Using conventional reasoning techniques for self-organising systems

Graeme Smith
School of Information Technology and Electrical Engineering
The University of Queensland
Australia
Email: smith@itee.uq.edu.au

J. W. Sanders
African Institute for Mathematical Sciences
and
Department of Mathematical Sciences
Stellenbosch University
South Africa
Email: jsanders@aims.ac.za

***Abstract*—Self-organising systems have become important relatively recently. It is frequently claimed that their complex nature necessitates new formalisms to express and reason about them. In this paper the opposite view is taken. Following Back's use of action systems to express a distributed system as an initialised possibly nonterminating loop, here two simple but representative case studies of self-organising systems are explored using only conventional techniques. The first deals with the configuration of an *ad hoc* network and shows how safety and liveness can be accurately expressed with an initialised loop. The second involves, like many self-organising systems, probabilistic behaviour and it is shown that existing techniques suffice to establish the system behaviour. In conclusion, the techniques illustrated can be used to provide a higher level of assurance than is possible with simulation alone.**

***Index Terms*—self-organising systems, formal reasoning, guarded command language**

I. Introduction

A self-organising system, as a special kind of distributed system, is composed of components (or agents) having limited, and typically no, access to global information. It thus achieves its global goals by operating with just local information gained via inter-component interactions. Consequently, its global behaviour often appears *emergent* in the sense that it cannot be directly determined by the behaviour of the components when considered in isolation. Examples include *ad hoc* sensor networks [4], swarm robotic systems [14], [6] and decentralised trust-management systems [7], [15].

While the design of such systems is itself challenging, so too is the assurance of their effectiveness. For example, to what extent does a decentralised trust-management system ensure a low probability of untrustworthy behaviour? Does it ensure that untrustworthy behaviour is detected within a reasonably short period of time? Is it independent of particular network topologies and configurations?

The state of the art for answering such questions is to use *simulation*, *i.e.*, to produce an executable model of the system at the level of agents and their local interactions, and to observe the model's behaviour over a large number of (systematically chosen) 'runs'. While demonstrably useful, especially for uncovering major errors in system design, this approach is nevertheless inherently limited.

It is well known that for highly decentralised systems with large numbers of interacting agents, only a fraction of the possible system behaviour can be explored using simulation [14]. Hence, only weak guarantees of behaviour can generally be given. For example, in the domain of wireless sensor networks, decentralised protocols for assigning time slots to sensors have been shown to be extremely sensitive to changes in network topology [4]. Correct behaviour of one topology does not imply that of a similar topology.

This has led to interest in mathematical models that allow *formal proofs* of system behaviour [4], [14], [6], [13], [12]. In an engineering process, such approaches can be used to complement simulations: simulations can be used initially to uncover problems with the model, and proofs can then establish properties for *all* of a corrected model's possible configurations and behaviours. The more mature work in this area is focused on particular applications, specifically wireless sensor networks [4] and modular robotics [14], [6], and has not yet led to general techniques.

The action system formalism of Back [1] allows a designer to represent a distributed system as an initialised possibly non-terminating loop. The loop acts on a set of variables: some of which represent the local state of particular components in the system, and others the global shared state. Each iteration of the loop corresponds to either (i) a single component updating its local variables and/or the global shared variables, or (ii) a collection of components interacting.

This representation of a distributed system as a loop facilitates reasoning about its safety and liveness properties. Safety properties (those that hold on each iteration) can be proved as loop invariants. Liveness properties (those that hold eventually) can be proved using a loop variant.

In this paper, we apply that approach to self-organising systems. We use Dijkstra's guarded command language [3] and an extension, the probabilistic guarded-command language [9], containing probabilistic choice and specification statements [10]. After an overview of the notation in Section II, we model and reason about a simple, but typical, example of self organisation in Section III: the cluster-formation process in a cluster-based routing protocol designed for use in *ad hoc* networks [5]. In Section IV, we model and reason about a decentralised trust-management system where probabilities are used to model the reliability of a node [8]. We conclude in Section V.

II. Notation

Dijkstra's *guarded-command language* [3] is a simple, high-level programming language whose constructs include assignment, sequential composition, conditional choice and loops.

