Formal development of multi-agent systems using MAZE

Qin Li[a,*], Graeme Smith[b]

[a]Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China.
[b]School of Information Technology and Electrical Engineering, The University of Queensland, Australia

Abstract

MAZE is an extension of the Object-Z specification language supporting the specification and development of multi-agent systems (MAS). Following recommendations from the agent-oriented software engineering community, it supports three distinct levels of abstraction: (i) the *macro* level which focusses on the system's overall, global behaviour, independently of how the agents of the system operate and interact, (ii) the *meso* level which focusses on agent interactions, and (iii) the *micro* level which focusses on the operation of individual agents. Object-Z's high-level support for component-based specification, which is well suited to modelling MAS, is complemented in MAZE with support for action refinement to facilitate the top-down development process from the macro to micro level, and with a number of syntactic conventions aimed at abstractly specifying the low-level mechanisms required for dealing with asynchronous communication and timing constraints at the micro level. The latter are shorthands for existing Object-Z notation and so require no redefinition of Object-Z's semantics. In this paper, we provide an overview of MAZE and illustrate its use on a non-trivial case study: a swarm robotic algorithm for self-assembly.

1. Introduction

A *multi-agent system* (*MAS*) is a system comprising a number of interacting, *autonomous agents*, i.e., components which can initiate actions without external control. Our notion of an agent includes not only "intelligent" agents [1, 2], but also components which autonomously follow simple protocols such as the sensors in a self-organising sensor network [3], or the nodes of an *ad-hoc* mobile network which continually adapt their routing patterns to the current network topology [4].

Zambonelli and Omicini [5] argue that the disciplined engineering of MAS should proceed at three distinct levels of abstraction.

1. At the *macro* level, the engineer is concerned with the overall system functionality, ignoring the operation and interaction of its agents.

[*]Corresponding author.
Email addresses: qli@sei.ecnu.edu.cn (Qin Li), smith@itee.uq.edu.au (Graeme Smith)

2. At the *meso* level, the engineer considers potential agent interactions and interaction paradigms that will lead to the desired system functionality.
3. At the *micro* level, the engineer is concerned with the operation of individual agents, choosing an implementation that results in the required meso-level interactions.

A correct design of MAS should keep the three levels consistent so that the system functionality can be achieved by the interactions of the agents.

We have adopted this three-level approach in the formal development of multi-agent systems [6, 7, 8], leading to the development of an extension to Object-Z [9] called MAZE [10]. The extension supports action refinement [11] as a means of developing a specification from the macro level, where operations are coarse-grained, through to the micro level, where the granularity of operations is usually much finer. Simulation rules for checking action refinement provide proof obligations that designers should satisfy at each development step. We show through our case study that they are useful for detecting design problems as well as selecting solutions. The extension also involves a number of syntactic conventions, aimed at facilitating the specification of inter-agent communication mechanisms and associated timing constraints at the micro level. The syntactic conventions are analogous to those used in Z for modelling sequential systems [12]. That is, they are merely a shorthand for what could otherwise be expressed using more basic syntax. For this reason, they do not require an extension to the existing semantics of Object-Z.

In this paper, we extend the work in [10] by providing a more practical verification method for action refinement in MAZE and proving it sound with respect to a high-level, trace-based definition of action refinement. We also apply MAZE to a non-trivial case study: a swarm robotic self-assembly algorithm based on the work of Støy and Nagpal [13, 14]. We begin by introducing our case study in Section 2. In Section 3, we introduce MAZE and our proof method for action refinement in MAZE along with the proof of its soundness. In Section 4, we illustrate the use of MAZE for incrementally developing specifications of MAS from the macro to the micro level including the use of syntactic conventions for modelling individual agents and their interactions at the micro-level. Related work is discussed in Section 5 before we conclude the paper in Section 6.

2. A gradient-based approach to self-assembly

Self-assembly algorithms allow a swarm of robots to autonomously form a shape or pattern appropriate for a given task. Many different algorithms have been devised for the process of *morphogenesis*, i.e., "growing" the pattern or shape from an unordered swarm of robots [15, 16, 14, 17, 18]. In this section, we describe the approach of Støy and Nagpal [13, 14]. Their algorithm utilises virtual gradients created and propagated by the robots in the swarm to recruit other robots in order for the mass to assemble into a required 3-dimensional shape. The self-assembly algorithm we verify in the paper is based loosely on this work; the differences are discussed below.

2.1. Støy and Nagpal's algorithm

The robots, referred to as *modules* by Støy and Nagpal, form a connected mass. Each robot is capable of local communication with immediate neighbours only, i.e., those with

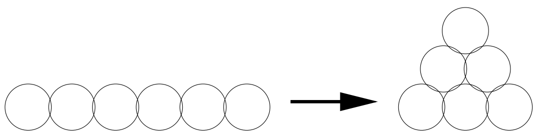
Figure 1: Two-dimensional representation of a connected mass of robots forming a desired shape.

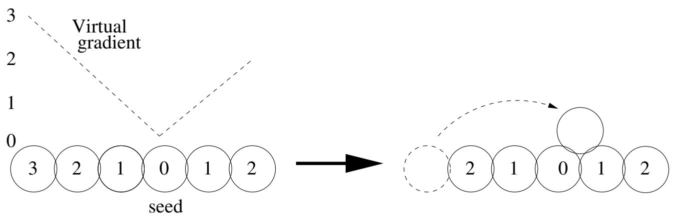

Figure 2: Robots follow a gradient established by the seed robot to reach a position in the target shape.

which they have physical contact. They are also capable of movement over or around their neighbours to form a required shape (see Figure 1).

Initially, one robot, called the *seed*, is provided with a representation of the shape to be formed (which we will refer to as the *target*). The seed assumes a particular position in this target, and then *recruits* other atoms to join the target. This is achieved by setting up a *recruitment gradient* to attract other robots to neighbouring positions in the target. The recruitment gradient is a virtual gradient, or slope, represented by integer values stored by the robots (see left-hand side of Figure 2). It is set up by the seed robot storing an integer value, say 0, then incrementing that value by 1 and broadcasting the result to each of its neighbours. These store the received value, increment it by one and then broadcast it to each of their neighbours. The result is that each robot stores its distance from the seed (see left-hand side of Figure 2).

A robot follows a gradient by moving from a position next to a robot with gradient value n to a position next to another robot with gradient value $n - 1$. By following a gradient in this way it eventually reaches the seed (see right-hand side of Figure 2). The seed passes such a robot the target representation and it takes on one of the vacant target positions. After this, it acts like a seed to recruit any neighbours it requires.

For the algorithm to work, Støy and Nagpal identify two constraints. The first of these is that the robots must remain connected. If one or more robots becomes disconnected from the rest, they can no longer receive messages (see Figure 3). Hence, an additional virtual gradient, called the *connection gradient*, is initially established by the seed, and a robot can only move if its distance from the seed is greater than or equal to those of its neighbours, i.e., it is not required to connect its neighbours to the mass.

The second constraint is that the desired shape must have enough spaces in it to allow robots to move to any required position. This is ensured by requiring that target shapes conform to a particular porous "scaffold" structure. Other than conforming to such a structure there are no constraints on the target shape except that it must be a connected mass of robots. Various methods for efficiently representing the target within a robot have been proposed by Støy and Nagpal; in this paper we abstract from such details.

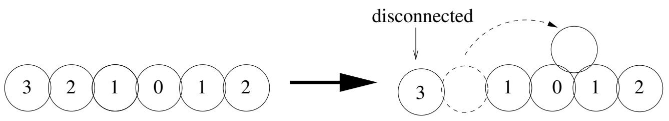

Figure 3: Robots should not become disconnected from the swarm as a consequence of other robots moving.

2.2. Modifications to the algorithm

The algorithm we verify in this paper differs from that of Støy and Nagpal in two aspects. Firstly, they do not discuss the issue of multiple, intersecting gradients. This can occur due to more than one robot creating a gradient, or when a single gradient returns to a robot due to a loop in the swarm structure. To deal with this, we require a robot which already has a recruitment gradient value to ignore any values that it receives which would result in increasing this gradient value. This results in robots storing a value representing the *shortest* distance to a robot which is the seed of a recruitment gradient.

To ensure progress under this approach, we require also that, once a robot has all the neighbours it needs, the gradient leading to it is 'cancelled' (thus allowing other gradients to propagate). This proposal will be specified, and verified to work, when we consider the meso-level model of the algorithm in Section 4.

The second change to the algorithm of Støy and Nagpal is that we do not have a connection gradient. Instead, we use the recruitment gradients to determine when an atom is closer than its neighbours to a seed atom in the target. If it is closer (such as the leftmost atom with gradient value 2 in Figure 3), than that atom should not move since its neighbours may rely on it staying in position to remain connected to the mass. We refer to such neighbouring atoms as being *dependent* on the atom which is closer to the seed.

3. MAZE

3.1. Macro-level specification

A macro-level specification in MAZE is captured by the class construct of Object-Z [9]. It encapsulates a collection of type and constant definitions, a schema describing the specification's state space (in terms of variables and an invariant), a schema defining the initial state, and a set of operations defining possible agent actions. A macro-level specification has the form:

Each action in MAZE is defined with an Object-Z operation schema of the following form:

Act_i

$\Delta(\vec{u})$

declarations of local variables $\vec{x}$

$body(\vec{c}, \vec{v}, \vec{x}, \vec{v}')$

where $\vec{u}$ is the subset of the state variables $\vec{v}$ whose values can be changed by the operation (all other variable values remain unchanged), and $\vec{v}'$ denotes the post-state values of the variables $\vec{v}$. Semantically, Object-Z operations are *guarded*. When the predicate $body(\vec{c}, \vec{v}, \vec{x}, \vec{v}')$ cannot be satisfied then the operation cannot occur. This is in contrast to Z operations which can occur at any time but have undefined behaviour when their predicate cannot be satisfied [12].

System

declarations of types $\vec{t}$ and constants $\vec{c}$

declarations of state variables $\vec{v}$

$inv(\vec{c}, \vec{v})$

Init

$init(\vec{c}, \vec{v})$

Act_1

$\vdots$

Act_n

Often the system is modelled at the macro level by a single action which reaches the desired state. Termination in the desired state is guaranteed by making the guard of the action evaluate to false in the desired state. In general, when there is more than one action, we need to prove that the specification reaches a state where no action is enabled and that the negation of the disjunction of the actions' guards implies the desired state.

This approach will not always reflect the ongoing reactive behaviour of the system we are modelling; however, it is sufficient for verifying, through the successive refinement of the action to a sequence of finer-grained actions at the meso and micro level, that a particular MAS design produces a desired goal under certain conditions (captured by the initial state schema). This will be demonstrated in Section 4.

3.2. Meso-level specification and action refinement

Macro-level specifications abstract from interactions between agents, focussing instead on the outcomes of those interactions. The goal at the meso level of development is to decompose the abstract actions of the macro level to actions representing agent interactions. The latter are still in terms of the global state of the system and act as a bridge between the macro level and micro level where individual agent behaviours are specified.

