	N	S	S	S	S	S	S	S	S	S	S	S	S
Mode r_1	FC	FC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC
Loc r_2	(0,1)	(0,1)	(0,2)	(0,2)	(0,1)	(0,1)	(0,0)	(0,0)	(0,4)	(0,4)	(0,3)	(0,3)	(0,2)	(0,2)	(0,1)
Dir r_2	N	N	N	N	S	S	S	S	S	S	S	S	S	S	S
Mode r_2	FC	FC	FNC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC

Figure 6: Failing trace for strict turn taking with a 5×5 grid with two robots.

travelling South forever trying to re-connect to the swarm. Note that in the first step, robot 1 tries to move Northwards to location (0,1) but cannot do this as robot 2 is there. Robot 1 carries out some avoidance and moves sideways to location (4,0). Note that, as both robots are still connected, this would be an alternative starting location which would not involve avoidance.

These results show that changing the computational abstraction (from fair asynchrony to non strict or strict turn taking) does not appear to be the best abstraction to model concurrency as a robot may have to wait one or more steps before reacting to a loss in connectedness. Next we consider a different method of concurrency, i.e. synchrony where robots all move at the same time. This is different from the previous cases where robots moved one at a time. Now model-checking returns '*true*' for all the two robot cases tried and '*false*' for all the three robot cases tried. The two robots cases take less than one second whereas whilst for the three robot cases the 5×5 case takes less than one hour the 6×6 case requires more than an hour.

Observing the successful traces of the two robot cases, if the robots are both moving in the same direction originally they continue in this same direction in the forward connected mode forever. If they are in different directions they move away from each other, correct this in the coherence mode by moving back together, become re-connected and repeat one of the above two patterns again. This type of movement can be observed from the experiments with the real robots.

Although real robot swarm are strictly speaking asynchronous, the real robot implementation of the alpha algorithm means that they behave collectively as if they were synchronous (as explained in more detail in Section 8.1). Thus we feel that runs of the model checker in synchronous mode is a better abstraction to model the alpha algorithm. This explains why the traces obtained match more closely the runs of the real world whereas the runs for the other modes of concurrency we considered do not.

With three robots and grid sizes of 5×5 and larger, it is possible for one robot to become disconnected from the other two. In Figure 7 we show a failing trace for three robots on a 5×5 grid. In Figure 7 we can see robots 1 and 2 remain connected with each other right from the start both travelling North. However robot 3 is moving South and becomes disconnected after one step. Although this robot turns and moves back to its starting position (1,0) it can never catch up with the other two robots and remains disconnected forever.

It could be argued that the reasons for this are either that, in the three robot case, we need a bigger

State	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8
Loc r_1	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,0)	(0,1)	(0,2)
Dir r_1	N	N	N	N	N	N	N	N
Mode r_1	FC	FC	FC	FC	FC	FC	FC	FC
Loc r_2	(0,1)	(0,2)	(0,3)	(0,4)	(0,0)	(0,1)	(0,2)	(0,3)
Dir r_2	N	N	N	N	N	N	N	N
Mode r_2	FC	FC	FC	FC	FC	FC	FC	FC
Loc r_3	(1,0)	(1,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,0)
Dir r_3	S	S	N	N	N	N	N	N
Mode r_3	FC	FNC	CNC	CNC	CNC	CNC	CNC	CNC

Figure 7: Failing trace for synchrony with a 5×5 grid with three robots.

α parameter, or that the wireless range is too small potentially resulting in frequent loss of connection (or both). Hence we next attempt to consider these cases specifically by increasing the wireless range so that it is larger than the step size and increasing the α parameter, i.e. in the three robot case each robot needs to remain connected to two others.

6.3. Results: Increasing Wireless Range or Alpha Parameter

We now increase the number of squares over which we can detect other robots. This was previously set at one square from a robot. We now increase this to be two squares from a robot, i.e. a robot can detect others in the five by five squares centred on the robot. In the following we assume synchronous concurrency. Obviously, as we have increased the detection area, in grid sizes of 5×5 and smaller all robots will always be connected at all moments. Hence we only consider grid sizes of 6×6, 7×7 and 8×8, 9×9, and 10×10 and three robots with grids sizes 6×6, 7×7 and 8×8. With the increased wireless range we still obtain '*true*' for all the two robot cases analysed and '*false*' for all the three robot cases tried similar to synchronous case with the wireless range being one. Times for the two robot cases are less than two seconds whereas the times taken for the three robots cases are greater than an hour.

In Figure 8 we show a failing trace for three robots on a 6×6 grid. Examining the failing trace with the increased wireless range, see Figure 8, we reach a cycle where all the robots move in the same direction, two within wireless range in the forward connected mode and one not within wireless range in the coherence non-connected mode. The disconnected robot never manages to catch the others.

State	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10
Loc r_1	(0,0)	(5,0)	(0,0)	(0,5)	(0,4)	(0,3)	(0,2)	(0,1)	(0,0)	(0,5)
Dir r_1	W	W	E	S	S	S	S	S	S	S
Mode r_1	FC	FNC	CC	FC	FC	FC	FC	FC	FC	FC
Loc r_2	(2,2)	(2,1)	(2,0)	(2,5)	(2,4)	(2,3)	(2,2)	(2,1)	(2,0)	(2,5)
Dir r_2	S	S	S	S	S	S	S	S	S	S
Mode r_2	FC	FC	FC	FC	FC	FC	FC	FC	FC	FC
Loc r_3	(2,1)	(2,2)	(2,3)	(2,2)	(2,1)	(2,0)	(2,5)	(2,4)	(2,3)	(2,2)
Dir r_3	N	N	N	S	S	S	S	S	S	S
Mode r_3	FC	FC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC

Figure 8: Failing trace for synchrony with a 6×6 grid with three robots and increased wireless range.

