



COMP 395 Internet Computing Final Year Project :
Simulation, Visualization and Experimental analysis for
Population Protocols and Network Constructor
in 2-Dimensional Case

Author:	Haoxuan Wang
Studnet ID:	201219597
First Supervisor:	Dr. Othon Michail
Second Supervisor:	Dr. Vitaliy Kurlin

May 9, 2018

Declaration of academic integrity

I confirm that I have read and understood the University's Academic Integrity Policy.

I confirm that I have acted honestly, ethically and professionally in conduct leading to assessment for the programme of study.

I confirm that I have not copied material from another source nor committed plagiarism nor fabricated, falsified or embellished data when completing the attached piece of work.

I confirm that I have not copied material from another source, nor colluded with any other student in the preparation and production of this work.

Date: May 9, 2018

Department of Computer Science
University of Liverpool

Abstract

The abstract[1] [2] [3]

Contents

1	Introduction	1
1.1	Aims and Objectives	1
1.2	The challenges in the project	1
1.2.1	Heterogeneous for different types of Models	1
1.2.2	Heterogeneous for different types of Protocols	1
1.2.3	Human factor: Lacking Experience for model visualization	1
1.2.4	Brief introduction to the programme	1
1.2.5	Evaluation of the project	2
2	Background	2
2.1	Existing solution	2
2.2	Brief Introduction to Population Protocols [1, 4]	2
2.3	From Population Protocols to Network Constructor [2]	3
2.4	Grid Terminating Network Constructor [3]	4
2.5	Vector transformation: handle coordinate rotations in 2-dimension	5
2.5.1	2D centred coordinate rotation, with origin point as the centre	6
2.5.2	2D centred coordinate rotation, with any points as the centre	6
2.5.3	Affine transformation form	7
3	Data Required	7
4	Design	7
4.1	Anticipated components of the system	7
4.2	Data structure, algorithms and pseudo-code for key method	8
4.2.1	Graph and adjacency list	8
4.2.2	Quad-direction linked structure	8
4.2.3	Terminating grid network constructor: Check acceptable configuration & Handle rotation	10
4.3	Interaction (Sequence) Design	14
4.4	UI Interface Design	16
4.5	Object used in the system	17
5	Realisation	18
5.1	From Start of Realisation to 3rd March, 2018: implement first two models	18
5.2	From 3rd March to 26th March, 2018: Further progression	20
5.3	From 27th March to 8th April, 2018: Implement terminating grid network con- structor	22
5.4	From 9th April to 15th April, 2018: User Interface implementation and experiment	24
6	Learning Points	25
6.1	Academic skills	25
6.2	Technical (Development) skills	26
6.3	Soft skills	26
	Bibliography	27
A	Brief Derivation for equation (1)	28
B	Brief Derivation for equation (3)	29
C	List of Implemented Protocol	29
D	Open source libraries used in the simulator	29

1 Introduction

1.1 Aims and Objectives

The project aimed to study general population protocols [1] and its two derived model, network constructor [2] and terminating grid network constructor [3] (specifically for grid network construction). It also attempted to experimentally simulate, visualise and compare these protocols via building the simulator and visualizer.

1.2 The challenges in the project

1.2.1 Heterogeneous for different types of Models

The theoretical models involved in three main different models initially originated in population protocols. These three models share inherently common points but there are also some conceptual differences in between them. For instance, the network constructor [2] and terminating grid network constructor [3] involve state of connections in between two nodes while the original population protocol does not. The node of terminating grid network constructor has its complexity structurally compared with the other two types of model.

1.2.2 Heterogeneous for different types of Protocols

The protocols discussed in the related papers [1, 2, 3] involve many different protocols. The protocols are totally different on many characteristics, such as their different computational ability, different ending in either or termination, computation target. These differences between protocol to protocol may lead the simulator and visualizer hard to develop and test.

1.2.3 Human factor: Lacking Experience for model visualization

Prior to this project, the author has totally no experiences on model simulation and also no knowledge on what kind of related library will be involved. Learning may take more time than its expected.