Adding the interactions as we develop the specification to the meso level, and ultimately the micro level, requires the addition of further actions, e.g., to model the sending and receiving of messages. Derrick and Boiten [19] define notions of *weak* and *non-atomic refinement* for Object-Z which allow an abstract operation to be refined to a sequence of concrete ones. These are not ideal for our purposes, however, as they do not allow guards to be strengthened. The complexity of individual agents often arises from their "intelligent" decision-making procedures. These procedures determine whether an agent performs a particular action in a given context. At a high-level of abstraction, we would like to ignore such procedures by leaving the choice of actions nondeterministic. Adding them at a lower level of abstraction would then require that the occurrence of certain actions be restricted, i.e., their guards strengthened. We therefore base our approach on the simulation rules for action refinement in action systems by Back and von Wright [11]. Here we consider the forward simulation rules only, and adapt them for Object-Z. The backwards simulation rules could be similarly adapted.

An action system has a state space, an initialisation condition, and a set of actions. The actions have guards which determine when they are enabled. Action systems behave by repeatedly executing enabled actions until none are enabled, or an enabled action *aborts* (its precondition is violated). For Object-Z, there are no aborting states since the guard of an operation guarantees that the operation's definition can be satisfied. A state in which no actions are enabled is called a *terminating state*.

A computation of an action system is a finite or infinite sequence of states generated by an execution of the action system. The first state of a computation satisfies the initial condition. Each two adjacent states correspond to a state transition satisfying an action. A finite computation ends with a terminating state.

Action refinement allows us to develop several concrete actions whose sequential composition simulates an abstract action. When action refinement is considered between two action systems, we only consider their observable behaviours. A pair of mappings $f : \Sigma_A \rightarrow \Sigma_E$ / $g : \Sigma_C \rightarrow \Sigma_F$ map the state of the abstract/concrete system to the state of observable variables. A relation $h : \Sigma_E \leftrightarrow \Sigma_F$ from the observable abstract state space to the observable concrete state space is defined to indicate which states are considered equivalent in both systems. In many cases, h is an identity relation and all variables of the abstract system are observable (i.e., $\Sigma_A = \Sigma_E$). A transition is called a *stuttering transition* of the abstract/concrete system if its pre-state and post-state are mapped to the same observable state according to f/g; otherwise, the transition is called a *change transition*.

Given a computation c of the abstract/concrete system, we can obtain an observable trace by removing any stuttering transitions. Let A be the abstract system and C be the concrete system. We use the notation $tr(A, f)$ and $tr(C, g)$ to denote the set of observable traces in A and C respectively. In general, we can define a retrieve relation $R \mathrel{\widehat{=}} f;\ h;\ g^{-1}$ relating the abstract state space to concrete state space, i.e., $R : \Sigma_A \leftrightarrow \Sigma_C$.

Definition 1. (Action refinement) Let C and A be action systems with mappings f, g and relation h. $R = f;\ h;\ g^{-1}$ is a retrieve relation. We say C is a refinement of A with respect to R, denoted by $C \sqsupseteq_R A$, if and only if for any finite trace t in $tr(C, g)$, there exists a finite trace s in $tr(A, f)$ such that $\#s = \#t$ and $\forall i : 1..\#t \bullet (s[i], t[i]) \in h$; and for any infinite trace t in $tr(C, g)$, there exists an infinite trace s in $tr(A, f)$ such that $\forall i : \mathbb{N} \bullet (s[i], t[i]) \in h$. $\diamond$

The definition of action refinement implies that

1. for any transitions in the concrete system from and to observable states, there is a transition in the abstract system from and to observable states that are related by the retrieve relation R;
2. if an abstract observable trace can be extended to an observable state, the related concrete trace can be extended to a related observable state.

For a given Object-Z specification, the change and stuttering transitions are particular *occurrences* of the specification's operations, i.e., particular pre-state/post-state pairs that satisfy an operation's predicate. A single operation can have some occurrences which are change transitions, and some which are stuttering transitions, depending on whether or not the post-state of the operation is related to the same observable state as the pre-state. An example of this will be given when we return to our case study below.

Proving action refinement using Definition 1 is not practical due to the need to perform checks on traces. Hence we introduce a simulation-based approach which simplifies the refinement checking to checks on actions. Although the conditions (inspired by those of Event-B [20]) are slightly stronger than Definition 1, they are much simpler to check. The rules are for forward simulation; backward simulation could be similarly defined.

Definition 2. (Forward Simulation) Let A be an Object-Z class with state schema $AState$, initial state schema $AInit$, and operation occurrences partitioned into change transitions $AChange_0, \ldots, AChange_n$ and stuttering transitions $AStutt_0, \ldots, AStutt_m$ for some $n, m : \mathbb{N}$. Similarly, let C be an Object-Z class with state schema $CState$, initial state schema $CInit$, and operation occurrences partitioned into change transitions $CChange_0, \ldots, CChange_l$ and stuttering transitions $CStutt_0, \ldots, CStutt_k$ for some $l, k : \mathbb{N}$. A is refined by C when there exists a retrieve relation $R : \Sigma_A \leftrightarrow \Sigma_C$ (modelled by a Z schema as in [19]) which relates the states of A to those of C such that the following hold. (Schemas are used below as declarations and predicates as in Z [12]. The Z notation $\mathrm{pre}\, A$ returns the guard of operation A.)

Initialisation: Initialisation in C simulates initialisation in A.

$$\forall\, CState \bullet CInit \Rightarrow (\exists\, AState \bullet AInit \land R)$$

Action Simulation: Any change transition $CChange_c$ in C simulates some change transition $AChange_a$ in A; any stuttering transition $CStutt_c$ in C simulates the identity transition in A.

$$\forall\, AState, CState, CState' \bullet R \land CChange_c \Rightarrow (\exists\, AState' \bullet AChange_a \land R')$$

$$\forall\, AState, CState, CState' \bullet R \land CStutt_c \Rightarrow (\exists\, AState' \bullet ID_A \land R')$$

where ID_A is the identity operation on $AState$.

Termination: Any terminating state in C is related only to terminating states in A.

$$\begin{aligned}&\forall\, AState, CState \bullet\\ &\quad R \land \neg\ \mathrm{pre}(CChange_0 \lor \ldots \lor CChange_l \lor CStutt) \Rightarrow\\ &\qquad \neg\ \mathrm{pre}(AChange_0 \lor \ldots \lor AChange_n \lor AStutt)\end{aligned}$$

where $AStutt = (AStutt_0 \lor \ldots \lor AStutt_m)$ and $CStutt = (CStutt_0 \lor \ldots \lor CStutt_k)$.

Stuttering Convergence: Any state in C from which infinite stuttering is possible is related only to states in A from which infinite stuttering is possible.

$$\begin{aligned}&\forall\, AState, CState \bullet\\ &\quad R \land (\forall\, i : \mathbb{N} \bullet \exists\, CState' \bullet CStutt^i \land (\mathrm{pre}\, CStutt)') \Rightarrow\\ &\qquad (\forall\, j : \mathbb{N} \bullet \exists\, AState' \bullet AStutt^j \land (\mathrm{pre}\, AStutt)')\end{aligned}$$

where T^i means sequentially performing transition T for i times. $\diamond$

To prove the above simulation rules are sound, we need to show that they are sufficient to imply action refinement as defined in Definition 1.

Soundness Proof: For any computation c of a concrete system C and its observable trace t belonging to $tr(C, g)$, we need to construct a trace s belonging to $tr(A, f)$ such that if t is finite, $\#s = \#t$ and $\forall\, i : 1..\#t \bullet (s[i], t[i]) \in h$, and if t is infinite s is infinite and $\forall\, i : \mathbb{N} \bullet (s[i], t[i]) \in h$. The constructive proof is as follows.

(1) According to the Initialisation rule, there exists an abstract initial state σ_1 which is related to $c[1]$ by R, i.e., $(f(\sigma_1), t[1]) \in h$. We let $s = \langle f(\sigma_1) \rangle$ for any such σ_1.

(2) For each further state $c[i]$ of c in turn, we proceed as follows. The transition $(c[i-1], c[i])$ is either a change transition or a stuttering transition.

(2.1) If it is a change transition, according to the Action Simulation rule, there is an abstract state σ_i such that the pair (σ_{i-1}, σ_i) belongs to a change transition of A and $(f(\sigma_i), t[i]) \in h$. We append $f(\sigma_i)$ to s.

(2.2) If it is a stuttering transition, we have $g(c[i-1]) = g(c[i])$ and $c[i]$ is removed in t. According to the Action Simulation rule, σ_{i-1} is related to $c[i]$ by R.

We then have $\#s = \#t$ and for $i \in 1..\#t$, $(s[i], t[i]) \in h$ if t is finite, and s is infinite and for $i \in \mathbb{N}$, $(s[i], t[i]) \in h$ if t is infinite. Next we need to show that s cannot be extended beyond the observable transitions of t when t is finite.

(4) According to the Termination rule, if c ends with a terminating state, all abstract states related by R are terminating states in A.

(5) The Stuttering Convergence rule guarantees that if the concrete computation c diverges, i.e., c is infinite while t is finite ending in a divergent state from which infinite stuttering is possible, the abstract trace s also ends in a divergent state. In the case where the abstract system has no stuttering transitions, the Stuttering Convergence rule guarantees that c does not diverge. $\square$

In the case where all transitions in the abstract system are change transitions, the stuttering convergence condition can be simplified to require that the execution of stuttering transitions in the concrete system converges, i.e., they can be only executed a finite number of times. It can be expressed as

$$\forall\, \mathit{CState} \bullet \exists\, N : \mathbb{N} \bullet \forall\, \mathit{CState}' \bullet \mathit{CStutt}^N \Rightarrow \neg \text{ pre } \mathit{CStutt}(\mathit{Cstate}') \qquad (*)$$

for a natural number N.

In general cases, it is sufficient to prove the above convergence condition by constructing a variant W which is a natural number or a finite set so that every stuttering transition in concrete system C decreases it, e.g.,

$$\forall\, \mathit{CState}, \mathit{CState}' \bullet \mathit{CStutt} \Rightarrow W' \subset W\ .$$

With the variant, the proof can be done by just considering the stuttering transitions once rather than checking its iterative executions. However, in many complex cases, constructing such a variant is complicated. In such cases, we have to analyse the behaviours of the stuttering transitions and prove condition $(*)$ is true.

3.3. Micro-level specification

At the micro level of development we produce a specification of the local behaviour of individual agents. A specification (syntactically and semantically equivalent to a MAZE macro-level specification) is provided for each type of agent in the system. A further system specification captures the collection of agents and their effect on their environment.

To show that such a collection of agents refines the meso-level specification, we need to be able to reason about the mechanisms by which the agents interact. In MAZE, all agents are modelled as interacting via asynchronous message-passing. The justification for this is as follows. As there is no centralised control, messages may be sent to an agent at any time, including when it is busy with another message. Hence, messages need to be buffered (since allowing messages to be lost would greatly complicate the simple micro-level protocols we are trying to establish). In the implementation of a MAS, the buffering may be part of the communication medium, e.g., when agents are distributed over a network, or part of the agent, e.g., when communication is wireless and effectively synchronous between agents. The robot self-assembly case study of this paper is an example of the latter, where agent-to-agent communication occurs via direct connections between atoms which are in physical contact with each other.

The system specification at the micro level differs from those at other levels in that it refers to the specification(s) of individual agents. The semantics, based on that of object instantiation in Object-Z [9], enables agent instances to be declared and the state of such instances to be accessed using the standard notation from object orientation, e.g., the notation $a.v$ denotes the state variable v of agent a. We also allow local types and constants of agent specifications to be accessed via dot notation, e.g., $A.T$ denotes the local type T of agent specification A. For each agent specification in MAZE, these local types include a type **message** defined as a Z free type. It defines the kinds of messages the agent can send and receive.