Assignment of an expression E to a variable x is written $x := E$. For $1 \leq i \leq n$, the multiple assignment of expressions E_i to variables x_i is written $x_1, \ldots, x_n := E_1, \ldots, E_n$.

Sequential composition of statements S and T is written $S \fatsemi T$. The null assignment is the identity for sequential composition and is written **skip**.

Conditional choice and loops are written using *guarded commands*. A guarded command $g \to P$ comprises a guard g, a predicate over program variables, and a program (or command) P.

Conditional choice,

$$\textbf{if} \; [] \; i : [0, n) \bullet g_i \to P_i \; \textbf{fi} \,,$$

is in practice usually written in vertical form, with a line for each guarded command $g_i \to P_i$. If at least one guard is true, one of the true guards is selected nondeterministically and its program executed. If none of the guards is true than the choice *aborts*, *i.e.*, can do anything. That allows the designer to concentrate on the case covered by the disjunction of the guards, outside which any behaviour is valid and so the design is unconstrained. In code, that is avoided by ensuring that the disjunction of the guards is true.

A loop is written similarly

$$\textbf{do} \; [] \; i : [0, n) \bullet g_i \to P_i \; \textbf{od} \,.$$

If at least one guard is true, one of the true guards is selected nondeterministically, its program executed and the process repeated. If none of the guards is true than the program terminates. A loop terminates, therefore, only when all of its guards are eventually false.

Following Morgan [10], we allow variables to be introduced to a program P using the notation $\textbf{var}\ x : T \bullet P$, and we allow *specification statements* to be used in place of programs. A *specification statement* is of the form

$$\mathbf{x} : [pre, post]$$

where the vector $\mathbf{x}$ of variables (referred to as the *frame*) contains all variables which may be changed; *pre* is the precondition (for which termination is required); and *post*, the postcondition (describing how the variables in the frame change whenever the precondition is satisfied). To distinguish initial and final values of variables in a postcondition we add a 0-subscript to the former, *e.g.*, x_0 is the initial value of variable x. However x in a precondition refers to the initial value of x. When the precondition is not satisfied, the specification statement aborts.

Finally, to model probabilities, we follow Morgan *et al.* [9] and extend the guarded-command language by a probabilistic choice operator. The program

$$P \;{}_{p}{\oplus}\; Q$$

executes program P with probability p and program Q with probability $(1 - p)$. The resulting language is called the *probabilistic guarded-command language*.

III. Cluster formation in *ad hoc* networks

As a typical example of a self organisation, we consider the cluster-formation process in a cluster-based routing protocol designed for use in *ad hoc* networks [5].

Cluster-based routing protocols are used to establish routes between network nodes. The idea is to generate a topology based around groups of nodes called clusters. Each cluster has a head node which is *adjacent*, *i.e.*, directly connected, to every other node in the cluster. Only the head nodes deal with routing requests, significantly reducing the network traffic compared with other approaches (such as flooding).

Cluster-based routing works by having every non-head node adjacent to a head node. Also, for efficiency, no two head nodes should be adjacent. Those conditions need to be established during cluster formation.

The approach we specify uses the "first declaration wins" strategy [5]: essentially, the first node in a neighbourhood to declare itself the head becomes the head. Then some means of dealing with contention, *i.e.*, when two nodes simultaneously declare themselves head, is required. But that is solved by standard techniques such as the node with the minimum (or maximum) identifier backing off, or both nodes backing off for random times. Here we abstract from such details.

We model the "first declaration wins" strategy by an initialised loop. Its safety and liveness properties can then be verified using conventional techniques involving loop invariants and variants. Let *Node* be the set of all nodes in the network and $conn : Node \leftrightarrow Node$ be an irreflexive, symmetric relation relating neighbouring nodes. Let $State = \{undecided, member, head\}$ be the type of the state of each node. The program representing the cluster-formation process has a single variable: a function $state : Node \to State$ mapping each node to its state. The use of functions such as *state* allow us to represent the large, or arbitrary, number of components typical of self-organising systems.