1.2.4 Brief introduction to the programme

The final programme contains an UI with an fix-sized area to illustrate the interaction process of a particular protocol and shows the states of elements* in the population. In addition, it contains a information panel containing some related information with regard to the population itself, including:

- Number of nodes
- Number of nodes distinguished in different status
- Number of selections for pairs of nodes[†] that scheduler had took
- Number of effective interactions the population executed

*"Elements" refers nodes in general population protocol, but also includes edge if the protocol involves edge states.

[†]may also include pair of ports for terminating grid network constructor

Additionally, it provides a set of parameters' settings regarding the initial configuration of a protocol and a population to be simulated, which includes:

- The number of nodes included in the simulation
- The initial state for each node[‡]
- The protocol type (and also different sets of transition rules for the protocol)
- Option on whether to use fast-forward simulation method for initially n -times selection, and the value of n if the option is enabled. A fast-forward simulation executed in the model but does not present that process in the viewer so it normally faster than the case that does not enable this option.

1.2.5 Evolution of the project

In general, the simulator successfully implemented a series of different protocols for the three model mentioned above.

UI The UI functions of simulator is verified through a large number of different population simulations. This ensures the UI functions work as they expected in design stage. These experiments on theoretical model may also be asserted the correctness of model through the output configuration of these simulations.

Model The model partition of the simulator developed through Testing driven development method. According to the specification, the testing suits had been written before the written of any model code . The model code has to pass all testing suits after their implementation.

2 Background

2.1 Existing solution

There is an existing simulator called NETCS [5] for network constructor, which is the model covered in this paper [2], the simulator was written in Java, provided an web user interface and assisted related research.

This work will explore the possibility to simulate a further variance of network constructor, grid network constructor, as well as attempting to provide a uniform simulation to different models and protocols, which is easily extensible.

2.2 Brief Introduction to Population Protocols [1, 4]

Population protocols are theoretical models for distributed computation [1, 4]. The model contains a collection of indistinguishable agents. They (i.e. the agents) carry out computation tasks through directly pair-wised interactions. The interaction pattern of agents is unpredictable from perspective of agents themselves and is controlled through an adversary scheduler with fairness constraints [1, 4].

[‡]The state of edge for network constructors (i.e. network constructor and terminating grid network constructor) should be always "0" (i.e. inactivated) at initial, so it is omitted here.

Formal Definition [1] A protocol can be formally defined as:

- Q , a finite set of passible states for an agent,
- Σ , a finite set of input alphabet,
- Y , a finite set of output range,
- $\iota = \Sigma \rightarrow Q$, is an input map from Σ to Q , hence $\iota(\sigma)$ represents the initial state whose input is $\sigma \in \Sigma$
- $\omega = Q \rightarrow Y$, is an output map from Q to Y , and
- $\delta \subseteq Q^4$, a transition relation describes how pairs of agents can interact.

A *configuration* for a population is a vector of all the agents' states. Because agents with a same state are indistinguishable with each other, each configuration could also an unordered multiset of states. It can be represented as C [1, 4].

At any point of the discrete time, the interaction is unpredictable from the perspective of the agents in the population and the population itself. The interactions at any time are decided by a adversary scheduler necessarily enforced with *fairness* condition [1, 4]. The fairness condition means that the scheduler cannot avoid a possible step forever. Formally, it means if a infinitely often configuration C and $C \rightarrow C'$, the C' must also appear infinitely of in the execution [1].