We now increase the α parameter from one to two, meaning that each robot must be able to detect two others (hence we need to have at least three robots). We note that now, with this modified alpha algorithm (i.e. with $\alpha = 2$) we still obtain results of '*false*' for all the three robot cases tried (grid sizes 5×5 to 8×8) which all finish in less than an hour. Considering the failing traces for three robots we reach a cycle where all the robots are moving in the same direction in the coherence non-connected mode, i.e. there are not two other robots in range. This is illustrated in the failing trace in Figure 9

State	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15	1.16
Loc r_1	(0,0)	(4,0)	(0,0)	(0,1)	(0,0)	(4,0)	(0,0)	(0,4)	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,0)	(0,1)	(0,2)
Dir r_1	W	W	E	N	S	W	E	S	N	N	N	N	N	N	N	N
Mode r_1	FC	FNC	CC	FNC	CC	FNC	CC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC
Loc r_2	(1,1)	(0,1)	(1,1)	(1,0)	(1,4)	(1,3)	(1,4)	(2,4)	(1,4)	(1,3)	(1,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
Dir r_2	W	W	E	S	S	S	N	E	W	S	N	N	N	N	N	N
Mode r_2	FC	FNC	CC	FC	FC	FNC	CC	FNC	CC	FNC	CNC	CNC	CNC	CNC	CNC	CNC
Loc r_3	(1,0)	(2,0)	(1,0)	(1,4)	(1,0)	(0,0)	(1,0)	(1,4)	(1,3)	(1,4)	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,0)
Dir r_3	E	E	W	S	N	W	E	S	S	N	N	N	N	N	N	N
Mode r_3	FC	FNC	CC	FNC	CC	FNC	CC	FC	FNC	CNC	CNC	CNC	CNC	CNC	CNC	CNC

Figure 9: Failing trace for synchrony with a 5×5 grid and three robots with $\alpha = 2$.

where after some initial moves all the robots move North in the coherence mode but are not connected to both the other two robots. We note that we can also find a failing trace for both $\alpha = 2$ and wireless range $= 2$ for an 8×8 grid. The failure of the three robot (synchronous) cases, illustrated by the traces in Figures 7, 8 and 9, seem to be due to a known flaw in the *alpha algorithm* described in [9]. This will be discussed further in Section 8.

We can continue this until either the abstractions are realistic enough, or until our verification attempts take too much time/space. We will discuss the latter in Section 8. However, as the failing traces appear to demonstrate a real (and known) flaw of the *alpha algorithm* we do not continue attempting verification for other cases.

7. Simulation

We have also developed a NetLogo [42] simulation of the alpha algorithm to investigate and compare the failing traces in the synchronous cases obtained from `NuSMV` with runs of the simulation. NetLogo is a multi-agent modelling environment where multiple of agents can be modelled and interactions between individual agent's behaviour and any emergent patterns can be observed. NetLogo has two principal agents, a mobile agent *turtle* and a stationary agent *patch*. In the graphical user interface, a turtle is represented as a chevron and a patch is represented as a tile. We use a turtle to model individual robots. A turtle is suitable to model robot behaviours as it has attributes such as heading, position and can move one step forward, detect the existence of other agents within a range etc. The space the turtle moves in is two dimensional and is divided up into a grid of patches, similar to the grid square abstraction.

The simulation code is developed based on the pseudocode for the alpha algorithm from [9]. There are two main classes: the GUI class responsible for the simulation's Graphical User Interface, and Robot class. The GUI class has attributes for configuring alpha parameter and robot's population, direction and procedures for initialising robot and space and triggering interactions between robots. The space is wrap-around but not restricted to be a limited grid size. If two robots are connected then this is depicted by a line drawn between them. The Robot class is used to model each robot in a swarm. It inherits from the Turtle class and several attributes are defined corresponding to the variables from the pseudocode for the alpha algorithm. An example of the simulation displaying three robots is provided in Figure 10.

We use the simulation to try and compare the synchronous failing traces shown in Section 6 with runs in the simulation. First we run the simulation with two robots and $\alpha = 1$. Similar to the successful traces described in Section 6.2 after several initial moves, the two robots stay connected and move towards the direction forever. A screen shot from this is shown in Figure 11, with the two robots moving East and the line between them depicting that they are connected.

We next consider the simulation with three robots case, also with $\alpha = 1$ to try and compare with the failing trace in Figure 7. In the failing trace we observe a run where after some moves all three robots move in the same direction but one robot never catches up with other two connected robots. This behaviour can also be observed in runs of the simulation and is shown in Figure 12 where all three robots move North, the top two are connected whilst the bottom robot is not connected.