Initially, the state of each node is *undecided*. The body of the program is a loop which allows, on each iteration, any undecided node to become a head, and all its neighbours to become members.

```
var  $state : Node \to State$ ;
$state : [true, \forall n : Node \bullet state(n) = undecided]$ ⨾
do [] $n : Node \bullet state(n) = undecided \to$
        $state(n) := head$ ⨾
        $state : [true, \forall m : Node \setminus \{n\} \bullet$
                    $(n,m) \in conn \Rightarrow state(m) = member \wedge$
                    $(n,m) \notin conn \Rightarrow state(m) = state_0(m)]$
od
```

The loop in this case terminates when there are no nodes in *undecided* state, *i.e.*,

$$\nexists n : Node \bullet state(n) = undecided\,. \qquad (1)$$

To verify that the "first declaration wins" strategy works, we first prove partial correctness, *i.e.*, that at termination every non-head node is adjacent to a head node. The condition required at termination is expressed formally as

$$\begin{aligned}&\forall n : Node \bullet\\&\quad state(n) \neq head \Rightarrow\\&\qquad (\exists m : Node \bullet\\&\qquad\quad (n,m) \in conn \wedge state(m) = head)\,. \qquad (2)\end{aligned}$$

This is not a loop invariant since, for example, it is not true initially. However, we can show that it holds when the loop terminates by first proving an invariant of the loop. This invariant states that a member node is always connected to a head, and only member nodes are connected to heads.

Theorem 1

$$\begin{aligned}&\forall n : Node \bullet\\&\quad state(n) = member \Leftrightarrow\\&\qquad (\exists m : Node \bullet\\&\qquad\quad (n,m) \in conn \wedge state(m) = head) \qquad (3)\end{aligned}$$

Proof (by induction)
Initially (before the loop executes)

$$\begin{aligned}&\forall n : Node \bullet state(n) = undecided\\&\Rightarrow (\nexists n : Node \bullet state(n) = member) \wedge\\&\quad (\nexists n : Node \bullet state(n) = head)\\&\Rightarrow (3).\end{aligned}$$

Assume that (3) holds. When the loop is executed there is a node n such that

$$\begin{aligned}&state_0(n) = undecided \wedge && 1\\&state(n) = head \wedge && 2\\&\forall m : Node \setminus \{n\} \bullet\\&\quad (n,m) \in conn \Rightarrow state(m) = member \wedge && 3\\&\quad (n,m) \notin conn \Rightarrow state(m) = state_0(m). && 4\end{aligned}$$

From 1 and the induction hypothesis, we deduce

$$\begin{aligned}&\forall m : Node \setminus \{n\} \bullet\\&\quad (n,m) \in conn \Rightarrow state_0(m) \neq head && 5\end{aligned}$$

and from 4 and 5 and the induction hypothesis, we deduce

$$\begin{aligned}&\forall m : Node \setminus \{n\} \bullet (n,m) \notin conn \Rightarrow\\&\quad state(m) = member \Leftrightarrow\\&\qquad (\exists l : Node \bullet (m,l)\\&\qquad\quad \in conn \wedge state(l) = head). && 6\end{aligned}$$

From 2 and 3, we deduce

$$\begin{aligned}&\forall m : Node \bullet (n,m) \in conn \vee m = n \Rightarrow\\&\quad state(m) = member \Leftrightarrow\\&\qquad (\exists l : Node \bullet (m,l)\\&\qquad\quad \in conn \wedge state(l) = head) && 7\end{aligned}$$

and, hence, from 6 and 7 we deduce (3). □

Condition (2) (and partial correctness) is then immediate from (1) and (3).

To prove total correctness, we also need to prove termination. To do this, we choose $\#\{n : Node \mid state(n) = undecided\}$ as a variant. Initially, that equals $\#Node$. On each iteration of the loop, it decreases by at least one (since the selected node n changes state from *undecided* to *head*). Hence, the variant eventually reaches 0 implying the termination condition (1).

To show additionally that the strategy is efficient, we need to show that no two heads are connected. This can be proved by showing the following to be invariant.

$$\begin{aligned}&\nexists n, m : Node \bullet\\&\quad state(n) = head \wedge state(m) = head \wedge\\&\quad (n,m) \in conn \qquad (4)\end{aligned}$$

The proof by induction is similar to that above and is omitted.