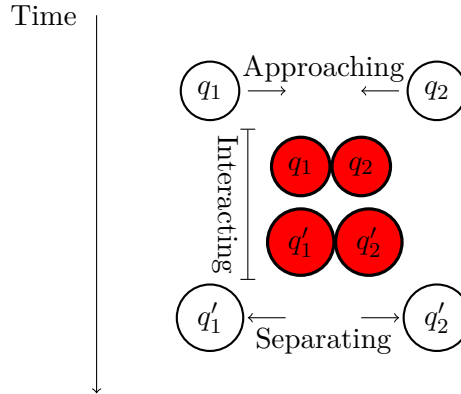


Figure 1: A typical (simple) interaction in population protocol

The current implementation of this work assumes a more strict scheduler called random scheduler, which resulting uniform random interactions (i.e. at each step, it presents equally possibility for every pair of agents to interact.) Essentially, a random scheduler is a "fair" scheduler, but a "fair" scheduler is not necessarily a random scheduler. The assumption actually increases the power of the model because it allows a leader agent detects the absence of agents in a particular state after a long enough waiting [1]. Note this is only for simplifying the current implementation and possibly extends to other fair schedulers that is not a random scheduler in the future.

2.3 From Population Protocols to Network Constructor [2]

A variance of population protocol is called "network constructor" [2]. It involves the extra elements do not existing in general population protocol called "edge", which is the connection

in between any two agents (or "processes") [2]. In general, the connection is similar to the agents, on the characteristic that both of them have a finite number of states. For instance, if a connection have $k + 1$ states, state 0 may represent the connection does not exist and for state $i \in \{1, 2, 3, \dots, k\}$ shows the strength for a particular connection.

In the following paragraph and the currently implemented simulator, we solely consider the simplest case containing only *on* and *off* two states for edges [2]. A connection is said to be in *on* or *active* state, if at any discrete time, the connection exists in between two particular nodes; otherwise if the connection is stated as in *off* or *inactive* state [2]. At initial, all edge are in *off* state, which means that there is no connection in a given population at discrete time 0 [2]. The state for an edge may or may not change through an interaction in between the two nodes it connected or disconnected to.

The network constructor also follows an adversary scheduler with "fairness" constraint. This is exactly same as the paragraphs mentioned in the previous section 2.2.

Formal Definition [2] Formally, a network constructor can be defined as:

- Q , a finite set of passible states for an agent,
- Q_{out} , a finite set of output range,
- $q_0 \in Q$, is initial state of node,
- $\delta : Q \times Q \times \{0, 1\} \rightarrow Q \times Q \times \{0, 1\}$, a transition function, where the $\{0, 1\}$ is the states for edges with initial value 0.

The main target of this model concentrates on network construction rather than specific function computation, which is the task for general population protocol [2]. Notice that a transition can be either *effective* or *ineffective*. Define $\delta(a, b, c) = (a', b', c')$ as a transition function (Here, a, b are "node-state" whereas c is state for edge.) as it is in the formal definition, $\delta_1(a, b, c) = a'$, $\delta_2(a, b, c) = b'$ (representing two nodes states' change) and $\delta_3(a, b, c) = c'$ (representing edge state change), the transition $(a, b, c) \rightarrow (a', b', c')$ is called effective if at least one $x \in \{a, b, c\} \neq x'$; otherwise it is called ineffective [2].

Name the set of nodes (or "distributed processes" under this context) V_I and the set of pairs of nodes as E_I . A configuration C is a mapping $V_I \cup E_I \rightarrow Q \cup \{0, 1\}$ determines the states of nodes and edges in the population at any discrete time. The output of configuration C is defined as the graph $G(C) = (V, E)$ where $V = \{u \in V_I : C(u) \in Q_{out}\}$ and $E = \{uv : u, v \in V, u \neq v, \text{ and } C(uv) = 1\}$ [2].

2.4 Grid Terminating Network Constructor [3]

Another paper [3] presents a different automata but very similar to network constructor. Each node in this model has a fixed number of ports with it. In the 2-dimensional case, it will have 4 distinguished ports p_y, p_x, p_{-y} and p_{-x} , may simply donated as u, r, d and l , respectively [3]. The port, which neighbour with each other, are also perpendicular to each other, forming local axes. Hence, $u \perp r, r \perp d, d \perp l$, and $l \perp u$ [3]. The coordinates are only for local purposes and do not necessarily represent the actual orientation of a node. A connection (or edge) can only be built through pairs of ports, which is different from the previous model (network constructor).