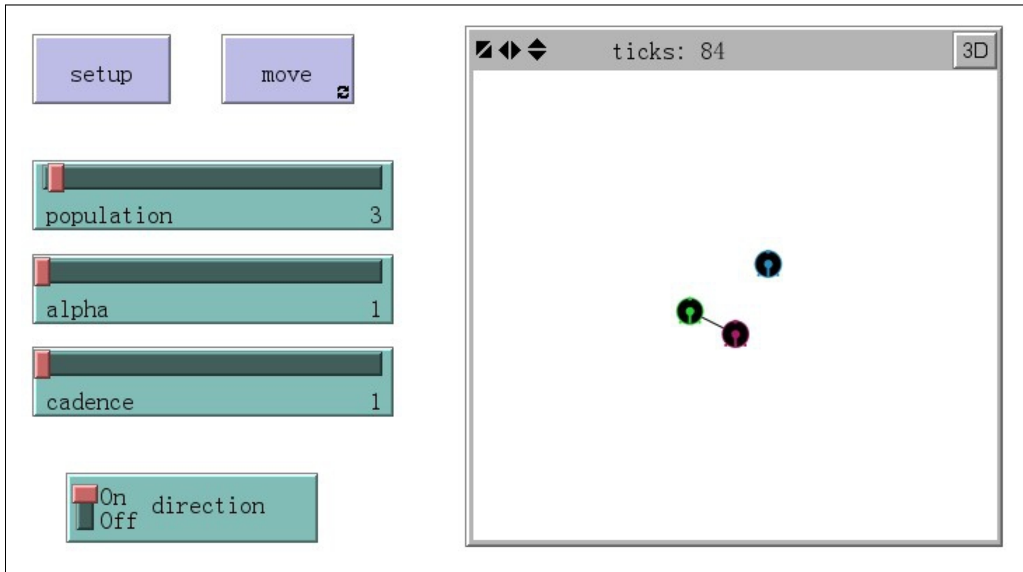

Figure 10: Interface to the simulation showing three robots with two connected

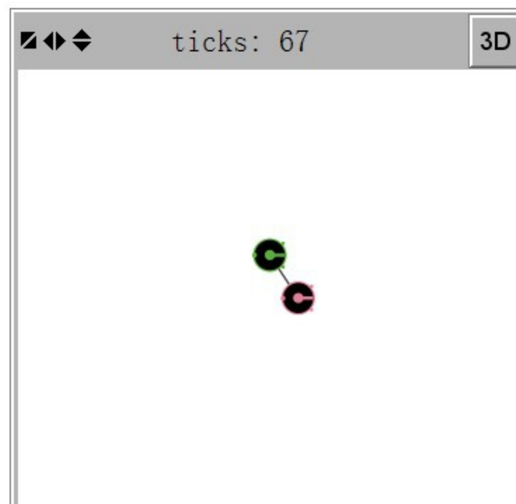

Figure 11: Two robots with $\alpha = 1$

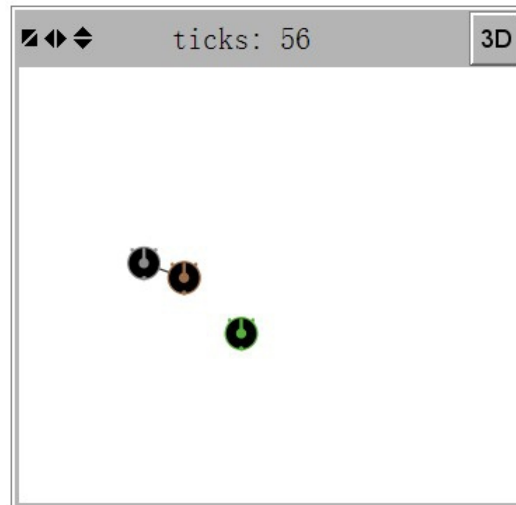

Figure 12: Three robots with $\alpha = 1$

If we increase the alpha parameter from one to two in the simulation we can obtain the type of failing trace as shown in Figure 9. Figure 13 shows a run in the simulation where all three robots move towards the same direction (Eastwards) but none of them is connected.

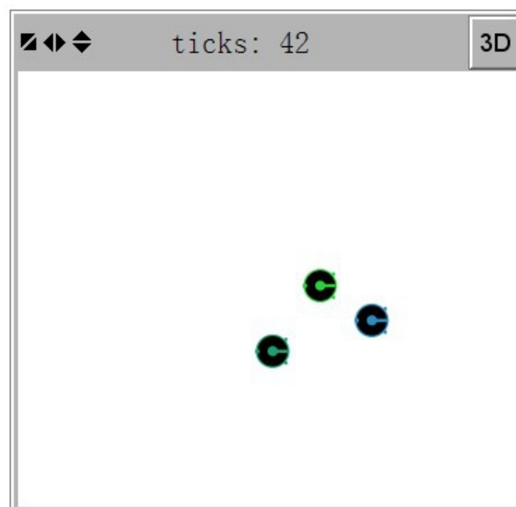

Figure 13: Three robots with $\alpha = 2$

Given the failing traces produced by `NuSMV`, we have used the relevant parameters from these traces eg alpha parameter, number of robots, starting positions etc as input parameters to the simulation and observed the resulting runs. In each case we have been able to observe a run of the same form as the failing trace output by the model-checker during verification. This helps us to confirm that there is indeed a problem with the algorithm, at least in the cases of small numbers of robots.

Note that we can represent larger swarms of robots with the NetLogo simulation. Experiments with larger swarms are described in [43], following the experiments in [9], which vary the number of robots and alpha parameter. Even for larger population sizes not all runs maintain coherence. This is explained in [9] and discussed in the next section.

8. Discussion

Next we discuss the abstractions we have taken, compare the model-checking results with existing results from the alpha algorithm and consider the size of the state space.

8.1. Abstractions

First we consider the method of concurrency used to model the problem. Intuitively, the choice of fair asynchrony seems to be the obvious choice because, with real robots moving in the world, there is no global clock that the robots have access to and we cannot guarantee that each robot will move at exactly the same speed since their physical components, e.g. motors, wheels, etc., may have small differences. Hence we considered this mode of concurrency first. However our model-checking results led us to believe that fair asynchrony, non-strict and strict turn taking are all unsuitable for modelling this problem. In the first case one robot may make many moves and in the second case one robot may be able to make two moves before the others have chance to react. In the case of strict turn taking a robot has to wait for the others all to make moves before it can react to a loss in connectedness. This is confirmed by the '*false*' results obtained in Section 6. Such failing traces would not be represented by the actual runs of the algorithm. Hence we believe that synchrony, counter to our original expectations, is the best abstraction for concurrency. This is because (i) the real robots move at almost the same speed and importantly (ii) they check their connectivity status multiple times per movement step, so two real robots that become disconnected both react to the loss of connectivity at very nearly the same time, in effect, synchronously.