The above shows that the "first declaration wins" strategy works from an initial state where all nodes are initially in *undecided* state. In a mobile *ad hoc* network, however, connections between nodes may be formed or broken as nodes move. This may result in two head nodes being connected, or a member node not being connected to a head node. In this situation, it is necessary for cluster formation to proceed from any initial configuration of node states. Gerla *et al.* [5] show how this can be achieved via head nodes sending periodic "hello" messages. If a member node does not receive such a message within a given time, it assumes its head has moved and resets itself to *undecided* state. If a head receives a "hello" message it changes to *member* state.

To verify that this approach is correct we respecify the strategy, removing the initialisation of *state* and adding additional guarded commands to the loop representing a head node sending a "hello" message, and a member node returning to *undecided* state. The latter is done only when the member node is not connected to a head node. This abstractly models the timeout. Similarly, we allow undecided nodes to become heads only when they are not connected to a head node. Again this abstractly models the undecided node waiting for a timeout before deciding to become a head.

To ensure that timeouts do occur, we introduce a notion of a round and allow each node to act once each round. This technique provides a general way of dealing with fairness in self-organising systems where the global behaviour typically relies on *all* components in the system making progress.

A variable *round* : $\mathbb{N}$ keeps track of the current round, and a variable *ready* : $\mathbb{P}\,Node$ keeps track of the nodes which are able to act but have not acted in the current round. It is initially set to the set of all nodes which are not member nodes connected to heads (these member nodes are the only nodes which do not need to act in a round). An additional guarded command resets *ready* whenever it is empty.

```
var $state : Node \to State$;
    $round : \mathbb{N}$; $ready : \mathbb{P}\,Node$ $\bullet$
$round := 0$ ⨟
$ready := \{n : Node \mid state(n) = member \Rightarrow$
            $\nexists m : Node \bullet$
                $(n,m) \in conn \wedge state(m) = head\}$ ⨟
do
[] $n : ready \bullet state(n) = undecided \wedge$
   $(\nexists m : Node \bullet (n,m) \in conn \wedge state(m) = head) \to$
      $state(n) := head$ ⨟
      $state : [true, \forall m : Node \setminus \{n\} \bullet$
               $(n,m) \in conn \Rightarrow state(m) = member \wedge$
               $(n,m) \notin conn \Rightarrow state(m) = state_0(m)]$ ⨟
      $ready :=$
            $ready \setminus (\{n\} \cup \{m : Node \mid (n,m) \in conn\})$
[] $n : ready \bullet state(n) = head \to$
      $state : [true, \forall m : Node \setminus \{n\} \bullet$
               $(n,m) \in conn \Rightarrow state(m) = member \wedge$
               $(n,m) \notin conn \Rightarrow state(m) = state_0(m)]$ ⨟
      $ready :=$
            $ready \setminus (\{n\} \cup \{m : Node \mid (n,m) \in conn\})$
[] $n : ready \bullet state(n) = member \wedge$
   $(\nexists m : Node \bullet (n,m) \in conn \wedge state(m) = head) \to$
      $state(n) := undecided$
      $ready := ready \setminus \{n\}$
[] $ready = \varnothing \to$
      $round := round + 1$ ⨟
      $ready := \{n : Node \mid state(n) = member \Rightarrow$
                  $\nexists m : Node \bullet$
                      $(n,m) \in conn \wedge state(m) = head\}$
od
```

In this case, it can be shown that the loop is non-terminating (by proving the invariant: $\nexists n : ready \bullet state(n) = member \wedge \exists\, m : Node \bullet (n,m) \in conn \wedge state(m) = head$). The property we want to prove is that after a finite number of iterations, (2) and (4) are true. To do this, we prove the following six invariants which describe how the state of the system evolves in the first three rounds.

$$round = 0 \Rightarrow h \subseteq ready \tag{5}$$
$$round > 0 \Rightarrow h = \varnothing \tag{6}$$
$$round = 1 \Rightarrow m \subseteq ready \tag{7}$$
$$round > 1 \Rightarrow m = \varnothing \tag{8}$$
$$round = 2 \Rightarrow u \subseteq ready \tag{9}$$
$$round > 2 \Rightarrow u = \varnothing \tag{10}$$

where

$$\begin{aligned} h &= \{n : Node \mid state(n) = head \wedge \\ &\qquad \exists\, m : Node \bullet (n,m) \in conn \wedge state(m) = head\} \\ m &= \{n : Node \mid state(n) = member \wedge \\ &\qquad \nexists m : Node \bullet (n,m) \in conn \wedge state(m) = head\} \\ u &= \{n : Node \mid state(n) = undecided\}\,. \end{aligned}$$

The formal proofs are omitted but informal reasoning for (5) and (6) is given below. The other proofs follow similarly.