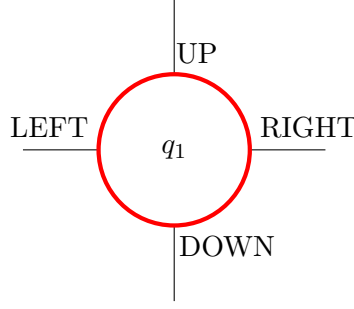


Figure 2: Structure of a node in Grid Terminating Network Constructor model

Formal Definition [3] Formally, a terminating grid network constructor in 2-D can be defined as:

- Q , a finite set of passible states for an agent,
- Q_{out} , a finite set of output range,
- $q_0 \in Q$, is initial state of node
- $\delta : (Q \times P) \times (Q \times P) \times \{0, 1\} \rightarrow Q \times Q \times \{0, 1\}$, a transition function, where the $\{0, 1\}$ is the states for edges with initial value 0.
- When required, there will be also a special initial leader-state $L_0 \in Q$ defined.

A transition can be either *effective* or *ineffective*. Define $\delta((a, p_1), (b, p_2), c) = (a', b', c')$ as a transition function as it is in the formal definition, it is called effective if at least one $x \in \{a, b, c\} \neq x'$; otherwise it is called ineffective [3].

Every configuration C forms a set of shapes $G[A(C)]$, where a shape means that the network induced by active edges of C [3]. Given the geometrical restrictions (i.e. the connection are made at unit distance and are perpendicular whenever they correspond to consecutive ports of a node), not all possible $A(C)$ are valid [3]. $A(C)$ is valid if any connected component defined by it is a subnetwork of 2D grid network with unit distances. A valid $A(C)$ at time $t - 1$ also restricts the possible valid $A(C)$ at time $t + k$, where $k \in Integer$ and $k \geq 0$ [3].

The paper [3] also defines a set of shapes $G_{out}(C) = \{V_s, E_s\}$ as output of a configuration where $V_s = \{u \in V : C_v(u) \in Q_{out}\}$ and $E_s = A(C) \cap \{(v_1, p_1) : v_1 \neq v_2 \in V_s; p_1, p_2 \in P\}$ [3]. Less formally, the output shapes of a configuration contains all nodes in output states and the active edges in between them. This model focuses on terminating protocols so it assumes $Q_{out} \subseteq Q_{halt} \subseteq Q$. All rules containing $q_{halt} \in Q_{halt}$ is ineffective [3].

2.5 Vector transformation: handle coordinate rotations in 2-dimension

As mentioned in the previous section 2.4, Grid Terminating Network Constructor enforces perpendicular ports and geometrical restrictions. This means that handling rotation in some cases becomes necessary. This section will cover some basic but related vector mathematics. The detailed algorithms will be discussed later on, in the section (4).