Regarding the *wrap around* grid abstraction we note that, potentially, some failing traces could not appear in actual runs of the algorithm. If we took smaller grids, for example 4×4, and a disconnected robot travelling in some direction, for example West, then the wrap around nature of the grid may mean the robot becomes re-connected to robots that are some distance East of the robot. For this reason we have only considered grid sizes of 5×5 or larger and have checked that the failing traces listed above do not suffer from this problem. This choice is supported by the notion of *grid independence* defined below.

Let the term *grid independent* mean that the robot movement in a failing trace for a wrap around grid can be translated into an infinite grid obtaining the same connectedness values for each robot. We can show that, in the case of synchrony, once we have found a *grid independent trace* for an $n \times n$ grid we can extend this to a *grid independent trace* for an $m \times m$ grid where $m > n$. Hence when we find one such trace for a number of synchronous robots and some grid size we do not have to try any larger sized grids. Additionally we assume that $2\alpha + 1 < n$ otherwise all the robots on an $n \times n$ finite grid will be connected. The theorems and proofs can be found in Appendix B.

Note that grid independent traces will not include traces where robots are moving in opposite directions in rows/columns that are less than or equal to α from each other. For example one robot travelling North and another travelling South with x values of 3 and 4 respectively. With a finite wrap-round grid such robots may meet each other having moved off the top and bottom of the grid respectively and become re-connected so the robots will take some action as they perceive that they have re-located the swarm, whereas in the infinite grid the robots would remain disconnected forever moving away from each other.

Finally we consider the relative sizes of the robot movement step size, avoidance range and wireless range. In all the examples we have considered, apart from where the wireless range is set to 2 in all directions (Section 6.3), we have set wireless range equal movement step size equal avoidance step. In real robots and sensor based simulations, the movement step size is usually less than avoidance range which is usually less than wireless range. This would be something to explore when applying verification to other swarm algorithms.

8.2. *The Alpha Algorithm*

The alpha algorithm has been well studied and, to date, analysed with both simulated and real robots [9], and using a probabilistic mathematical modelling approach [13]. However, neither simulation, real robot experiments, nor mathematical modelling provide formal proof of the algorithm's correctness. Consider simulation (or real robot experiments, which may be regarded as 'embodied' simulations). Given that simulated (or real robots) move in a real-valued, not grid-based, world with typical swarm sizes of 40 robots and alpha values of 5, 10 or 15, for 10,000 seconds [13], we have a practically infinite state-space and each simulation run tests only a tiny number of paths through that state-space.

As outlined in Section 4, the model-checking approach of this paper attempts to formally prove correctness by reducing the state-space to a tractable size so that *every* path through that state-space can be tested. That reduction is achieved by means of introducing, firstly, a number of simplifying assumptions and, secondly, by running very small swarm sizes (2 or 3 robots) and very small grid-worlds (5×5 up to 8×8). We thus need to ask ourselves whether the failure to verify the correctness of the algorithm in some cases of Section 5 is because (a) the algorithm is flawed or (b) the simplifying assumptions (necessary for tractability) are so severe that they go beyond the bounds within which the algorithm should be expected to work.

The model-checking results of Section 6 show that, with the three robot synchronous cases, we can still obtain failing traces. Some of these are of the form presented in Figure 14. This illustrates a possible scenario in which robot A has become disconnected from the swarm and turns round but never catches up with B and C. Because robots B and C remain connected to each other, with $\alpha = 1$, B and C do not react to the loss of robot A and A remains disconnected. We consider both increasing the

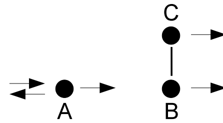

Figure 14: The bridge or cut-vertex problem

α parameter and wireless range but still obtain failing traces similar to the above form. These results confirm a known problem with the alpha algorithm, which occurs when a robot or group of robots is linked to the rest of the swarm by a single link (known as a bridge or cut-vertex). This problem was discussed by Nembrini [9] and overcome by the improved beta algorithm [9].

Finally, we discuss the size of the state space that needs to be explored.

8.3. *The State Explosion Problem*

Obviously we would like to consider larger numbers of robots. Even with the previous observation about grid independent traces, we may need to consider larger grid sizes but are faced with the well known state explosion problem [29]. Even with the simplifications we use here, the state space explored is huge. Modelling a robot's position on an $n \times n$ grid, with 4 directions, and two motion modes for r robots requires of the order of $(n \times n \times 4 \times 2)^r$ states to be explored. The state space increases further with variables to deal with turn taking etc. Let $R_n = n \times n \times 4 \times 2$. For the non-strict turn taking case (the column TT in the table below) and two robots the total state size will be $(R_n)^2 \times 4$ and for three robots $(R_n)^3 \times 21$ (where 4 and 21 represent global variables that deal with the non-strict turn taking). For strict turn taking case (the column STT in the table below) and two robots the total state size will be $(R_n)^2 \times 2$ and for three robots $(R_n)^3 \times 3$ (where 2 and 3 represent global variables that deal with the strict turn taking). For fair asynchrony and synchrony

(the column FA/SYN in the table below) and two robots the total state size will be $(R_n)^2$ and for three robots $(R_n)^3$. The change in wireless range or alpha parameter will not change the size of the state space. For example, in the case of three robots, using non-strict turn taking within a grid of 7×7,

Problem	TT No. of states	STT No. of states	FA/SYN No. of states
5×5 grid, 2	160,000	80,000	40,000
6×6 grid, 2	331,776	165,888	82,944
7×7 grid, 2	614,656	307,328	153,664
8×8 grid, 2	1,048,576	524,288	262,144
5×5 grid, 3	168,000,000	24,000,000	8,000,000
6×6 grid, 3	501,645,312	71,663,616	23,887,872
7×7 grid, 3	1,264,962,048	180,708,864	60,236,288
8×8 grid, 3	2,818,572,288	402,653,184	134,217,712

Figure 15: The state space for different modes of concurrency for differing sized grids and numbers of robots.

we have more than a thousand million states. Most of the two robot cases finish in times less than two minutes. However the three robot cases take much longer with some cases taking several hours. It is well known that model-checking suffers from the state explosion problem [29]. To combat this we will have to apply and develop some of the work that has been carried out in the model-checking field using more sophisticated abstractions, clever representations, and reduction techniques such as slicing, and symmetry in order to limit the state space [44, 45, 46].