Initially, *ready* includes all head nodes. Hence, (5) is true. Assume (5) holds before the execution of the loop. When the loop body executes, one of the following occurs.

(i) A set of nodes is removed from *ready*. These include a node which is changed from an undecided to a head node, and all of its neighbours which become members. All other nodes are unchanged, and hence (by the induction hypothesis) (5) holds.

(ii) A set of nodes is removed from *ready*. These include a head node, and all of its neighbours which become members. All other nodes are unchanged, and hence (by the induction hypothesis) (5) holds.

(iii) A single node is removed from *ready*. This node changes from a member to an undecided node. All other nodes are unchanged, and hence (by the induction hypothesis) (5) holds.

(iv) *ready* is reset to incude all head nodes, and hence (5) holds.

Hence, (5) is an invariant.

Next we consider (6). Initially, $round = 0$ and so (6) holds. Assume (6) holds before the execution of the loop. When the loop body executes, none of the commands which removes a node from *ready* introduces adjacent heads. Hence, (6) continues to hold. When the round increases from 0 to 1, since $ready = \varnothing$ before the command, from (5) there are no nodes in h before the command. Therefore, there are no nodes in h after the command, and (6) holds. Similarly, when the round increases to a number greater than 1, from the induction hypothesis $h = \varnothing$ before the command. Therefore, $h = \varnothing$ after the command, and (6) holds. Hence, by induction (6) is an invariant.

Given these invariants, we need only prove that *round* increases to ensure that (2) and (4) will become, and remain, true. Since each action either decreases $\#ready$, or increases *round* by 1 while also increasing $\#ready$ by at most $\#Node$, the following is a loop variant.

$$\#ready - (\#Node + 1)round$$

Since *ready* is finite and the above variant decreases, we deduce that *round* eventually increases.

IV. Trust values for node reliability

Peer-to-peer trust management systems [7], [15] set up virtual networks of trust within networks containing malicious, or otherwise untrustworthy, nodes. This is done in a decentralised fashion by allowing nodes within the network to adjust their trust values for other nodes based on a combination of reputations and direct observation. As a second example, we consider a simple algorithm for the local evaluation of trust in a network based purely on direct observation [8]. The notion of trust this algorithm is concerned with is the reliability of nodes and the connections to them. Over time a node will determine which of its communication partners reliably receive its messages and then send messages to them alone: unreliable nodes are excluded from receiving further messages.

The trust value a node n derives for another node m represents the probability that m will receive a message sent to it from n. To allow node n to determine this probability, the algorithm requires nodes to acknowledge messages from other nodes but, for efficiency, they do this only when they send a message to the other node (the acknowledgements are *piggybacked* on the sent message). Hence, the algorithm is referred to as the "delayed ack" method.

Following the approach of Section III, we model the "delayed ack" method as an initialised loop. In this case, to model node reliabilities we require the use of the probabilistic choice operator in the loop body. We focus on the derivation of trust values, abstracting from the contents of messages (which are not relevant to the trust evaluation method) and the subsequent use of the trust values in choosing to which nodes messages are sent.

Let *Node* be the set of all nodes in the network and $rel : Node \to [0,1]$ be a constant function returning the reliability of a node. The program has three variables: $sent(n,m)$ denotes the number of messages sent from a node n to a node m, $rec(n,m)$ denotes the number of messages received by n from m, and $trust(n,m)$ denotes n's estimate of m's reliability. Note that for an unreliable node m, $sent(n,m)$ will be greater than $rec(m,n)$.

Variables *sent* and *rec* are initialised to the constant function $\mathbf{0}$: the function mapping all domain values to 0. The initial value of *trust* is undefined. The loop body models a node n sending a message to a node m which is received with a probability of $rel(m)$ (with a probability of $1 - rel(m)$ it does **skip**).