2.5.1 2D centred coordinate rotation, with origin point as the centre

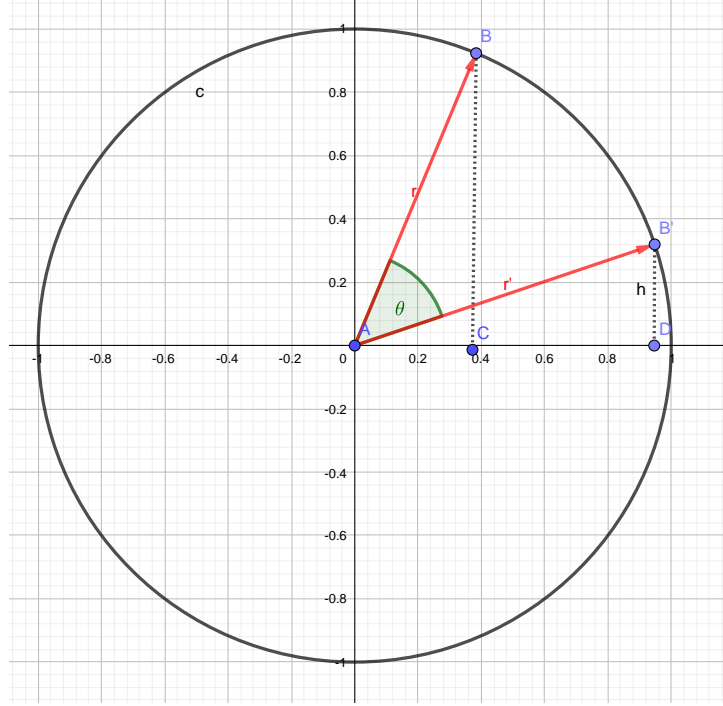


Figure 3: Vector \vec{r} and \vec{r}' in unit circle

The figure 3 has two same-length vectors, \vec{r} (or \vec{AB}) and \vec{r}' (or $\vec{AB'}$) in its Cartesian coordinate and their destination point located at the same unit circle. The point A is the origin point with coordinate (0,0). Suppose it has already known for the coordinate of B is (x, y) , and the θ angle. The question is that what the coordinate of B' (representing in (x', y')) is, after the first vector \vec{r} rotates the given angle θ and becomes \vec{r}' . Note that, the "positive" rotation direction in current context and the implementation of the simulator is defined as "clock-wised".

To solve this, the following equation can be concluded:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad (1)$$

A brief derivation can be found in appendix partition below and it also can be found in [6] with a more precise and general proof.

2.5.2 2D centred coordinate rotation, with any points as the centre

Given the equation (1), it can be easily deduced that the rotation equation for any centred points c with coordinate (c_x, c_y) can be achieved through panning c to origin point with coordinate (0,0), carry out the rotation and panning back to its original coordinate (c_x, c_y) . Suppose the target point that is required to be rotated is called p (with coordinate (x, y)), it then has :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} c_x \\ c_y \end{bmatrix} \right) + \begin{bmatrix} c_x \\ c_y \end{bmatrix} \quad (2)$$

p' (with coordinate (x', y')) will be point after p rotating centred on c .

2.5.3 Affine transformation form

After a series of transformation, the equation 2 becomes the following form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} c_x(1 - \cos \theta) - c_y \sin \theta \\ c_y(1 - \cos \theta) + c_x \sin \theta \end{bmatrix} \quad (3)$$

Let $\mathbf{M} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$ and $\mathbf{B} = \begin{bmatrix} c_x(1 - \cos \theta) - c_y \sin \theta \\ c_y(1 - \cos \theta) + c_x \sin \theta \end{bmatrix}$, then $\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{M} \begin{bmatrix} x \\ y \end{bmatrix} + \mathbf{B}$, where it forms an *affine transformation* [7]. \mathbf{M} and \mathbf{B} can be calculated directly after knowing the rotation centre (c_x, c_y) and the rotation angle θ , and therefore the calculated \mathbf{M} and \mathbf{B} can be applied for multiple target coordinates (those coordinates required to be rotated). This simplifies the calculation for a large batch of centred rotation with multiple target coordinates.

3 Data Required

There is also no real human data, real non-human data or human participants involved in the project. For information used in papers, they are properly cited and referenced. The simulator itself, developed based on some open sourced libraries, these libraries are open to public and freely available in the public domain. The list of open sources library and their licence would be listed as in appendix.

4 Design

4.1 Anticipated components of the system

The main partition of the system was a software that enables to simulate and visualise each interaction before convergence (or terminate). The project would also attempt to explore one or more protocol to check their internal status after finishing the software.

The simulator would allow users to choose from a set of well-defined protocol and specify the parameters for these protocol (e.g. number of agents participating). The simulator would allow users to easily extend their own protocol though writing structured few lines of code, and the extension should be as easy as possible.

The system expected to have a ability to simulate some variations (which at least includes the network constructor [2] and terminating grid network constructor in 2D [3]) of the original protocol according the protocol and dynamically visualise how process (nodes) transfer for each interaction.