Although it is possible to realise the asynchronous message passing in standard Object-Z, it can lead to specifications which are awkward to read due to the details of the particular system under development being intermingled with those of the underlying communication mechanisms. In MAZE, we separate these details by capturing the latter implicitly via a number of syntactic conventions.

Special syntax is introduced to specify a collection of interacting agents. $\mathbb{T}\,A$ defines a topology of agents of type A in terms of a finite function whose domain is the set of all agents in the topology and maps each such agent to the agents to which it can send messages, *i.e.*, $\mathbb{T}\,A = (A \ffun \mathbb{F}\,A)$. Note that there are no constraints on the function allowing uni-lateral sending of messages, and agents which are isolated and unable to send or receive messages. Typically, constraints will be added in the specification to restrict the function as required.

The notation $\mathbb{T}\,A$ not only introduces a topology of agents of type A, but also implicitly introduces their initialisation (according to the initial state schema of A) and system actions allowing any agent in the topology to perform any enabled action. These actions send messages to and receive messages from an implicit global buffer. The buffer is unordered allowing messages to be received by the agent in a different order to which they are sent. This may model the use of different routes through a communication medium such as the Internet, or the ability of an agent to prioritise messages in its internal buffer.

Finally, the system specification may include explicitly defined *system actions* that change the agents' environment and topology. Such actions may occur either independently (when the environment itself can change) or in response to an agent action. In the latter case we follow the name of the system action with a tag $< a : \mathrm{dom}\, t \bullet a.Act >$ (where t is an object topology). The tag declares an agent a belonging to the topology t, and an action of that agent A which must occur for the system action to occur. The agent a declared by the tag may be referenced throughout the system action's definition.

Definition 3. (System specification) The following two specifications are semantically equivalent where $Act_1, \ldots, Act_n$ are the actions of agent specification A. ($[]$ is the Object-Z distributed choice operator which is semantically equivalent to an existential quantifier [9]).

S

$t : \mathbb{T}\, A$

$SysAct_1$

details of $SysAct_1$

$SysAct_2 < a : \mathrm{dom}\, t \bullet a.Act_1 >$

details of $SysAct_2$

S

$t : A \twoheadrightarrow \mathbb{F}\, A$

$\mathbf{buffer} : \mathrm{bag}(A.\mathbf{message} \times A \times A)$

$\textsc{Init}$

$\forall\, a : \mathrm{dom}\, t \bullet a.\textsc{Init}$

$\mathbf{buffer} = [\![\ ]\!]$

$SysAct_1$

details of $SysAct_1$

$$\begin{array}{l} SysAct_2 \mathrel{\widehat{=}}\ []\ a : \mathrm{dom}\, t \bullet \\ \qquad\qquad a.Act_1 \wedge [\mathit{details\ of\ SysAct_2}] \\ Act_2 \mathrel{\widehat{=}}\ []\ a : \mathrm{dom}\, t \bullet a.Act_2 \\ \vdots \\ Act_n \mathrel{\widehat{=}}\ []\ a : \mathrm{dom}\, t \bullet a.Act_1 \end{array}$$

where the implicit variable **buffer** models the unordered, global buffer as a bag (allowing an agent to send a message multiple times). Each element of the buffer is a tuple (m, a, b) where m is a message, a is the agent which sent m, and b is the agent to which the message has been sent. $\diamond$

For specifying agent actions in MAZE, two message-related predicates are introduced: **send** for modelling messages being sent to the global buffer (**send**(m, a) models a message m being sent to agent a, and **send**(m) models a message being broadcast to all connected agents) and **receive** for modelling messages being received from the buffer (**receive**(m, a) models the receipt of a message m from agent a). Each action in an agent specification may have a single **receive** predicate (as part of its guard) and a single **send** predicate (as part of its postcondition).

A predicate **progress**(s, r) is also introduced for use in agent specifications. It is true when the system has progressed to a point where all messages in set s that have been sent by the agent and all messages in set r that have been sent to the agent have been received. This mechanism provides a way of abstracting from the use of timers and timing constraints required in an implementation of the MAS [21, 22].

For agents of type A, the formal definitions of **send**, **receive** and **progress** are given in terms of an implicit variable **buffer** in the system specification as follows.

Definition 4. (Message passing) When we apply the agent action Act (using the notation $a.Act$) in a MAS specification with a topology of agents t, the **receive** and **send** predicates introduce an additional guard G and additional postcondition $\mathbf{buffer}' = (\mathbf{buffer} \uminus R) \uplus S$ to the system operation where

- G is $(m, b, a) \sqsubseteq$ **buffer** when Act includes $\mathbf{receive}(m, b)$, and *true* otherwise.
- R is $\llbracket (m, b, a) \rrbracket$ when Act includes $\mathbf{receive}(m, b)$, and $\llbracket\, \rrbracket$ otherwise.
- S is $\llbracket (m, a, b) \rrbracket$ when Act includes $\mathbf{send}(m, b)$ and $b \in t(a)$, and, given $t(a) = \{b_1, \ldots, b_n\}$, $\llbracket (m, a, b_1), \ldots, (m, a, b_n) \rrbracket$ when Act includes $\mathbf{send}(m)$, and $\llbracket\, \rrbracket$ otherwise. ◇

Definition 5. (Progress) When we apply the agent operation Act (using $a.Act$), each predicate of the form $\mathbf{progress}(s, r)$ where s and r are of type $\mathbb{P}\,\mathbf{message}$ introduces an additional guard:

$$\forall\, b : A \bullet (\forall\, m : s \bullet (m, a, b) \not\sqsubseteq \mathbf{buffer}) \wedge (\forall\, m : r \bullet (m, b, a) \not\sqsubseteq \mathbf{buffer}) \qquad \diamond$$

To illustrate the use of these message-related predicates, imagine a system in which an agent $a : Agent$ broadcasts a request for assistance with a certain task. Those neighbouring agents that are able to assist respond to the broadcast message, and all others ignore it. Agent a needs to wait for all responses before it can proceed with dividing the task. Since it doesn't know how many responses it is waiting on, this would probably be implemented by waiting for a given time. In MAZE, we would abstract from this timing constraint using the **progress** predicate (see below).

The operation for sending a request would be specified with a **send** predicate with a single parameter (*request* : **message**). The operation for receiving and responding to a message would be specified with a **receive** and **send** predicate. In this case, the **send** predicate would have a second parameter to specify that the message is sent only to the agent which initiated the request. The operation for ignoring a request would have just a **receive** predicate. Of course, the operations would have additional predicates (elided below) relating to when they would send a request, and under what circumstances they would respond or ignore a request they have received.

SendRequest
$\mathbf{send}(request)$
$\ldots$

Respond
$a : Agent$

$\mathbf{receive}(request, a)$
$\ldots$
$\mathbf{send}(response, a)$

IgnoreRequest
$a : Agent$

$\mathbf{receive}(request, a)$
$\ldots$

The agent a which sent the request would also have an operation to receive responses, and another to divide the task once all responses have been received. The latter should only occur after all connected agents have received the request, and all subsequent responses have been received by a. In a real system, this would involve a timing constraint based on maximum times for message transmission between connected agents, and maximum message processing and response times [21, 22]. In MAZE, we abstract from this using a **progress** predicate which does not allow the operation to occur while either request messages from a, or *response* messages to a, remain in the buffer, i.e., while such messages have not yet been received.

ReceiveResponse

$b : Agent$

$\mathbf{receive}(response, b)$

$\ldots$

DivideTask

$\mathbf{progress}(request, response)$

$\ldots$

4. Case study

In our earlier work [10], we illustrated MAZE on a simple leader-election protocol employed in the cluster-based routing protocol of Gerla et al. [4]. Here we apply it to a more significant case study: the development of Støy and Nagpal's self-configuring robot swarm.

4.1. Macro level

System1

$[Position]$

$target : \mathbb{F}\, Position$

$atoms : \mathbb{F}\, Position$

$\#atoms = \#target$

Init

$true$

Configure

$\Delta(atoms)$

$atoms \neq target$

$atoms' = target$

The initial macro-level specification *System*1 has a constant *target* which is a finite set of positions in 3-dimensional space representing the desired configuration of the robots. A variable *atoms* represents the positions of the finite set of robots, referred to in this paper as 'atoms'. An invariant constrains the number of atoms to be the same as the number of positions in the target shape. We introduce the type *Position* without constraining it in any way. We could be more precise and define positions to consist, for example, of a triple of real numbers. For our purposes, however, the more abstract definition suffices.

The behaviour is modelled by a single action which reaches the desired goal. Initially, the positions of the atoms are not constrained. The action *Configure* captures the desired goal of the system: placement of an atom at each target position. It is enabled at most once after which the system's behaviour terminates.

In the more concrete macro-level specification *System*2, we distinguish those atoms that are fixed in a target position from those that are still able to move despite being in a target position: the former represented by a variable *placed*. There is no explicit restriction on *placed* initially, although it is implicitly restricted via the invariant which requires it to be a subset of $atoms \cap target$.

The action *Place* fixes a single atom in a target position, modelled by increasing the number of atoms in the set *placed*. It also allows the set *atoms* to change in

System2

$[Position]$

$target : \mathbb{F}\, Position$

$atoms : \mathbb{F}\, Position$

$placed : \mathbb{F}\, Position$

$\#atoms = \#target$

$placed \subseteq atoms \cap target$

Init

$true$

Place

any way that satisfies the invariant. The specification performs *Place* once for each atom that is not initially in *placed*. After this, the specification terminates with $placed = target$ and hence due to the invariant the desired state, $atoms = target$.

The mappings f and h for this refinement step are identity relations. The mapping g captures the fact that the atoms are only regarded as being observed when the target is achieved. To improve the readability, we mark the variables in both specifications with subscripts , i.e., v_1 denotes variable v from $System1$, v_2 denotes v from $System2$, and write the observable variables using a sans serif font.

$$\begin{aligned}
&f \mathrel{\widehat{=}} target_1 = \mathsf{target}_1 \wedge atoms_1 = \mathsf{atoms}_1\\
&h \mathrel{\widehat{=}} \mathsf{target}_1 = \mathsf{target}_2 \wedge \mathsf{atoms}_1 = \mathsf{atoms}_2\\
&g \mathrel{\widehat{=}} \mathsf{target}_2 = target_2 \wedge (placed_2 = target_2 \Rightarrow \mathsf{atoms}_2 = atoms_2)
\end{aligned}$$

The refinement from $System1$ to $System2$ can be verified using the retrieve relation $R \mathrel{\widehat{=}} f;\ h;\ g^{-1}$ relating the states from $System1$ to states from $System_2$.

$$R \mathrel{\widehat{=}} target_1 = target_2 \wedge (placed_2 = target_2 \Rightarrow atoms_1 = atoms_2)$$

For brevity, we directly give the definition of the retrieve relation R for all subsequent refinement steps in the case study instead of f, g and h.

The division into change and stuttering actions is according to the retrieve relation R. Every occurrence of *Configure* is a change action in $System1$. For $System2$, only the occurrence of *Place* where the post state establishes $placed_2 = target_2$ is a change action. Its other occurrences are stuttering actions. In this case, the proof of the rules of Definition 2 is straightforward.

Initialisation. Trivially holds since *Init* of $System1$ is true.

Action Simulation. Holds since each occurrence of *Place* where $placed_2 = target_2$ simulates *Configure*, and each other occurrence of *Place* simulates the identity transition of $System1$.