If a message is received and $sent(m,n)$ is not zero, *i.e*, m has sent messages to n in the past, then m's trust value for n is updated to $rec(n,m)/sent(m,n)$ (the *mean metric* described in [8]). Note that we assume $rec(n,m)$ is available to m (piggybacked onto the message) rather than a set of acknowledgements. This simplifies the original algorithm in that a node m's observations of a node n are made with 100% reliability. When sets of acknowledgements are sent instead, only those acknowledgements not sent in a previous message to m are sent. Hence, if m misses a message from n, it misses the fact that n has acknowledged certain messages, affecting its evaluation of n's reliability.

```
var $sent, rec : Node \times Node \to \mathbb{N}$;
    $trust : Node \times Node \to [0,1]$ $\bullet$
$sent, rec := \mathbf{0}, \mathbf{0}$ ;
do [] $n, m : Node$ $\bullet$
        $sent(n,m) := sent(n,m) + 1$ ;
        $((rec(m,n) := rec(m,n) + 1$ ;
                if
                [] $sent(m,n) = 0 \to$ skip
                [] $sent(m,n) \neq 0 \to$
                        $trust(m,n) := rec(n,m)/sent(m,n)$
                fi)
          $_{rel(m)}\oplus$ skip)
od
```

To reason about loops involving probabilities, we reason about the *expected*, *i.e.*, mean, values of variables affected by the probabilities. We let $\mathsf{E}(x)$ denote the expected value of a variable x.

To verify the "delayed ack" method, we begin by showing that for all nodes n and m, $\mathsf{E}(rec(m,n)) = rel(m)sent(n,m)$ is an invariant. (Note that in this example, analysis of the program reveals that the distribution of *rec* is binomial, implying that the above invariant holds. Here we include a proof for completeness and to show how our approach can be applied more generally.)

Theorem 2

$$\forall\, n, m : Node \bullet \mathsf{E}(rec(m,n)) = rel(m)sent(n,m) \qquad (11)$$

Proof (by induction)

Initially,

$$sent = rec = \mathbf{0}$$
$$\Rightarrow \forall\, n, m : Node \bullet rec(m,n) = sent(n,m) = 0$$
$$\Rightarrow (11)$$

Assume (11) holds. When the loop is executed, two nodes n and m are selected and updated as follows (and all other nodes remain unchanged).

$$sent(n,m) = sent_0(n,m) + 1 \,\wedge \qquad 1$$
$$\mathsf{E}(rec(m,n)) = rel(m)(\mathsf{E}(rec_0(m,n)) + 1) + (1 - rel(m))\mathsf{E}(rec_0(m,n)) \qquad 2$$

From 2, we deduce

$$\mathsf{E}(rec(m,n))$$
$$= rel(m)\mathsf{E}(rec_0(m,n)) + rel(m) + \mathsf{E}(rec_0(m,n)) - rel(m)\mathsf{E}(rec_0(m,n))$$
$$= rel(m) + \mathsf{E}(rec_0(m,n)) \qquad 3$$

and from 3 and the induction hypothesis, we deduce

$$\mathsf{E}(rec(m,n))$$
$$= rel(m) + rel(m)sent_0(n,m)$$
$$= rel(m)(sent_0(n,m) + 1). \qquad 4$$

Hence from 4 and 1, we deduce (11). □

Given (11), we can show that the expected trust value calculated by a node n for a node m is node's m reliability.

$$\mathsf{E}(trust(n,m)) = \mathsf{E}(rec(m,n)/sent(n,m)) = rel(m).$$

What is also interesting in this example is the number of messages a node n needs to send to a node m before it can be assured that its *actual* (as opposed to expected) trust value for m is a good approximation of the reliability of node m [8]. Obviously, this will not be the case for any $rel(m)$ after sending one message when the trust value will be either 1 (if the message was received) or 0 (if the message was not received). Let us assume that the actual value is a good approximation when the standard deviation of its distribution σ is within 5% of the expected value. The standard deviation of a binomial distribution is the square root of $(np(1-p))$ where n is the number of trials, and p the probability of success. Hence,

$$\sigma = sqrt(sent(n,m)rel(m)(1 - rel(m)))$$

and a node n will have a good approximation of the expected value of trust for node m when

$$sqrt(sent(n,m)rel(m)(1 - rel(m))) = 0.05rel(m)sent(n,m)$$

So for a node m with a reliability of 0.9, it will take approximately 44 messages before a good approximation is achieved and for a node m with reliability 0.5, it will take approximately 400 messages.