The interface provided would be a desktop application running on Java Virtual Machine Platform. A web interface was desirable but would not be guaranteed to deliver.

Modification against original design The original design mentioned it could include "a configuration parser to enable user define a protocol from configuration files" finally replaced by a set of well-defined set of protocol and a programmable interface. This is because the original design requires a design and implementation of Domain Specific Language (DSL), and a parser

of this kind of "new language", which is a expected large work load and might be infeaisible to finish all of them during the project period. This might be done in the following development plan after finishing the project itself.

4.2 Data structure, algorithms and pseudo-code for key method

4.2.1 Graph and adjacency list

For some specific kinds of model, specifically ,the network constructor, a data structure to maintain a undirected graph G formed from current configuration C is required. In other words, it requires to maintain a group of status of *active* edges. It aims to construct "shapes" and there are some status for the edge uv in between any nodes (processors) u and v in the population.

To achieve this, an adjacency list would be used to record the connections associated with each nodes. Initially, it would be an map with all nodes in the Population as keys and an empty list associated as the value. If the connection in between u and v builds up, the node u will be added to the list which the node v corresponds to and vice versa (i.e. the v will be added to the list which the key u corresponds to). When an *active* connection, say a connection between u' and v' "cancelled" (i.e. running from *active* state to *inactive* state). The node reference v' in the list that u' as the key would be deleted and so the node reference u' in the list that v' as the key.

Modification against original design The original design used an adjacency matrix rather than a adjacency list to represent the undirected graph possibly appeared in some model to be simulated. The modification was made based on the consideration on space cost. A adjacency matrix requires a fixed $S(N^2)$ space allocated overhead for N nodes in the population while the proposed adjacency list implementation would consume $S(E)$ where E is the number of "active" edges, which at most has a value of $\frac{N*(N-1)}{2}$.

For every model defined, the initial configuration starts with no *active* edge so it would be a zero matrix if the adjacency matrix is used in the program and maintain a sparse state for most of time during simulation process. A adjacency list might be a better choice since the list would save spaces required than adjacency matrix.

4.2.2 Quad-direction linked structure

In 2-dimensional case, the agents in grid terminating grid network constructor model would have exactly 4 ports associated with them and any edges have to be connected through ports. As mentioned before, the port themselves are perpendicular with each other. To implement port concept and connection attached to ports, the system would use a quad-direction linked structure to construct this. Each nodes would have 4 null-able references to other nodes, one of which represents a port associated to this node. A connection (or edge) e between node n and node n' is treaded as "active" on a port p of n if the reference that p associated pointing to an another non-null node n' and so similar for the symmetric reference in n' . In contrast, the connection (or "edge") on a port is deemed as "inactive" if the port reference pointing to null.

Listing 1: Pseudo-code demonstration of quad-direction linked structure

```
class LocallyCoordinatedModelNode(...) {
    var up: LocallyCoordinatedModelNode? = null
```

```

var down: LocallyCoordinatedModelNode? = null
var left: LocallyCoordinatedModelNode? = null
var right: LocallyCoordinatedModelNode? = null
...
}

```

Relative node coordinate and rotation For convenience of implementation, the relative node coordinate would be defined here. Given a node n , with its coordinate (x_n, y_n) . The nodes that connected to its port p_x, p_{-x}, p_y, p_{-y} will have coordinate $(x_n + 1, y_n), (x_n - 1, y_n), (x_n, y_n + 1), (x_n, y_n - 1)$ respectively. Note that the coordinate is localized and from different node, they will have a different perspective of coordinate. It is also required to define a parameter φ_i representing the rotated degree from normal Cartesian coordinates. Here, the "positive" direction can be defined as clockwise.

Relative port degree In terminating grid network construction model, the 4 ports associated with a node are called port p_y, p_x, p_{-y} and p_{-x} , may simply denoted as u (UP), r (RIGHT), d (DOWN) and l (LEFT), respectively. They are perpendicular to each other. To model the perpendicularity of node numerically, it is proposed to notate a degree for each port. Specifically, u (UP) is with a degree value of 0° , r (RIGHT) is with a degree value of 90° , d (DOWN) is with a degree value of 180° and l is with a degree value of 270° .