Termination. Holds since $System2$ terminates only when $placed_2 = target_2$ which is related to the terminating state $atoms_1 = target_1$ of $System1$.

Stuttering Convergence. The stuttering occurrences of *Place* in $System2$ will terminate when all atoms are in target positions. To prove this we use the number of atoms not in a target position as a variant, i.e., $W = \#target - \#placed$.

4.2. Meso-level

The goal at the meso level of development is to decompose the abstract actions of the macro level to actions representing agent interactions. The latter are still in terms of the global state of the system and act as a bridge between the macro level and micro level where individual agent behaviours are specified.

We begin developing the meso-level specification of our case study by decomposing action *Place* from *System*2. The strategy for placing robots in our target implementation is to have robots which are already placed recruit them. Each *Place* action could therefore be decomposed into a sequence of two concrete actions: the first corresponding to the recruitment, and the second to the recruited robot moving. This is specified in the class *System*3. To specify that only atoms with vacant neighbouring positions send recruitment messages, we introduce a constant $nb : Position \leftrightarrow Position$ denoting the neighbour relation between positions. We assume that the target is fully connected, i.e., $\forall\, p, q : target \bullet (p, q) \in nb^*$. To ensure an atom moves only when it is recruited, we introduce a variable $recruiters : \mathbb{F}\, Position$ to denote those atoms that recruited others.

*System*3

$[Position]$

$target : \mathbb{F}\, Position$
$nb : Position \leftrightarrow Position$

$\forall\, p, q : target \bullet (p, q) \in nb^*$

$atoms : \mathbb{F}\, Position$
$placed : \mathbb{F}\, Position$
$recruiters : \mathbb{F}\, Position$

$\#atoms = \#target$
$placed \subseteq atoms \cap target$
$recruiters \subseteq placed$

INIT
$recruiters = \varnothing$

⋮

The action *Place* is then replaced by two concrete actions *Recruit* and *Move* defined as follows.

Recruit
$\Delta(recruiters)$
$p, q : Position$

$(p, q) \in nb$
$p \in placed \setminus recruiters$
$q \in target \setminus placed$
$recruiters' = recruiters \cup \{p\}$

Move
$\Delta(atoms, placed)$
$p, q : Position$

$(p, q) \in nb$
$p \in recruiters$
$q \in target \setminus placed$
$placed' = placed \cup \{q\}$

The action *Recruit* corresponds to an atom at position p recruiting an atom for a vacant neighbouring position q. The action *Move* corresponds to an atom moving to a vacant position neighbouring a 'recruiter' atom. Note that the set *atoms* is changed implicitly by this action.

In order to ensure that the above design of the system is correct, we need to check that *System*3 refines *System*2. Let $R2$ be the retrieve relation between the states of *System*2 and *System*3. We consider every variable in *System*2 to be mapped to the variable with the same name in *System*3.

$$R2 \mathrel{\widehat{=}} (atoms_2 = atoms_3) \land (target_2 = target_3) \land (placed_2 = placed_3)$$

Since $R2$ equates the variables *atoms*, *target* and *placed* of *System*2 and *System*3, these variables are linked via the mappings f, g and h, and hence are observable. Therefore, every occurrence of *Place* (since it changes variable *placed*) is a change action.

Similarly, every occurrence of *Move* (since it changes *atoms* and *placed*) is a change action. Occurrences of *Recruit* change only the variable *recruiters*. We choose this variable to be non-observable. Hence, all occurrences of *Recruit* are stuttering actions.

Initialisation. It is trivial to prove $System3.\textsc{Init} \Rightarrow System2.\textsc{Init} \wedge R2$.

Action Simulation. We can easily prove that *Move* simulates *Place*. It is also trivial to prove that stuttering action *Recruit* simulates $ID_{System2}$ since it only changes the new variable *recruiters*.

Termination. The termination condition of $System2$ is $placed = target$. To satisfy this rule, the termination condition of $System3$ should not be stronger. In fact, $System3$ will terminate when both *Recruit* and *Move* are not enabled. According to their definition and the invariant $recruiters \subseteq placed$, the termination condition of $System3$ holds when either

(a) there is no $p \in placed$ or,

(b) there is such a p, and there is no $q \in target \setminus placed$ such that $(p, q) \in nb$.

In case (b), since *target* is fully connected and $placed \subseteq target$, we can deduce that $placed = target$. Hence, $\neg\, pre(Recruit \vee Move)$ is $placed = \varnothing \vee placed = target$. However, $\neg\, pre(Place)$ is $placed = target$ and so, with $R2$ as the retrieve relation, the termination condition does **not** hold.

This is a typical example where checking simulation rules can help in detecting design problems during development. It tells the designer that something needs to change in the concrete specification. One possible solution is to weaken the guard of *Recruit* so that it is enabled even when $placed = \varnothing$. However, this does not correspond to the design we are aiming at where only atoms fixed in target places recruit other atoms. Similarly, it does not make sense to weaken the guard of *Move* to allow it to occur when $placed = \varnothing$. The other possibility is to strengthen the invariant of the concrete specification to exclude states where $placed = \varnothing$. This can be done either explicitly, or implicitly by strengthening the initial state to include $placed \neq \varnothing$ (since elements are never removed from *placed* by any action). The latter solution corresponds to the algorithm of Støy and Nagpal where a single (seed) atom is placed initially. This atom is the one that starts the self-assembly process. We modify the Init of $System3$ to capture this approach.

```
── Init ─────────────────────
│ $placed \neq \varnothing$
│ $recruiters = \varnothing$
└────────────────────────────
```

This modification does not affect the correctness of the Initialisation or Action Simulation conditions proved above. It illustrates a process of problem detection and iterative development which is common at the meso and micro levels and which is facilitated by the simulation rules.

Stuttering Convergence. Since the number of atoms is finite, the stuttering action *Recruit* can only happen a finite number of times and will be disabled when $recruiters = placed$. This can be proved with the variant $W2 = \#placed - \#recruiters$.

The above system specification $System3$ classifies atoms into recruiters and followers without mentioning the mechanism facilitating the recruiting and the movement detail of the followers.

Next, the development will push the specification more closer to the micro level. At this abstraction level, we model that atoms can only communicate with their immediate neighbours and can only move a short distance for each step. The design intuition of specification *System*4 is to refine the recruiting process (*Recruit*) and the moving of the followers (*Move*) in *System*3. The 'gradients' mechanism will be introduced to express the abstract interactions between recruiters and their followers. A gradient is a nature number indicating the distance from a follower to its recruiter. An atom with gradient 0 means that it is a recruiter waiting a follower to fill a neighbouring target position. The recruiter publishes its gradient to its neighbours as a recruiting signal. An atom responding to the recruiting will become a follower by setting its gradient to a number one more than the gradient it received. The propagation terminates when all atoms have a gradient value. For a follower, its gradient indicates the distance between itself and its recruiter. Its neighbours' gradients tell whether it should move and where it should move to. For instance, in figure **??**, an atom with gradient 3 should move towards an atom with a less gradient and hence decrease its distance from its recruiter.

The class *System*4 extends *System*3 with an additional variable $grad : Position \pfun \mathbb{N}$ mapping a subset of atoms to gradient values. Initially, no atoms have gradient values. A seed atom which is already placed and needs to recruit generates gradient 0 and propagate it to its neighbours. A constant relation $nb : Position \leftrightarrow Position$ is introduced to indicate the neighbouring positions, *e.g.* let p, q be positions, $(p, q) \in nb$ means they are neighbouring positions. The propagation process continues until all atoms have a gradient value. In order to ensure that gradients can propagate to all atoms, *System*4 has an invariant that all atoms are connected, *i.e.*, $\forall\, p, q : atoms \bullet (p, q) \in nb^*$ where nb^* is the transition closure of relation nb.

The retrieve relation $R3$ between *System*3 and *System*4 relates the abstract recruiting mechanism with the concrete gradient mechanism, i.e., an atom in set *recruiters* in *System*3 which has vacant neighbouring positions is mapped to an atom with gradient 0 in *System*4.

*System*4

$[Position]$

$target : \mathbb{F}\, Position$
$nb : Position \leftrightarrow Position$

$\forall\, p, q : target \bullet (p, q) \in nb^*$

$atoms : \mathbb{F}\, Position$
$placed : \mathbb{F}\, Position$
$grad : Position \pfun \mathbb{N}$

$\# atoms = \# target$
$placed \subseteq atoms \cap target$
$\mathrm{dom}\, grad \subseteq atoms$

INIT

$\forall\, p, q : atoms \bullet (p, q) \in nb^*$
$placed \neq \varnothing$
$grad = \varnothing$

$\vdots$

$$
\begin{aligned}
R3 \mathrel{\widehat{=}}\ & target_3 = target_4 \wedge placed_3 = placed_4 \wedge nb_3 = nb_4 \wedge \\
& \quad (\forall\, p : Position \bullet p \in recruiters_3 \wedge (\exists\, q : target_3 \setminus placed_3 \bullet (p, q) \in nb) \Rightarrow \\
& \qquad (p \in \mathrm{dom}\, grad_4 \wedge grad_4(p) = 0))
\end{aligned}
$$

Following the retrieve relation, the abstract action *Recruit* can be simulated by a concrete change action *CreateGrad* in which a placed atom requiring neighbours creates a gradient by setting its gradient value to 0. The second to last line of the predicate in *CreateGrad* ensures that the action can only occur when the atom's gradient value is not already 0. This prevents an atom creating a gradient more than once (satisfying the abstract specification where *Recruit* can only occur once per atom).

CreateGrad

$$
\begin{array}{|l}
\Delta(grad) \\
p, q : Position \\
\hline
(p, q) \in nb \\
p \in placed \land p \notin \operatorname{dom} grad \\
q \in target \setminus placed \\
grad' = grad \oplus \{p \mapsto 0\}
\end{array}
$$

Propagate

$$
\begin{array}{|l}
\Delta(grad) \\
p, q : Position \\
\hline
p \in \operatorname{dom} grad \\
grad(p) = 0 \lor \exists\, r \in \operatorname{dom} grad \bullet \\
\qquad (p, r) \in nb \land grad(r) = grad(p) - 1 \\
q \in atoms \land (p, q) \in nb \\
q \in \operatorname{dom} grad \Rightarrow grad(q) > grad(p) + 1 \\
grad' = grad \oplus \{q \mapsto grad(p) + 1\}
\end{array}
$$

The propagation of the gradients is specified by stuttering action *Propagate* which do not change the abstract state according to the retrieve relation. *Propagate* sets the gradient value of an atom (at position q) to 1 greater than the gradient value of its neighbour (at position p). This creates a virtual gradient as was illustrated in Figure 2. If an atom already has a gradient value then the action will be enabled only when its gradient value will be decreased. This ensures that the gradient value represents the shortest distance to an atom seeking a neighbour. For any atom, we call its neighbour atom which has the gradient value 1 less than its own to be its *reference* atom.

The abstract action *Move* is simulated by a concrete change action *Join* which specifies the final movement of an available atom filling its target position. *Join* moves an atom (at position p) from one neighbouring position of a placed atom seeking a neighbour (at position q) to another neighbouring position of that atom which is part of the target (at position r). The moving atom becomes placed. The third to last line of the predicate ensures that the action does not lead to the mass of atoms becoming disconnected; the actual mechanism by which this would be achieved is left to the micro level.