9. Conclusions and Future Work

In this paper we have shown how formal verification can be used as part of the development of reliable robot swarm algorithms. In particular, we have applied verification using model-checking to a particular swarm algorithm, namely the *alpha algorithm*. To apply model-checking we have had to develop a number of abstractions to make the model of the swarm discrete, finite and not too large. Contrary to our initial intuition, we believe that synchrony seems to be the best way to model concurrency. This is because although robot swarms are inherently asynchronous and each robot will move at a different rate real world robot swarms do not allow one robot to make a large number of steps before the other can move, one of the implications of modelling the swarm movement using fair asynchrony with the model-checker. Additionally, although we have used a finite *wrap around* grid we have shown for particular types of trace, *grid independent traces*, once we have found a failing trace for a particular grid size and set of input parameters we need not check larger grid sizes as we can extend the failing trace into one for a larger grid. The failing traces we present for the synchronous three robot examples show the bridge or cut-vertex problems with the alpha algorithm identified in [9]. We have also demonstrated these failing traces using a NetLogo simulation.

Although our examples involving two or three robots might be considered too small to constitute a swarm, they do represent a valid test-case for self-organised flocking or aggregation algorithms, such as the case study of this paper. Indeed swarms of two or three robots are useful in that they test the lower limit bounds of swarm size and are therefore much more demanding of the algorithm than larger swarm sizes with higher values of alpha, as tested in simulation studies. Indeed the abstractions required by the approach also represent a tough set of boundary cases for the algorithm. Further the production of failing traces provides a focus for deeper analysis and consideration by the swarm designer.

We clearly need further work to tackle the state explosion problem. This will involve development of abstractions and reduction techniques [45, 46] relevant to swarm algorithms as well as the application of techniques such as symmetry detection [44] to allow the analysis of larger grids, swarm

sizes and further investigation of the relative sizes of the robot step size, avoidance range and wireless range. Further, we could experiment with other model-checkers to see whether these allow larger number of robots and grid sizes than we have used in this paper. Additionally we could use probabilities to represent uncertainty such as the unreliability of the robot sensor near to the maximum range leading to the use of probabilistic model-checkers such as PRISM [22].

We would like to extend and apply the ideas developed in this paper to other swarm algorithms. Several more complex variants and extensions of the alpha-algorithm have been developed such as Nembrini's Beta algorithm [9] and the Omega-algorithm developed by Bjerknes [47]. The Omega algorithm does not make use of wireless connectivity to maintain aggregation but instead times the duration since a robot last made an avoidance manoeuvre; if that value exceeds threshold, omega, then the robot turns toward its estimate of the centre of the swarm based on sensor readings.

A simple extension to the Omega-algorithm referred to above allows for a more complex swarm behaviour, namely emergent swarm taxis toward a beacon (i.e. an Infra-Red light source). This is a particularly interesting future candidate for analysis using formal verification and model-checking since the swarm-taxis behaviour is truly emergent; it requires a minimum of, typically, five robots.

References

[1] J. Yuh, Design and Control of Autonomous Underwater Robots: A Survey, Autonomous Robots 8 (1) (2000) 7–24.

[2] T. Arai, E. Pagello, L. Parker, Editorial: Advances in Multi-Robot Systems, IEEE Trans. Robotics and Automation 18 (5) (2002) 655–661.

[3] W. Truszkowski, H. Hallock, C. Rouff, J. Karlin, J. Rash, M. Hinchey, R. Sterritt, Swarms in Space Missions, in: Autonomous and Autonomic Systems: With Applications to NASA Intelligent Spacecraft Operations and Exploration Systems, NASA Monographs in Systems and Software Eng. Springer, 2009, pp. 207–221.

[4] E. Bonabeau, M. Dorigo, G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems, Journal of Artificial Societies and Social Simulation 4 (1), (2001).

[5] G. Beni, From Swarm Intelligence to Swarm Robotics, in: Proc. International Workshop on Swarm Robotics (SAB), Revised Selected Papers, Vol. 3342 of Lecture Notes in Computer Science, Springer, 2005, pp. 1–9.

[6] E. Sahin, A. F. T. Winfield, Special issue on Swarm Robotics, Swarm Intelligence 2 (2-4) (2008) 69–72.

[7] W. M. Spears, D. F. Spears, J. C. Hamann, R. Heil, Distributed, Physics-Based Control of Swarms of Vehicles, Autonomous Robots 17 (2-3) (2004) 137–162.

[8] K. Støy, Using situated communication in distributed autonomous mobile robotics, in: SCAI '01: Proceedings of the Seventh Scandinavian Conference on Artificial Intelligence, IOS Press, 2001, pp. 44–52.

[9] J. Nembrini, Minimalist Coherent Swarming of Wireless Networked Autonomous Mobile Robots, Ph.D. thesis, University of the West of England (2005).

[10] E. Clarke, O. Grumberg, D. A. Peled, Model Checking, MIT Press, 2000.

[11] G. Fainekos, H. Kress-Gazit, G. Pappas, Temporal Logic Motion Planning for Mobile Robots, in: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), IEEE Computer Society Press, 2005, pp. 2020– 2025.