V. Conclusion

In this paper, we have shown how conventional techniques for reasoning about programs can be used to reason about two simple, yet typical, self-organising systems. Use of an initialised possibly non-terminating loop to model such a system means that the method of invariants is available for establishing safety; and liveness may be reasoned about using a variant.

Our first example identified the use of functions to model the states of large, or arbitrary, collections of components, and 'rounds' to model fairness as two useful techniques. Stochastic behaviour is typical of such systems and was the feature of the second example. Existing methods provide simple techniques for reasoning about (a) expected values of variables, and (b) their distribution [9].

The examples were purposefully presented at a high level of abstraction. This simplifies reasoning about the functional properties of the systems before considering the complexities inherent in the protocols responsible for component interactions. The verification of such protocols would follow the type of analysis in this paper using step-wise refinement to incrementally add the required details to the programs. Such an approach is supported by the existing refinement calculus for guarded-command language programs [10], as well as that for probabilistic guarded-command language programs [9].

Most treatments of self-$*$ systems in general and self-organisation in particular have concentrated on designs that achieve particular, stated, forms of organisation. Rigorous treatments centre around self-organisation as decreasing entropy; for example see the paper by Polani [11]. By comparison this paper uses standard ideas from Formal Methods to validate typical designs.

To aid with the verification of more complex systems, we envisage the use of static analysis tools on the programs that model them. One possibility is to use a static analysis tool for concurrent programs, such as VCC [2], so that threads could be used to model nondeterministic choice of commands.

Acknowledgements

This work was supported by Australian Research Council (ARC) Discovery Grant DP110101211 and the Macao Science and Technology Development Fund under the EAE project, grant number 072/2009/A3. The authors acknowledge Larissa Meinicke for her insights into reasoning about probabilistic loops.

References

[1] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–555, 1988.

[2] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S.Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42. Springer-Verlag, 2009.

[3] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall International, 1976.

[4] A. Fehnker, L. van Hoesel, and A. Mader. Modelling and verification of the LMAC protocol for wireless sensor networks. In J. Davies and J. Gibbons, editors, *International Conference on Integrated Formal Methods (IFM 2007)*, volume 4591 of *LNCS*, pages 253–272. Springer, 2007.

[5] M. Gerla, T. J. Kwon, and G. Pei. On demand routing in large ad hoc wireless networks with passive clustering. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC 2000)*, volume 1, pages 100–105, 2000.

[6] H. Hamann and H. Wörn. A framework of space-time continuous models for algorithm design in swarm robotics. *Swarm Intelligence*, 2:209–239, 2008.

[7] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *12th International Conference on World Wide Web (WWW '03)*, pages 640–651. ACM, 2003.

[8] R. Kiefhaber, B. Satzger, J. Schmitt, M. Roth, and T. Ungerer. The delayed ack method to measure trust in organic computing systems. In *Self-Adaptive and self-Organizing Systems Workshop (SASOW'10)*, pages 184–189. IEEE Computer Society, 2010.

[9] A. K. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer, 2005.

[10] C. C. Morgan. *Programming from Specifications*. Prentice Hall International, second edition, 1990.

[11] D. Polani. Foundations and formalizations of self-organization. In M. Prokopenko, editor, *Advances in Applied Self-organizing Systems*, Advanced Information and Knowledge Processing, pages 19–37. Springer-Verlag, 2008.

[12] J. W. Sanders and G. Smith. Emergence and refinement. *Formal Aspects of Computing*, 24(1):45–65, 2012.

[13] G. Smith and J. W. Sanders. Formal development of self-organising systems. In *International Conference on Autonomic and Trusted Computing (ATC'09)*, volume 5586 of *LNCS*, pages 90–104. Springer-Verlag, 2009.

[14] A. Winfield, W. Liu, J. Nembrini, and A. Martinoli. Modelling a wireless connected swarm of mobile robots. *Swarm Intelligence*, 2:241–266, 2008.

[15] Li Xiong and Ling Liu. PeerTrust: supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):843–857, 2004.