Follow

$$
\begin{array}{|l}
\Delta(atoms, grad) \\
p, q, r, s : Position \\
\hline
Movable(p, q, r, s) \\
atoms' = (atoms \setminus \{p\}) \cup \{s\} \\
grad' = \{p\} \ndres grad
\end{array}
$$

Join

$$
\begin{array}{|l}
\Delta(atoms, placed) \\
p, q, r : Position \\
\hline
(p, q) \in nb \land (q, r) \in nb \land grad(q) = 0 \\
p \in atoms \setminus placed \land r \in target \setminus placed \\
\forall\, a, b : (atoms \setminus \{p\}) \cup \{r\} \bullet (a, b) \in nb^* \\
atoms' = (atoms \setminus \{p\}) \cup \{r\} \\
placed' = placed \cup \{r\}
\end{array}
$$

Follow moves an unplaced atom (at position p) from a neighbouring position of one atom with a smaller gradient value (at position q) to that of one of its neighbours (at position r) with an even smaller gradient value. The moving atom's gradient value is removed. The movement of atom at position p subjects to a condition $Movable(p, q, r, s)$

which ensures that

(1) p is a follower (not a placed atom).

(2) q is a neighbour of p and r is a neighbour of q. The gradients of p, q and r form a decreasing trail.

(3) s is a vacant position neighbouring position q and r.

(4) The movement of atom from position p to s does not lead to disconnection of the atoms.

The *Movable* condition can be formalised as follows.

$$
\begin{array}{rl}
\mathit{Movable}(p,q,r,s) \mathrel{\widehat{=}} & p \in \mathit{atoms} \setminus \mathit{placed} \wedge q, r \in \operatorname{dom} \mathit{grad} \wedge s \notin \mathit{atoms} \\
& \wedge\ (p,q) \in \mathit{nb} \wedge (q,r) \in \mathit{nb} \wedge (q,s) \in \mathit{nb} \wedge (r,s) \in \mathit{nb} \\
& \wedge\ \mathit{grad}(q) > \mathit{grad}(r) \wedge (p \notin \mathit{grad} \vee \mathit{grad}(p) > \mathit{grad}(q)) \\
& \wedge\ \forall\, a, b : (\mathit{atoms} \setminus \{p\}) \cup \{s\} \bullet (a,b) \in \mathit{nb}^*
\end{array}
$$

Finally, as mentioned in Section 2.2, we require a means to cancel gradients when they are no longer required. When a recruiter realises that all its neighbouring target position have been filled, it should tell the other unplaced followers that it is not a recruiter any more. This message is sent out by dissipating its gradient. The followers which received this message should also dissipate their current gradients and update them towards a new recruiter if there is any. This process is captured by action *DissipateGrad*. It occurs when there is an atom (recruiter) with gradient value 0 which has no vacant target positions in its neighbourhood, or an atom (follower) with a non-zero gradient value which loses its reference atom.

DissipateGrad

$$
\begin{array}{l}
\Delta(\mathit{grad}) \\
p : \mathit{Position} \\
\hline
p \in \operatorname{dom} \mathit{grad} \\
\mathit{grad}(p) = 0 \Rightarrow (\nexists q : \mathit{target} \setminus \mathit{placed} \bullet (p,q) \in \mathit{nb}) \\
\mathit{grad}(p) \neq 0 \Rightarrow (\nexists q : \operatorname{dom} \mathit{grad} \bullet (p,q) \in \mathit{nb} \wedge \mathit{grad}(q) = \mathit{grad}(p) - 1) \\
\mathit{grad}' = \{p\} \mathbin{\lhd\!\!\!-} \mathit{grad}
\end{array}
$$

To ensure the correctness of the design, we need to check the action refinement conditions and fix any problems detected. According to the retrieve relation $R3$, all occurrences of *Recruit* and *Move* are change actions in *System*3. All occurrences of *CreateGrad* and *Join* are change actions in *System*4, and all occurrences of *Propagate* and *Follow* are stuttering actions.

Initialisation. The condition is true as the fact that initial state schema of *System*4 is related to that of *System*3 under retrieve relation $R3$, i.e., $\forall \Sigma_4 \bullet \mathit{grad}_4 = \varnothing \Rightarrow \exists \Sigma_3 \bullet \mathit{recruiter}_3 = \varnothing \wedge R3$.

Action Simulation. The guard of *CreateGrad* (all lines but the last) implies the guard of *Recruit* according to $R3$. Also, the final line $\mathit{grad}_4' = \mathit{grad}_4 \oplus \{p \mapsto 0\}$ of *CreateGrad* corresponds to the line $\mathit{recruiters}_3' = \mathit{recruiters}_3 \cup \{p\}$ of *Recruit*. It is trivial to conclude that *CreateGrad* simulates *Recruit*. The proof for *Join* simulating *Move* follows similarly.

It is trivial to prove that stuttering actions *Propagate*, *Follow* and *DissipateGrad* refine $ID_{System3}$ since their post-states are related to the exact abstract states related to their pre-states under $R3$.

Termination. Checking this condition is to check that

$\forall\,\Sigma_3, \Sigma_4 \bullet R3 \Rightarrow (\neg\,\mathrm{pre}\,System4.Actions \Rightarrow \neg\,\mathrm{pre}\,System3.Actions)$.

As discussed before, $\neg\,\mathrm{pre}\,System3.Actions = (target_3 = placed_3)$. After trivial deduction, the conclusion we need to establish is that $\forall\,\Sigma_4 \bullet (\neg\,\mathrm{pre}\,System4.Actions) \Rightarrow (target_4 = placed_4)$. It can be done by check the guard of each single action in $System4$. Unfortunately we found a problem when we check $\neg\,\mathrm{pre}\,Follow \Rightarrow (target_4 = placed_4)$. Referring to the guard *Movable* of action *Follow*, this problem occurs when *Follow* is blocked not due to all atoms are in target positions but due to an atom at a position satisfying the constraints of r have no vacant neighbouring positions. This problem, as discussed by Støy [13], is fixed by assuming the target configuration is in the form of a 'scaffold' with vacant positions around any positions in which atoms are to be fixed, *i.e.*, $Scaffold \mathrel{\widehat=} \forall\,p : atoms \bullet \exists\,q : Position \setminus atoms \bullet (p, q) \in nb$. The scaffold assumption is then added as an invariant to the system specification. As in Støy's solution, it requires adding a similar invariant on *target*: $\forall\,p : target \bullet \exists\,q : Position \setminus target \bullet (p, q) \in nb$. With these two invariants, the guard of *Follow* is always *true* until $target_4 = placed_4$, which satisfies the termination rules. The modification does not affect the proofs of Initialisation or Action Simulation.

Stuttering Convergence. According to the retrieve relation, the stuttering actions in $System4$ include *Propagate*, *Follow* and *DissipateGrad*. To prove the stuttering convergence, we need to show that these actions cannot be executed infinitely. In formal, let $Stutt_1 \mathrel{\widehat=} Propagate \lor Follow \lor DissipateGrad$, we need to find a finite number N satisfying that:

$\forall\,\Sigma_4, \Sigma_4' \bullet \exists\,N : \mathbb{N} \bullet Stutt_1^N \Rightarrow \neg(\mathrm{pre}\ \ Stutt_1)'$

In the following discussion, we intend to find the upper bound of N by showing that the execution number of each stuttering action and their interleaving has a upper bound.[1]

Let m be the number of recruiters with gradient 0 in the pre-state. According to the specification, every execution of *Follow* decreases the following variant W_F for any movable follower.

$W_F \mathrel{\widehat=} \sum_{p \in P} min\{d(p, q) \mid grad(q) = 0\}$

where $P \mathrel{\widehat=} \{p \mid p \in atoms \setminus placed\}$, $d(p, q) \geq 0$ is the distance between position p and q, *i.e.*, $d(p, q) = i$ iff $(p, q) \in nb^i$.

The variant W_F has a lower bound 0 when all atoms are in *placed*. The execution of *Follow* leads the state to enable the change action *Join* which places a follower to a neighbouring target position. Therefore, for any atom $i : 1..n$, let d_i be the minimum distance to a recruiting atom in the pre-state. The upper bound for the number of executing *Follow* is $d_i - 1$ (since the last move is done by *Join*). For $n - m$ followers,

[1]It is enough to prove the existence of the upper bound. For simplicity, we assume the worst case in every step of execution. Hence the actual upper bound of N is smaller than the number we obtained.

the bound is $\sum\limits_{i=1}^{n-m}(d_i - 1)$. The execution of *Propagate* does not increase W_F since it does not change the position of any atom. The execution of *DissipateGrad* may increase W_F when it cancels a recruiter. But this increasing influence can only happen for at most m times (the number of recruiters and in the case where every time the canceled recruiter is the nearest one). The increased value of W_F is also bounded to $n-1$ (the maximum distance between two atoms). Hence the upper bound of executing *Follow* is $\overline{N_F} = (\sum\limits_{i=1}^{n-m}(d_i - 1)) + m(n-1)$.

The execution of *DissipateGrad* decreases a variant W_D which is a set containing

(1) any recruiter with gradient 0 which no longer has the requirement of recruiting (all its neighbouring target positions have been filled).

(2) any follower with gradient greater than 0 whose recruiter has dissipated its own gradient.

$$W_D \mathrel{\widehat{=}} \{p \mid grad(p) = 0 \land \nexists r : target \setminus placed \bullet (p, r) \in nb\}$$
$$\cup \{p \mid grad(p) \neq 0 \land \nexists q : \mathrm{dom}\, grad \bullet (p, q) \in nb \land grad(q) = grad(p) - 1\}$$

Note that when a recruiter dissipates its gradient, it implies that all its neighbouring target position have been filled, hence it will no longer be a recruiter again. Therefore, the maximum execution number of *DissipateGrad* for canceling a recruiter is n (assuming every atom has been a recruiter). After a recruiter being canceled, the followers having their gradient leading to it also dissipate their gradients through executing *DissipateGrad*. The maximum number of executing *DissipateGrad* for every follower of one recruiter is $n-1$ (assuming every other atom is a follower). In summary, the maximum number of executing *DissipateGrad* is $\overline{N_D} = n(n-1)$. Note that the execution of *Propagate* does not increase W_D since it does not increase the number of atoms having gradient 0 or not having a neighbour with a gradient 1 less than itself. And neither does *Follow*.

The execution of *Propagate* decrease a variant W_P standing for the set of atoms that does not have a "proper" gradient for the nearest recruiter.

$$W_P \mathrel{\widehat{=}} \{p \mid p \in atoms \land$$
$$(p \notin \mathrm{dom}\, grad \lor grad(p) > min\{d(p, q) \mid grad(q) = 0\}\}$$

The variant W_P has a lower bound $\varnothing$ when every atom has a gradient leading to the nearest recruiter. For every atom that is not a recruiter, the number of executing *Propogate* to update its gradient is at most m. Hence, without considering the other stuttering actions, *Propagate* can be executed for at most $m(n-m)$ times. Note that W_P can be increased by *Follow* which remove the moving follower from *grad*. But every execution of *Follow* will enable the execution of *Propagate* at most once so that the increasing influence has an upper bound $\overline{N_F}$. The execution of *DissipateGrad* may also increase W_P whenever it removes the gradient of an atom. For every atom which dissipate its gradient, *Propagate* will be executed at most once to reset its gradient leading to another recruiter. The gradient set by such *Propagate* will not be removed until this recruiter is cancelled. Hence, the additional execution number is $\overline{N_D}$. In summary, the upper bound of executing *Propagate* is $\overline{N_P} = m(n-m) + \overline{N_F} + \overline{N_D}$. The upper bound of N is $\overline{N} = \overline{N_F} + \overline{N_D} + \overline{N_P}$.