[12] C.Dixon, A. Winfield, M. Fisher, Towards Temporal Verification of Emergent Behaviours in Swarm Robotic Systems , in: Towards Autonomous Robotic Systems (TAROS), Vol. 6856 of LNCS, Springer, 2011, pp. 336–347.

[13] A. Winfield, W. Liu, J. Nembrini, A. Martinoli, Modelling a wireless connected swarm of mobile robots, Swarm Intelligence 2 (2-4) (2008) 241–266.

[14] A. Winfield, J. Sa, M.-C. Fernández-Gago, C. Dixon, M. Fisher, On Formal Specification of Emergent Behaviours in Swarm Robotic Systems, International Journal of Advanced Robotic Systems 2 (4) (2005) 363–370.

[15] D. Chen, *A Simulation Environment for Swarm Robotic System based on Temporal Logic Specifications*, Master's thesis, University of the West of England (November 2005).

[16] A. Behdenna, C. Dixon, M. Fisher, Deductive Verification of Simple Foraging Robotic Behaviours, International Journal of Intelligent Computing and Cybernetics 2 (4) (2009) 604–643.

[17] W. Liu, A. Winfield, J. Sa, J. Chen, L. Dou, Strategies for Energy Optimisation in a Swarm of Foraging Robots, in: Proc. 2nd International Workshop on Swarm Robotics (SAB), Vol. 4433 of LNCS, Springer, 2007, pp. 14–26.

[18] U. Hustadt, B. Konev, TRP++ 2.0: A temporal resolution prover, in: Automated Deduction—CADE-19, Vol. 2741 of Lecture Notes in Artificial Intelligence, Springer, 2003, pp. 274–278.

[19] M. Ludwig, U. Hustadt, Fair derivations in monodic temporal reasoning, in: Conference on Automated Deduction–CADE, Vol. 5663 of Lecture Notes in Computer Science, Springer, 2009, pp. 261–276.

[20] S. Konur, C. Dixon, M. Fisher, Analysing Robot Swarm Behaviour via Probabilistic Model Checking, Robotics and Autonomous Systems 60 (2) (2012) 199–213.

[21] W. Liu, A. F. T. Winfield, Modeling and optimization of adaptive foraging in swarm robotic systems, The International Journal of Robotics Research 29 (14) (2010) 1743–1760.

[22] A. Hinton, M. Kwiatkowska, G. Norman, D. Parker, PRISM: A Tool for Automatic Verification of Probabilistic Systems, in: Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Vol. 3920 of LNCS, Springer, 2006, pp. 441–444.

[23] M. Kloetzer, C. Belta, Hierarchical abstractions for robotic swarms, in: Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on, 2006, pp. 952 –957.

[24] M. Kloetzer, C. Belta, Temporal Logic Planning and Control of Robotic Swarms by Hierarchical Abstractions, IEEE Transactions On Robotics 23 (2007) 320–330.

[25] S. Jeyaraman, A. Tsourdos, R. Zbikowski, B. White, Kripke Modelling Approaches of a Multiple Robots System with Minimalist Communication: A Formal Approach of Choice, International Journal of Systems Science 37 (6) (2006) 339–349.

[26] C. Rouff, A. Vanderbilt, W. Truszkowski, J. Rash, M. Hinchey, Verification of NASA Emergent Systems, in: Proc. 9th IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age, IEEE Computer Society Press, 2004, pp. 231–238.

[27] C. Rouff, M. Hinchey, J. Pena, A. Ruiz-Cortes, Using Formal Methods and Agent-Oriented Software Engineering for Modeling NASA Swarm-Based Systems, in: Proc. International Swarm Intelligence Symposium (SIS), IEEE Computer Society Press, 2007, pp. 348–355.

[28] C.-C. Chen, S. Nagl, C. Clack, A calculus for multi-level emergent behaviours in component-based systems and simulations, in: Proceedings of Emergent Properties in Artificial and Natural Systems (EPNACS), 2007, pp. 35–51.

[29] E. M. Clarke, O. Grumberg, Avoiding The State Explosion Problem in Temporal Logic Model Checking, in: ACM Symposium on Principles of Distributed Computing (PODC), 1987, pp. 294–303.

[30] H. Barringer, M. Fisher, D. Gabbay, G. Gough (Eds.), Advances in Temporal Logic, Vol. 16 of Applied Logic Series, Kluwer, 2000.

[31] M. Fisher, D. Gabbay, L. Vila (Eds.), Handbook of Temporal Reasoning in Artificial Intelligence, Vol. 1 of Foundations of Artificial Intelligence Series, Elsevier, 2005.

[32] M. Fisher, An Introduction to Practical Formal Methods Using Temporal Logic, Wiley, 2011.

[33] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, The Temporal Analysis of Fairness, in: Proceedings of the Seventh ACM Symposium on the Principles of Programming Languages, Las Vegas, Nevada, 1980, pp. 163–173.

[34] E. M. Clarke, E. A. Emerson, Design and Synthesis of Synchronisation Skeletons Using Branching Time Temporal Logic, in: D. Kozen (Ed.), Proceedings of the Workshop on the Logic of Programs, Vol. 131 of LNCS, Springer, 1981, pp. 52–71.

[35] G. J. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.

[36] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: An OpenSource Tool for Symbolic Model Checking, in: Proceedings of International Conference on Computer-Aided Verification (CAV), 2002, pp. 359–364.

[37] W. Visser, K. Havelund, G. Brat, S. Park, Model Checking Programs, in: Proc. 15th IEEE International Conference on Automated Software Engineering (ASE), IEEE Computer Society Press, 2000, pp. 3–12.

[38] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, W. Yi, UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems, in: Proc. Workshop on Verification and Control of Hybrid Systems III, no. 1066 in LNCS, Springer, 1995, pp. 232–243.