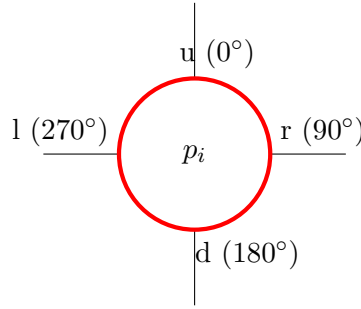


Figure 4: Defined degree for each port

The difference θ of degree between two ports p_1, p_2 can be defined as

$$\theta = ||\theta_1 - \theta_2|| = ||Angle(p_1) - Angle(p_2)||$$

where $Angle$ is a mapping from a port $p_i \in \{p_y, p_x, p_{-y}, p_{-x}\}$ to its degree $\theta_i \in \{0^\circ, 90^\circ, 180^\circ, 270^\circ\}$. Suppose there are two nodes a, b and two ports where p_a is one port of a and p_b is one port of b , then the "relative difference" Θ could be:

$$\Theta_{p_{ab}} \equiv ||\theta_{p_a} - \varphi_a - (\theta_{p_b} - \varphi_b)|| \bmod 360 \quad (4)$$

which is the port difference after involving the rotation of nodes.

Relative docking port Also for convenience of implementation, here two pairs of "relative docking port" would be defined. A pair of relative docking port is the pair of ports of which relative difference is 180° . Certainly, this limits only two possible pairs of relative docking port:

port $\{p_x, p_{-x}\}$ as the first pair of docking port, and port $\{p_y, p_{-y}\}$ as the second pair of docking port. Suppose there are two nodes shown as exactly in diagram 4, that is, two nodes with 0° rotations and if assuming there is no rotation allowed in the configuration, the only legal connection will be connections in between the two pairs of relative docking port, where the two ports belonging to different nodes and the two nodes can be interchangeable. For instance, suppose the port of the first node is selected as p_x , then the only legal choice for the port of the second node is p_{-x} , which is the relative docking port of p_x .

Breadth-First Search (BFS) Algorithm [8] on Quad-direction linked structure Breadth-First Search (BFS) algorithm is a famous and broadly-used graph traversing algorithm and is also foundation for some other algorithm [8]. Here, it would not be repeated for the details of BFS algorithm but there is a modification when applying the algorithm the Quad-direction linked structure. Because there are four nodes reference (edge) attached to one node, it treat all the 4 references as possible source of "son nodes" and if any of them are not null and also had not been visited, they will be marked and be added to the tail of a queue for further traversal.

Consider a *clique* or a connected (sub -)graph in a configuration for the terminating grid network constructor model, the BFS can be used to aggregate this kind of clique from any node inside that clique as a set. This set of nodes S can be used as an input for a function applying on a batch of nodes, for instance, rotating all nodes according to a centred node.

4.2.3 Terminating grid network constructor: Check acceptable configuration & Handle rotation

Problem: Detecting legal configuration As mentioned in background section, not all possible configurations are acceptable, or say, belong to $A(C)$ given the geometrical restriction[3]. Hence, this will requires some inspections to ensure that only the acceptable configurations form after a node interacts with each other during simulation process. There are two ways to finish this kind of inspections:

1. List all possible interactions at each discrete step, filter out all illegal ones to get $A(C)$, and then select one configuration from $A(C)$.
2. Select one possible interaction from all possible interactions, reject if the interaction forms a configuration $c_w \notin A(C)$ and repeat the process until a acceptable configuration is found. The entire process would be happened in one discrete step.

The second approach is more feasible since it supposed to cost a impracticable long time to enumerate all possible elements in $A(C)$ at a particular discrete step. Hence, it would be discussed for the cases that should be rejected in an interaction.