4.3. Micro level

In our case study we have exactly one type of agent, an atom, whose state can be specified as follows. We will return to the actions of an atom in Section **??**. *Position* and *nb* play the same role as in the meso-level specifications. $\mathbb{N}_\infty$ is a type to denote the gradient value of an atom; it is either a natural number or the symbol ∞ denoting that no numerical gradient value has been established. A relation $<$ is defined to relate gradient values: $n_1 < n_2$ iff $n_2 = \infty$ and n_1 is a number, or both n_1 and n_2 are numbers and n_1 is less than n_2. This relation is used both in establishing gradient values, and in determining whether an atom can move, i.e., whether the connectedness of atoms will be maintained by the move.

The type **message** of atoms includes gradient values (*grad*), requests for a position to move to (*request*), responses to such requests (*response*), notifications that an atom is moving (*moving*), notifications that a gradient has dissipated (*dissipate*), specifications of the target and and the receiving atom's position within it (*target*), a notification that an atom has successfully joined the target (*joined*), and an acknowledgement of this notification (*ack*). Some of the messages (*grad*, *response*, *target*, *joined* and *ack*) carry values.

Following Stoy [13], we assume that the atoms that are placed in a target position have a copy of the target (*target*), and know their position within it (*pos*). All other atoms will have an empty target, i.e., $target = \varnothing$, in which case the value of *pos* is meaningless. As in the meso-level specification, the target is in the form of a 'scaffold'. Placed atoms also know which of their neighbouring target positions are filled (*filled*).

To allow atoms to be recruited, each atom also has a gradient value (*grad*) to indicate its distance from the recruiting atom. Each atom also has a set *next* to record its neighbouring atoms which have a smaller gradient value and a set *dependents* to record its neighbouring atoms which have a larger gradient value. The former indicates the direction the atom needs to move to join the target, and the latter is to ensure it does not isolate atoms (as in Figure 3) by moving. A boolean state variable *joined* indicates that an atom has notified its neighbours that it has joined the target.

Initially, *filled* is empty, there is no gradient value and no atoms in *next* and *dependents*, and *joined* is true only for atoms which have a copy of the target.

Atom

$[Position]$

$$nb : Position \leftrightarrow Position$$

$$\mathbb{N}_\infty ::= num\langle\langle \mathbb{N} \rangle\rangle \mid \infty$$

$$\_ < \_ : \mathbb{N}_\infty \leftrightarrow \mathbb{N}_\infty$$

$$\forall n_1, n_2 : \mathbb{N}_\infty \bullet n_1 < n_2 \Leftrightarrow (n_2 = \infty \land n_1 \neq \infty \lor n_1 \neq \infty \land n_2 \neq \infty \land n_1 < n_2)$$

$$\textbf{message} ::= grad\langle\langle \mathbb{N}_\infty \rangle\rangle \mid request \mid response\langle\langle Atom \rangle\rangle \mid moving \mid dissipate \mid target\langle\langle \mathbb{F}\, Position \times Position \rangle\rangle \mid joined\langle\langle Position \rangle\rangle \mid ack\langle\langle Position \rangle\rangle$$

$pos : Position$
$target, filled : \mathbb{F}\, Position$
$grad : \mathbb{N}_\infty$
$next, dependents : \mathbb{F}\, Atom$
$joined : \mathbb{B}$

$target \neq \varnothing \Rightarrow pos \in target$
$\forall p : target \bullet \exists q : Position \setminus target \bullet (p, q) \in nb$

INIT

$filled = \varnothing \land grad = \infty \land next = dependents = \varnothing \land (joined \Leftrightarrow target \neq \varnothing)$

⋮

The use of the agent specification *Atom* as a type (in the declaration of *next* and *dependents*) is borrowed from Object-Z where classes are similarly used as types. As in Object-Z, instances of agent specifications in MAZE are *references* to the agents. Such a reference is independent of the agent's state and does not change as the agent performs actions. In an implementation of our case study, we would not expect an atom to store such references. Their use is simply an abstraction for another means of referring to particular neighbouring atoms, such as the port through which they communicate.

In our example, we model a topology where neighbouring atoms can send messages to each other. To do so, as well as the topology (*atoms*) we include an injective function (*position*) mapping agents to their positions in the system state. An invariant is included to relate the function *position* to the topology. When an atom is placed (i.e., its *target* variable is non-empty) its position corresponds to the position (*pos*) it stores as part of its state. Initially, all atoms are connected and, following Stoy and Nagpal [13, 14], there is exactly one atom in the target.

*System*5

$$
\begin{array}{l}
atoms : \mathbb{T}\, Atom \\
position : Atom \rightarrowtail\!\!\!\!\rightarrow Atom.Position \\
\hline
\mathrm{dom}\, position = \mathrm{dom}\, atoms \\
\forall\, a, b : \mathrm{dom}\, atoms \bullet (a, b) \in atoms \Leftrightarrow (position(a), position(b)) \in Atom.nb \\
\forall\, a : \mathrm{dom}\, atoms \bullet a.target \neq \varnothing \Rightarrow position(a) = a.pos
\end{array}
$$

$$
\begin{array}{l}
\textsc{Init} \\
\forall\, a, b : atoms \bullet (position(a), position(b)) \in Atom.nb \\
\exists_1\, a : \mathrm{dom}\, atoms \bullet a.joined
\end{array}
$$

$\vdots$

The variable *position* is required to relate the local constant *nb* and the local variable *pos* of *Atom* to the topology described by the variable *atoms*. Future work will look at extending MAZE actions to specify movement in 2D or 3D space directly, without the need for such local variables.

To develop the micro-level specification from the meso-level specification, we use the following retrieve relation between the states of *System*4 and *System*5.

$$
\begin{array}{l}
R4 \mathrel{\widehat{=}} (\forall\, a : \mathrm{dom}\, atoms_5 \bullet a.target \neq \varnothing \Rightarrow a.target = target_4) \wedge Atom.nb = nb_4 \wedge \\
\quad atoms_4 = \{a : \mathrm{dom}\, atoms_5 \bullet position_5(a)\} \wedge \\
\quad placed_4 = \{a : \mathrm{dom}\, atoms_5 \mid a.joined \bullet position_5(a)\} \wedge \\
\quad grad_4 = \{a : \mathrm{dom}\, atoms_5 \mid a.grad \neq \infty \bullet (position_5(a), a.grad)\}
\end{array}
$$

Since $R4$ relates each of the abstract variables to the concrete state by R, they are observable and hence each abstract operation (since it changes at least one of these abstract variables) is a change operation.

The Initialisation condition holds since initially in both *System*4 and *System*5 the mass of atoms are connected, no atom has a gradient value, and there is at least one atom placed in a target position (exactly one in *System*5) .

The micro-level agent actions must simulate the actions of *System*4. Hence we verify the action simulation conditions during the design of actions in this section. Below we focus on agent actions for creating and propagating gradients, and for moving. Actions for dissipating gradients and joining the target can be defined in a similar fashion (see Appendix).

4.3.1. Gradients

The meso-level action *CreateGrad* can be simulated by the application of the following agent action *NewGrad* which allows an agent which has joined the target and has unfilled neighbouring positions and a non-zero gradient value to set its gradient value to zero and broadcast a *grad* message. The progress predicate ensures that its *joined* message has been received by each of the atom's neighbours, and the atom has received their acknowledgments.

Similarly, the meso-level action *Propagate* can be simulated by the application of the following agent action *SetGrad* which models an agent, on receiving a *grad* message with a gradient value g less than its own, setting its own value to g and broadcasting its own *grad* message with value $g + 1$.

NewGrad

$\Delta(grad)$

$\mathbf{progress}(\{joined(pos)\},$
$\quad \{p : Position \bullet ack(p)\})$
$joined$
$filled \neq \{p : target \mid (pos, p) \in nb\}$
$grad \neq 0 \wedge grad' = 0$
$\mathbf{send}(grad(1))$

SetGrad

$\Delta(grad, next)$
$g : \mathbb{N}_\infty$
$a : Atom$

$\mathbf{receive}(grad(g), a)$
$g < grad$
$grad' = g \wedge next' = next \cup \{a\}$
$\mathbf{send}(grad(g + 1))$

The proof that the implicit *System*5 action $[\!]\ a : \mathrm{dom}\, atoms \bullet a.NewGrad$ simulates *CreateGrad* is straightforward: *CreateGrad* is enabled when a placed atom with a non-zero gradient value has an empty target position in its neighbourhood while *a.NewGrad* has a stronger guard which also requires the receipt of all acknowledgements of the atom's *joined* message as specified by the **progress** statement. Both actions result in the atom's gradient value being set to 0.

The **progress** statement in *NewGrad* is required so that the atom is aware of which of its neighbouring positions are filled before creating the gradient. More details about *joined* messages and acknowledgements are provided in the Appendix.

The proof that $[\!]\ a : \mathrm{dom}\, atoms \bullet a.SetGrad$ simulates *Propagate* is not possible without an additional assumption. *Propagate* is enabled only when an atom p has a neighbouring atom q with a gradient value at least 2 less than its own (since p sends its gradient value plus 1 and this must be less than that of q). $[\!]\ a : \mathrm{dom}\, atoms \bullet a.SetGrad$ is enabled whenever a message $(grad(g), b, a)$, for some $b : \mathrm{dom}\, atoms$, is in the global buffer with g less than $a.grad$. Since *grad* messages are only sent by *NewGrad* and *SetGrad* and these actions send values 1 greater than the *grad* value of their sender, *SetGrad* only occurs when the sender's gradient value is at least 2 less than it own. Hence, to prove the refinement, we need to ensure that receiving such a *grad* message implies b is a neighbour of a, i.e.,

$$\forall\, a, b : \mathrm{dom}\ atoms, g : \mathbb{N} \bullet (grad(g), b, a) \sqsubseteq \mathbf{buffer} \Rightarrow b \in atoms(a)$$

It requires (i) that a $grad(g)$ message can be sent from b to a only when b is a neighbour of a with gradient value g, and (ii) that after sending such a message b cannot stop being a neighbour of a before the message is received. The former is ensured in *System*5 by the restriction on the agent topology that atoms can only send messages to their neighbours (see the state schema of *System*5 in Section **??**). The latter requires any action which changes the neighbourhood of an agent which may have sent a *grad* message to have a **progress** statement stating that all such messages have been received. This requirement is considered in Section 4.3.2.