[39] T. Henzinger, P.-H. Ho, H. Wong-Toi, HYTECH: A Model Checker for Hybrid Systems, International Journal on Software Tools for Technology Transfer 1 (1-2) (1997) 110–122.

[40] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-Guided Abstraction Refinement for Symbolic Model Checking, J. ACM 50 (5) (2003) 752–794.

[41] R. Alur, T. Dang, F. Ivančić, Counterexample-Guided Predicate Abstraction of Hybrid Systems, Theoretical Computer Science 354 (2) (2006) 250–271.

[42] S. Tisue, U. Wilensky, Netlogo: Design and implementation of a multi-agent modeling environment, in: Proc. Agent 2004, 2004, pp. 7–9.

[43] C. Zeng, Simulation and Verification of Robot Swarms, Master's thesis, University of Liverpool (2011).

[44] A. Miller, A. Donaldson, M. Calder, Symmetry in temporal logic model checking, ACM Computing Surveys 38 (3), (2006).

[45] O. Grumberg, H. Veith (Eds.), 25 Years of Model Checking - History, Achievements, Perspectives, Vol. 5000 of LNCS, Springer, 2008.

[46] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, 2008.

[47] J. D. Bjerknes, Scaling and Fault Tolerance in Self-organised Swarms of Mobile Robots, Ph.D. thesis, University of the West of England (2010).

Appendix A. Sample Input File

We provide the NuSMV input file for the two robot synchronous case where $\alpha = 1$.

```
---This file has two robots and a 5x5 grid.
---Robots are synchronous and have avoidance.

MODULE main
VAR
robot1: robot(connected1,lenmax,0);
robot2: robot(connected2,lenmax,{10,11});

INVAR !(robot1.loc = robot2.loc);

DEFINE
range := 1;
lenmax := 5;
abs12x:=
     case
     (robot2.xcoord >= robot1.xcoord): (robot2.xcoord - robot1.xcoord);
      1: (robot1.xcoord - robot2.xcoord);
   esac;

abs12y:=
     case
     (robot2.ycoord >= robot1.ycoord): (robot2.ycoord - robot1.ycoord);
      1: (robot1.ycoord - robot2.ycoord);
   esac;

connected1 := (((abs12x) <= range) | ((lenmax - abs12x) <= range))  &
              (((abs12y) <= range) | ((lenmax - abs12y) <= range));
connected2 := (((abs12x) <= range) | ((lenmax - abs12x) <= range))  &
              (((abs12y) <= range) | ((lenmax - abs12y) <= range));

LTLSPEC G F connected1  & G F connected2;

-- definition of each robot
MODULE robot(connected,lenmax,initloc)
VAR
loc: {00,01,02,03,04,10,11,12,13,14,20,21,22,23,24,30,31,32,33,34,40,41,42,43,44};
direction: {n,s,e,w};
motion: {for,coh};
```

```
ASSIGN
init(loc) := initloc;
next(loc):=
    case
      fcon & (direction = n)  : mn;
      fcon & (direction = n) : {me, mw};

      fcon & (direction = s)   : ms;
      fcon & (direction = s)  : {me, mw};

      fcon & (direction = e)  : me;
      fcon & (direction = e) : {mn,ms};

      fcon & (direction = w) : mw;
      fcon & (direction = w) : {mn,ms};

      cohnocon & (direction = n): mn;
      cohnocon & (direction = s): ms;
      cohnocon & (direction = e): me;
      cohnocon & (direction = w): mw;

      fnocon & (direction = n): ms;
      fnocon & (direction = s): mn;
      fnocon & (direction = e): mw;
      fnocon & (direction = w): me;

      cohcon & (direction = n) & next(direction = e): me;
      cohcon & (direction = n) & next(direction = w): mw;
      cohcon & (direction = s) & next(direction = e): me;
      cohcon & (direction = s) & next(direction = w): mw;
      cohcon & (direction = e) & next(direction = n): mn;
      cohcon & (direction = e) & next(direction = s): ms;
      cohcon & (direction = w) & next(direction = n): mn;
      cohcon & (direction = w) & next(direction = s): ms;
    1: loc;
    esac;



init(motion) := for;
next(motion):=
    case
      fcon: for;
      fnocon: coh;
      cohnocon: coh;
      cohcon: for;
    1: motion;
    esac;

init(direction) := {n,s,e,w};


next(direction):=
    case
      fcon : direction;
      fnocon & direction = n : s;
```

```
      fnocon & direction = e : w;
      fnocon & direction = s : n;
      fnocon & direction = w : e;
      cohnocon: direction;
      cohcon & direction = n: {w,e};
      cohcon & direction = s: {w,e};
      cohcon & direction = e: {n,s};
      cohcon & direction = w: {n,s};
    1: direction;
    esac;

-- variables defined from the above parameters.
DEFINE
fcon := (motion = for) & (connected = 1);
fnocon := (motion = for) & (connected = 0);
cohcon := (motion = coh) & (connected = 1);
cohnocon := (motion = coh) & (connected = 0);

mn:= 10*(loc/10) + (((loc mod 10)+1) mod lenmax);
ms:= 10*(loc/10) + (((loc mod 10) + lenmax -1) mod lenmax);
me:= (loc +10) mod (lenmax*10);
mw:= (loc +(lenmax*10) -10) mod (lenmax*10);

xcoord:= loc/10;
ycoord:= loc mod 10;
```

Appendix B. Proofs of Theorems

We provide the relevant Theorems, Lemmas and their Proofs for the notion of *grid independence* discussed in Section 8.

Lemma 1. *Let r_1 and r_2 be robots in an $n \times n$ finite grid $G_{n \times n}$ moving synchronously with a grid independent trace. Robots r_1 and r_1 are connected in $G_{n \times n}$ if and only if they are connected in the infinite grid G.*

PROOF. From the definition of grid independence.

Lemma 2. *Robot r_1 has position (x, y) in an $n \times n$ (finite) wrap round grid if and only if its position following the same sequence of movements in an infinite grid is $(jn + x, kn + y)$ for some integers j and k.*

PROOF. We show this by induction on the movement of the robot. The base case is the robot's initial position. Let us assume initially the robot is at (x_0, y_0) in the finite grid. Thus in the infinite grid it is also at (x_0, y_0) which can be represented as $(jn + x_0, kn + y_0)$ where $j = k = 0$. Assume that the Lemma holds for m moves of the robot. We show it holds for the $m + 1$th step. At step m assume the robot is at position (x', y') in the finite grid and by the inductive hypothesis at $(jn+x', kn+y')$ in the infinite grid. Because x' and y' are positions in the $n\times n$ finite grid $0 \leq x' \leq n-1$ and $0 \leq y' \leq n-1$. We assume the movement is North. We consider two cases. Firstly assume $0 \leq y' \leq n - 2$. The new position of the robot in the finite grid is $(x', y' + 1)$ and hence the new position of the robot in the infinite grid is $(jn + x', kn + y' + 1)$ (where $1 \leq (y' + 1) \leq n - 1$) as required. Next assume that $y' = n - 1$ in the finite grid. The new position of the robot in the finite grid after moving North is $(x', 0)$. In the infinite grid it is in position $(jn + x', kn + y' + 1) = (jn + x', kn + n - 1 + 1) = (jn+x', kn+n) = (jn+x', (k+1)n+0)$. In both cases we obtain the required form. The movement South, East and West is analogous.

Theorem 3. *Given a grid independent trace for an $n \times n$ grid we can extend this to a grid independent trace for an $m \times m$ grid where $m > n$.*

PROOF. First note that by assumption $2\alpha + 1 < n$ so $\alpha < n$. We show that any pair of connected robots in an $n \times n$ grid is also connected in an $m \times m$ grid (where $m > n$). Let r_1 and r_2 be robots in an $n \times n$ finite grid $G_{n \times n}$ moving synchronously with a grid independent trace. Let r_1 be at position (x, y) and robot r_2 be at position (x', y'). From Lemma 2 and the assumption that this is a grid independent trace, in the infinite grid robot r_1 is in position $(jn + x, kn + y)$ and r_2 is in position $(j'n + x', k'n + y')$ for some integers j, j', k, k'. Let m be a larger grid size, i.e. where $m > n$. Representing the robots position in the infinite grid with respect to m rather than n we have robot r_1 is at position $(j_1 m + x_1, k_1 m + y_1)$ and r_2 is in position $(j'_1 m + x'_1, k'_1 n + y'_1)$ where

$$\begin{array}{rcl} (jn + x) & = & (j_1 m + x_1) \\ (kn + y) & = & (k_1 m + y_1) \\ (j'n + x') & = & (j'_1 m + x'_1) \\ (k'n + y') & = & (k'_1 m + y'_1) \end{array}$$

Thus by Lemma 2 robot r_1 is at (x_1, y_1) in the $m \times m$ finite grid and robot r_2 at (x'_1, y'_1) in the $m \times m$ finite grid. If robots r_1 and r_2 are connected in the $n \times n$ finite grid then by Lemma 2 they are also connected in the infinite grid so

$$|(jn + x) - (j'n + x')| \leq \alpha \text{ and } |(kn + y) - (k'n + y')| \leq \alpha$$

and therefore

$$|(j_1 m + x_1) - (j'_1 m + x'_1)| \leq \alpha \text{ and } |(k_1 m + y_1) - (k'_1 m + y'_1)| \leq \alpha.$$

To show that robots r_1 and r_2 are connected in the $m \times m$ finite grid we need to show that

$$\begin{array}{c} |x_1 - x'_1| \leq \alpha \text{ or } (m - |x_1 - x'_1|) \leq \alpha \text{ and} \\ |y_1 - y'_1| \leq \alpha \text{ or } (m - |y_1 - y'_1|) \leq \alpha. \end{array}$$

The second disjunct in each pair deals with when the connectedness is measured over the edges of the grid eg with (2,5) and (2,0) where $m = 6$.

We consider the case for the x co-ordinates. The case for the y co-ordinates is similar. There are three cases to consider $j_1 = j'_1$, $j_1 = j'_1 + 1$, and $j_1 = j'_1 - 1$. Note that if $j_1 > j'_1 + 1$ or $j_1 < j'_1 - 1$ the distance between the robots in the infinite grid is $> m$. This contradicts the assumption that the robots are connected, $m > n$ and $\alpha < n$.

- $j_1 = j'_1$ As $|(j_1 m + x_1) - (j'_1 m + x'_1)| \leq \alpha$ we have $|x_1 - x'_1| \leq \alpha$ as required.
- $j_1 = j'_1 + 1$ As $|(j_1 m + x_1) - (j'_1 m + x'_1)| \leq \alpha$ we have $|((j'_1 + 1)m + x_1) - (j'_1 m + x'_1)| \leq \alpha$ and therefore $|(m + x_1 - x'_1)| \leq \alpha$. As $m > n$ and $n > \alpha$ then $x'_1 > x_1$ so $m - |x_1 - x'_1| \leq \alpha$ as required.
- $j_1 = j'_1 - 1$ This is similar to the previous case.

This shows that the difference in x co-ordinates of the robots in the finite $m \times m$ grid are less than or equal to α (or their wrap round distance from each other is less than or equal to α). The proof for the y co-ordinates is similar. So any pair of connected robots in an $n \times n$ grid is also connected in an $m \times m$ grid (where $m > n$). Thus for grid independent traces we have shown that given a failing trace from an $n \times n$ grid we can construct one for a $m \times m$ where $m > n$.