First, however, we specify two stuttering actions associated with gradient propagation. *AddDependent* models an atom receiving a gradient value greater than its own. This happens when the sending atom is further from the source than the receiving atom, in which case the sender may be relying on the receiver to be connected to the gradient source and hence becomes a dependent of the receiver. *IgnoreGrad* models an atom receiving a gradient value equal to its own. In this case, the sending atom is not relying on the receiving atom to be connected to the gradient source and the message can be ignored.

```
AddDependent
  $\Delta(dependent)$
  $g : \mathbb{N}_\infty$
  $a : Atom$
  ----
  $\mathbf{receive}(grad(g), a)$
  $g > grad$
  $dependent' = dependent \cup \{a\}$
```

```
IgnoreGrad
  $g : \mathbb{N}_\infty$
  $a : Atom$
  ----
  $\mathbf{receive}(grad(g), a)$
  $grad = g$
```

4.3.2. Moving

The meso-level action *Follow* requires an atom (e.g., a) to move from being a neighbour of one atom b to being the neighbour of another c which is closer to the source of a gradient. More specifically, b is a neighbour with a lower gradient value. In the micro-level specification, such a neighbour b can be chosen from the a's local variable *next*, and its neighbour c from the set $b.next$. As $b.next$ is not a part of the local state of the atom wanting to move, we introduce an action *Request* which enables an atom to request the reference to an atom in the *next* set of one of its neighbours. We also introduce an action *Respond* modelling the response to such a request.

```
Request
  $b : Atom$
  ----
  $\mathbf{progress}(\{grad(grad), request\},$
                  $\{grad(grad + 1), response\})$
  $target = \varnothing \land dependent = \varnothing$
  $b \in next$
  $\mathbf{send}(request, b)$
```

```
Respond
  $a, c : Atom$
  ----
  $\mathbf{receive}(request, a)$
  $c \in next$
  $\mathbf{send}(response(c), a)$
```

The *Request* action is enabled for atoms which have a non-empty *target* and have no dependents. The former will be true for atoms which have not yet obtained a target position. The guard $b \in next$ ensures *next* is not empty and hence the atom has a neighbour with a smaller gradient (this is an invariant which can be proven once all concrete actions are defined).

The progress condition ensures two things. Firstly, that all responses to the atom's *grad* message (sent as part of *SetGrad*) have been received. This will be the case when there are no $grad(grad)$ messages from the atom in the buffer, nor any $grad(grad + 1)$ messages to the atom. The latter ensures the atom is aware of any dependents, and is also sufficient to satisfy the second requirement for proving *SetGrad* simulates *Propagate*.

The second thing ensured by the progress condition is that a response has been received for any previous *request* message from the agent. This ensures an atom sends only one *request* message at a time. As with waiting for responses to its *grad* message, this could be implemented by the atom waiting for a worst-case response time dependent on the communication medium. The progress condition enables us to abstract from such an implementation-dependent timing constraint.

When a response is received, the requesting atom can move, setting its gradient value to ∞ and its *next* set to include just the atom to whose neighbourhood it moves. It also broadcasts to its neighbours that it is moving. This is captured by agent action *Move*. On receiving a notification that an atom is moving, the receiving atom can remove it from its dependent set if necessary. This is captured by action *RemoveDependent*.

$$\begin{array}{|l}
\hline \mathit{Move} \\
\Delta(\mathit{grad}, \mathit{next}) \\
b, c : \mathit{Atom} \\
\hline \mathbf{receive}(\mathit{response}(c), b) \\
\mathit{grad}' = \infty \land \mathit{next}' = \{b\} \\
\mathbf{send}(\mathit{moving}) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline \mathit{RemoveDependent} \\
\Delta(\mathit{dependent}) \\
a : \mathit{Atom} \\
\hline \mathbf{receive}(\mathit{moving}, a) \\
\mathit{dependent}' = \mathit{dependent} \setminus \{a\} \\
\hline
\end{array}$$

When the agent action *Move* is performed, a synchronised system action $\mathit{Move} < a : \mathrm{dom}\, \mathit{atoms} \bullet a.\mathit{Move} >$ changes the topology: an agent a moves to a position p neighbouring atoms b and c. The actual values of b and c are constrained by those of $a.\mathit{Move}$ whose common-named declarations are equated with those of *Move*.

$$\begin{array}{|l}
\hline \mathit{Move} < a : \mathrm{dom}\, \mathit{atoms} \bullet a.\mathit{Move} > \\
\Delta(\mathit{position}) \\
b, c : \mathrm{dom}\, \mathit{atoms} \\
p : \mathit{Position} \\
\hline p \notin \mathrm{ran}\, \mathit{position} \land (\mathit{position}(b), p) \in \mathit{Atom}.\mathit{nb} \land (\mathit{position}(c), p) \in \mathit{Atom}.\mathit{nb} \\
\mathit{position}' = \mathit{position} \oplus \{a \mapsto p\} \\
\hline
\end{array}$$

The intended simulation of the meso-level action *Follow* is system action *Move*. The proof of the simulation depends on the proposition that the guard condition of *Move* is stronger than the guard of *Follow*. In fact, the guards of *Request* and *Respond* along with the invariant on *next* ensures that the agent action *Move* is enabled only when there exists a neighbour b of atom a with a smaller gradient value than a's, and a neighbour c of b with a smaller gradient value than b's. The additional guard of the system action *Move* ensures that there is a vacant position in the neighbourhood of c. The guard of the abstract action *Follow* also requires that the atoms remains connected. With the mechanism of *dependent*, an atom can only move if it does not have a neighbour with a greater gradient. Hence, the guard of *Move* implies the connection of the atoms and the simulation holds.

Figure 4 shows the gradient setting phase and the movement of the agent with the highest gradient towards its recruiter. The right hand side is the changes of the system configuration. The arrows between atoms show the atom's *next* set and the suffix of the atom indicates its gradient. The dashed boxes represent empty target positions.

4.3.3. Checking Termination

Analysis of the guards of the abstract and concrete actions reveals a case where the concrete specification deadlocks and the abstract does not. The deadlock is caused when

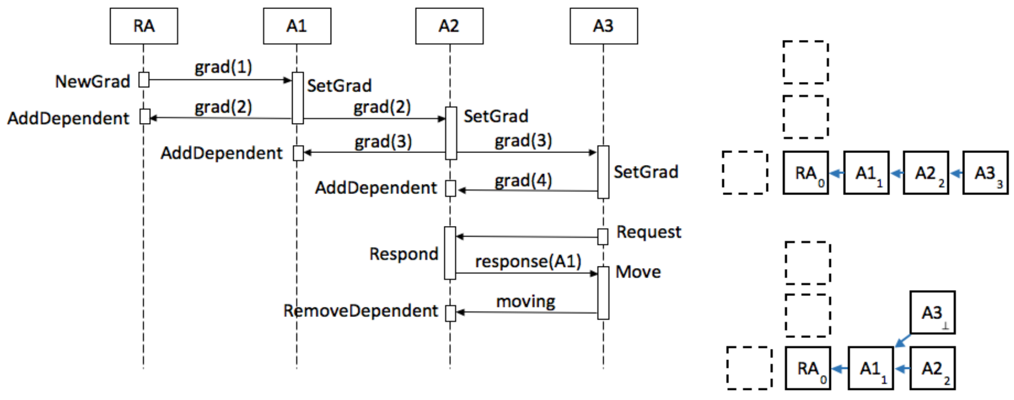

Figure 4: Sequence diagram for gradient setting

an atom is ready to move, but is unable to move as there is no vacant position adjacent to the referred atom. To fix this, we add an additional agent action *CancelMove* and its associated system action as follows.

CancelMove

$b, c : Atom$

$\mathbf{receive}(response(c), b)$

CancelMove $<a : atoms \bullet a.CancelMove>$

$b, c : \mathrm{dom}\ atoms$

$\nexists p : Position \bullet$
$\quad (position(c), p) \in nb \land p \notin \mathrm{ran}\ position$

4.3.4. Checking Stuttering Convergence

The stuttering actions of *System*5 are agent actions *AddDependent*, *IgnoreGrad*, *Request*, *Respond*, *RemoveDependent* and *CancelMove*. To prove the refinement, we need to check that these actions cannot be executed an infinite number of times. In fact, the guards of each of these actions apart from *Request* depend on corresponding messages in the global buffer. The progress condition in *Request* ensures that an atom sends only one request at a time, and must wait for a response before sending another. Infinite stuttering is avoided, therefore, if we can prove that there can be only a finite number of messages in the buffer during the execution of the system.

If we check all agent actions which can send a message, we can find that these actions can only generate a finite number of messages. For example, the number of messages that action *NewGrad* generates equals the number of its current neighbours which is finite. Furthermore, it can be only executed once. These *grad* messages sent to the buffer would be received by neighbouring atoms and trigger their *SetGrad* action. *SetGrad* can only be executed when it has received a *grad* message having a gradient value less than its own. Since it has a finite number of neighbours n, the gradient would be set at maximum n times until a movement of a neighbour atom, which is a change action, occurs.

5. Related Work

The formal specification of agents and MAS has been explored by many researchers. A formal agent framework using Z is proposed by d'Inverno and Luck in [23]. The framework captures the autonomous local behaviour of an agent and inter-agent interactions. Hilaire et al. [24] propose a prototyping approach for specifying multi-agent systems. It employs a composition of Object-Z and statecharts for formalising the interaction patterns based on agent roles. Unlike our approach, these formal specifications of MAS do not provide a development approach for achieving system-level properties. Aştefănoaei and de Boer [25] define a notion of refinement for BDI agents (agents which are driven by their beliefs, desires and intentions) [2, 26]. However, abstract and concrete specifications are not in the same notation. Therefore, their approach allows only a single refinement step from an abstract to a concrete representation of an agent, not the incremental development of an agent.

The Event-B formalism also advocates top-down development of software systems using refinement techniques [20][27]. The simulation rules for action refinement in this paper are inspired by the rules in Event-B. Since publication of our first paper on MAZE [10], a similar approach based on Event-B has been proposed [28]. This paper describes a formal development approach for achieving desired system-level properties by cooperative behaviour of 'foraging ants'. The development begins from an abstract macro-level specification and, for each refinement step, the specification provides finer and more detailed mechanisms of the agents' local behaviour and introduces cooperation. Unlike our work, however, this paper does not use the concepts of the macro, meso and micro levels from agent-based software engineering to provide a generic development approach. We believe this helps the designer to separate the macro-level concerns of system functionality, the meso-level concerns of agent interaction and the micro-level concerns of local agent behaviour by focussing on one of these sets of concerns at a time. Additionally, our syntactic conventions at the micro-level allow us to readily abstract from low-level mechanisms dealing with asynchronous communication and timing constraints. Such low-level mechanism are central to the operation of asynchronous MAS [21, 22] and is not clear how such they would be handled in the approach of [28].

6. Conclusion

This paper has presented MAZE, an extension of Object-Z [9] for the specification and development of multi-agent systems, and its application to a swarm robotic self-assembly algorithm. MAZE supports the development of multi-agent systems at three distinct levels of abstraction proposed by researchers in the agent-oriented software engineering community. Macro-level specifications capture global system functionality, meso-level specifications additionally include agent interactions and interaction paradigms, and micro-level specifications focus on the local functionality of individual agents. To ensure the three levels are consistent, MAZE employs a notion of action refinement based on that of action systems [11], with practical proof rules inspired by those of Event-B [20]. To facilitate specification at the micro-level, MAZE also includes a number of syntactic constructs. These enable the specifier to abstract from the low-level details of asynchronous communication and timing mechanisms common in such systems [21].

Future work on MAZE will focus on tool support via translation from the MAZE notation to Event-B allowing the use of the Rodin toolkit [29], as well as experiments on other multi-agent systems exhibiting emerging properties. Introducing time and probabilistic notions to MAZE is also an interesting direction to follow.

Acknowledgements This work was supported by Australian Research Council (ARC) Discovery Grant DP110101211 and National Science Foundation of China NSFC 61402176.

[1] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd Edition, Wiley, 2009.

[2] A. Rao, M. Georgeff, BDI agents: From theory to practice, in: International Conference on Multi-Agent Systems (ICMAS-95), 1995, pp. 312–319.

[3] F. Dressler, Self-organization in sensor and actor networks, Wiley, 2007.

[4] M. Gerla, T. Kwon, G. Pei, On demand routing in large ad hoc wireless networks with passive clustering, in: Wireless Communications and Networking Conference (WCNC 2000), Vol. **1**, 2000, pp. 100–105.

[5] F. Zambonelli, A. Omicini, Challenges and research directions in agent-oriented software engineering, Autonomous Agents and Multi-Agent Systems 9 (3) (2004) 253–283.

[6] G. Smith, J. Sanders, Formal development of self-organising systems, in: J. González Nieto, W. Reif, G. Wang, J. Indulska (Eds.), International Conference on Autonomic and Trusted Computing (ATC 2009), Vol. 5586 of LNCS, Springer-Verlag, 2009, pp. 90–104.

[7] S. Eder, G. Smith, An approach to formal verification of free-flight separation, in: Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010), IEEE Computer Society Press, 2010, pp. 166–171.

[8] G. Smith, K. Winter, Incremental development of multi-agent systems in Object-Z, in: SEW-35, IEEE Computer Society Press, 2012.

[9] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 2000.

[10] G. Smith, Q. Li, MAZE: An extension of Object-Z for multi-agent systems, in: Y. A. Ameur, K.-D. Schewe (Eds.), 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014), Vol. 8477 of LNCS, Springer-Verlag, 2014, pp. 72–85.

[11] R. Back, J. von Wright, Trace refinement of action systems, in: B. Jonsson, J. Parrow (Eds.), 5th International Conference on Concurrency Theory (CONCUR '94), Vol. 836 of LNCS, Springer-Verlag, 1994, pp. 367–384.

[12] J. Spivey, The Z Notation: a reference manual, 2nd Edition, Prentice-Hall, 1992.

[13] K. Støy, Using cellular automata and gradients to control self-reconfiguration, Robotics and Autonomous Systems 54 (2006) 135–141.

[14] K. Støy, R. Nagpal, Self-reconfiguration using directed growth, in: International Symposium on Distributed Autonomous Robotic Systems (DARS 6), Springer-Verlag, 2007, pp. 3–12.

[15] C. Jones, M. Matarić, From local to global behavior in intelligent self-asssembly, in: IEEE International Conference on Robotics and Automation (ICRA'03), IEEE, 2003, pp. 721–726.

[16] E. Klavins, R. Ghrist, D. Lipsky, A grammatical approach to self-organizing robotic systems, IEEE Transactions on Automatic Control 51 (6) (2006) 949–962.

[17] A. Christensen, R. O'Grady, M. Dorigo, SWARMORPH-script: A language for arbitrary morphology generation in self-assembling robots, Swarm Intelligence 2 (2-4) (2008) 143–165.

[18] W. Lui, A. Winfield, Autonomous morphogenesis in self-assembling robots using IR-based sensing and local communications, in: M. Dorigo, M. Birattari, G. D. Caro, R. Doursat, A. Engelbrecht, D. Floreano, L. Gambardella, R. Groß, E. Şahin, H. Sayama, T. Stützle (Eds.), International Conference on Swarm Intelligence (ANTS 2010), Vol. 6234 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 107–118.

[19] J. Derrick, E. Boiten, Refinement in Z and Object-Z, Foundations and Advanced Applications, 2nd Edition, Springer-Verlag, 2014.

[20] J.-R. Abrial, Modelling in Event-B, Cambridge University Press, 2010.

[21] C. Chou, I. Cidon, I. Gopal, S. Zaks, Synchronizing asynchronous bounded delay networks, IEEE Trans. Communications 38 (2) (1990) 144–147.

[22] Q. Li, G. Smith, Using bounded fairness to specify and verify ordered asynchronous multi-agent systems, in: International Conference on Engineering of Complex Computer Systems (ICECCS 2013), IEEE Computer Society Press, 2013, pp. 111–120.

[23] M. d'Inverno, M. Luck, Development and application of a formal agent framework, in: First IEEE International Conference on Formal Engineering Methods., IEEE computer society, 1997.

[24] V. Hilaire, A. Koukam, P. Gruer, J.-P. Müller, Formal specification and prototyping of multi-agent systems, in: A. Omicini, R. Tolksdorf, F. Zambonelli (Eds.), Engineering Societies in the Agents World, Vol. 1972 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 114–127.

[25] L. Aştefănoaei, F. de Boer, The refinement of multi-agent systems, in: M. Dastani, K. Hindriks, J.-J. Meyer (Eds.), Specification and Verification of Multi-agent Systems, Springer-Verlag, 2010, Ch. 2, pp. 35–65.

[26] M. Wooldridge, N. R. Jennings, Intelligent agents: Theory and practice, Knowledge Engineering Review **10** (1995) 115–152.

[27] W. Su, J.-R. Abrial, R. Huang, H. Zhu, From requirements to development: Methodology and example, in: 13th International Conference on Formal Engineering Methods, (ICFEM 2011), Springer, 2011, pp. 437–455.

[28] L. Laibinis, E. Troubitsyna, Z. Graja, F. Migeon, A. Hadj Kacem, Formal modelling and verification of cooperative ant behaviour in Event-B, in: D. Giannakopoulou, G. Salaün (Eds.), Software Engineering and Formal Methods, Vol. 8702 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 363–377.

[29] J.-R. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, International Journal on Software Tools for Technology Transfer 12 (6) (2010) 447–466.

Appendix

This appendix gives the definitions of the rest of the actions in the micro-level specification of the swarm robotic self-assembly system in Section 4. It is only for the review and to comply with page limits will not be included in the published version.

A concrete agent action *Join* is defined to simulate the abstract action *Join*. It models an atom moving to the position recruited by a placed atom. Its definition is similar with the action *Move* except that it is enabled when the atom receives a *target* message instead of a *response* message. A synchronised system action $Join < [\!] \; a : \mathrm{dom}\, atoms \bullet a.Join >$ is also needed to change the topology of the system when the moving succeeds.

Join

$\Delta(grad, next, target, pos)$
$t : \mathbb{F}\, Position$
$p : Position$

$\mathbf{receive}(target(t, p), b)$
$grad' = \infty \wedge next' = \varnothing$
$target' = t \wedge pos' = p$
$\mathbf{send}(moving)$

$Join < [\!] \; a : \mathrm{dom}\, atoms \bullet a.Join >$

$\Delta(position)$
$p : Position$

$p = position(a) \Rightarrow position' = position$
$p \notin \mathrm{ran}\, position \Rightarrow position' = position \oplus \{a \mapsto p\}$

The abstract action *Join* is intended to be simulated by a system action *Join* together with additional stuttering agent actions *SendPosition*, *Fill*, *UpdateFilled*, *IgnoreJoined* and *ReceiveAck*.

SendPosition models a placed atom with vacant neighbouring positions sending a *target* message with one of these positions in response to a *request* message.

$$\begin{array}{|l}
\textit{SendPosition} \\
\hline
b : \textit{Atom} \\
p : \textit{Position} \\
\hline
\mathbf{receive}(\textit{request}, b) \\
\textit{joined} \\
p \in \{p : \textit{target} \mid (\textit{pos}, p) \in \textit{nb}\} \setminus \textit{filled} \\
\mathbf{send}(\textit{target}(\textit{target}, p), b) \\
\hline
\end{array}$$

Fill models an atom which joined the target position sending out a *joined* message to let its new neighbours update their *filled* set. The variable *joined* is used to ensure that the *joined* messages are sent by the atom to its new neighbours after moving to its target position. *UpdateFilled* models the updating when an atom receives a *joined* message.

$$\begin{array}{|l}
\textit{Fill} \\
\hline
\Delta(\textit{joined}) \\
\hline
\textit{target} \neq \varnothing \wedge \neg\, \textit{joined} \\
\textit{joined}' \\
\mathbf{send}(\textit{joined}(\textit{pos})) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{UpdateFilled} \\
\hline
\Delta(\textit{filled}) \\
b : \textit{Atom} \\
p : \textit{Position} \\
\hline
\mathbf{receive}(\textit{joined}(p), b) \\
\textit{joined} \\
\textit{filled}' = \textit{filled} \cup \{p\} \\
\mathbf{send}(\textit{ack}(\textit{pos}), b) \\
\hline
\end{array}$$

IgnoreJoined models an atom which is not placed in a target position ignoring a *joined* message. *ReceiveAck* models an atom receiving an *ack* message from one of its neighbours and updating its *filled* variable with the neighbours position. Note that *ack* messages are only sent in response to a *joined* message and hence will only be received by atoms which have joined the target.

$$\begin{array}{|l}
\textit{IgnoreJoined} \\
\hline
b : \textit{Atom} \\
p : \textit{Position} \\
\hline
\mathbf{receive}(\textit{joined}(p), b) \\
\neg \textit{joined} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{ReceiveAck} \\
\hline
\Delta(\textit{filled}) \\
b : \textit{Atom} \\
p : \textit{Position} \\
\hline
\mathbf{receive}(\textit{ack}(p), b) \\
\textit{filled}' = \textit{filled} \cup \{p\} \\
\hline
\end{array}$$

RemoveGrad models an atom with $\textit{grad} = 0$ and all its neighbouring target positions filled setting its gradient value to ∞ and broadcasting a *dissipate* message. *Dissipate* models an atom with $a \in \textit{next}$ receiving a *dissipate* message from a, setting its gradient value to ∞ and broadcasting a *dissipate* message.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{RemoveGrad}} \\ \hline
\Delta(\mathit{grad}) \\ \hline
\mathit{grad} = 0 \\
\mathit{filled} = \{p : \mathit{target} \mid (\mathit{pos}, p) \in \mathit{nb}\} \\
\mathit{grad}' = \infty \\
\mathbf{send}(\mathit{dissipate}) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Dissipate}} \\ \hline
\Delta(\mathit{grad}, \mathit{next}) \\
a : \mathit{Atom} \\ \hline
\mathbf{receive}(\mathit{dissipate}, a) \\
a \notin \mathit{next} \Rightarrow \mathit{grad}' = \mathit{grad} \land \mathit{next}' = \mathit{next} \\
a \in \mathit{next} \Rightarrow \mathit{grad}' = \infty \land \mathit{next}' = \mathit{next} \setminus \{a\} \\
\qquad \land\ \mathbf{send}(\mathit{dissipate}) \\ \hline
\end{array}$$

Both of the two change actions $[]\ a : \mathit{atoms} \bullet a.\mathit{RemoveGrad}$ and $[]\ a : \mathit{atoms} \bullet a.\mathit{Dissipate}$ refine the abstract action *Dissipate*.

Similar with the *Move* case, when an atom receives a *target* message but is unable to move to the target position (note that a guard requiring that position to be free would be part of a system action associated with the atom action *Join*) the atom needs to ignore the position and send another request. This is modelled by an additional stuttering agent/system synchronisation action *CancelJoin*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CancelJoin}} \\ \hline
b : \mathit{Atom} \\
t : \mathbb{F}\,\mathit{Position} \\
p : \mathit{Position} \\ \hline
\mathbf{receive}(\mathit{target}(t, p), b) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CancelJoin} < a : \mathit{atoms} \bullet a.\mathit{CancelJoin} >} \\ \hline
p : \mathit{Position} \\ \hline
p \in \operatorname{ran} \mathit{position} \\
\mathit{position}(a) \neq p \\ \hline
\end{array}$$