Avoid self-connection Recall the definition of transition function of grid network constructor, two nodes and two edges associated them respectively are selected. This not excludes the case that one same identity select twice, which leading to a unsound connection on the selected node itself. This should be avoid through prohibiting the second nodes being selected if it is the same entity with the first one.

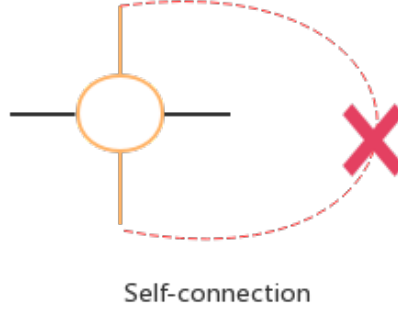


Figure 5: Illegal configuration: Self-connection

Avoid unit-distance violation A connection in terminating grid network constructor has to follow the unit distance restriction.

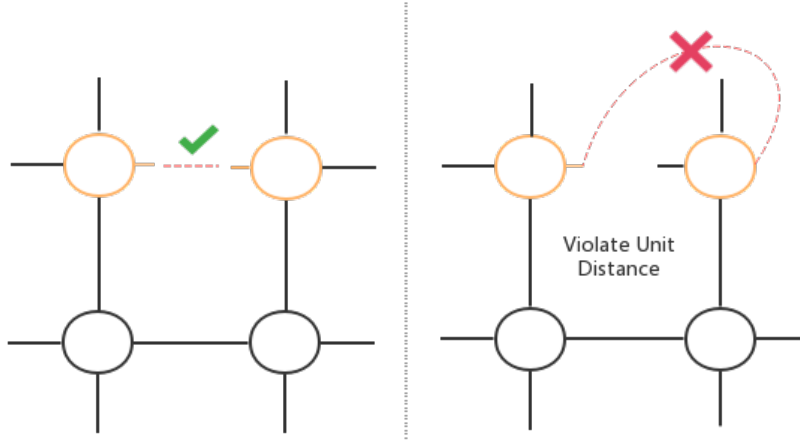


Figure 6: Illegal configuration: Unit distance violation

As illustrated in diagram 6, the left-hand side configuration is acceptable but the right-hand side configuration is not because it violate the unit distance. After the connection in the right-hand side being active, the configuration will not be sub-graph of 2-D grid network with unit distance restrictions.

It can be noticed that unit distance violation always happens when two interacting nodes are belonging to one same clique since a rotation will helps if the two nodes are in different clique. The condition that two nodes are in the same clique simplifies the judgement process. First, it

requires to check the whether selected two port are docking port according to equation 4 (so this ensures after activating connection the direction of two ports will conform the grid network restriction). If the two ports selected are a pair of docking port, the initializer node then will mark itself with coordinate (0,0) and populate the coordinate using BFS algorithm. Finally, it will check the Euclidean distance of the two interacting nodes using propagated coordinates, which is asserted to be 1 otherwise it means that unit-distance restriction will be broken after interaction.

Algorithm 1 Algorithm for detecting unit distance violation

```

1: procedure DETECT
2:    $((nodeA, portA), (nodeA, portA)) \leftarrow scheduler.select()$ 
3:    $isViolation \leftarrow false$ 
4:   if  $nodeA.belongToGroup \neq nodeB.belongToGroup$  then return  $isViolation$ 
5:   if  $isDockingPort(portA, portB)$  then
6:      $nodeA.coordinate \leftarrow (0, 0)$ 
7:      $connectedSet \leftarrow bfsForPopulatingCoordinate(nodeA)$ 
8:      $distance \leftarrow euclideanDistance(nodeA \in connectedSet, nodeB \in connectedSet)$ 
9:     if  $distance \neq 1$  then
10:       $isViolation \leftarrow true$ 
11:   else
12:      $isViolation \leftarrow true$ 
   return  $isViolation$ 

```

Avoid overlapping Overlapping is another situation that persuades a configuration become an acceptable configuration. This normally happens when two nodes that belongs to different cliques interacting with each other. This is not allowed for some cases when more than one nodes appears in the same position.

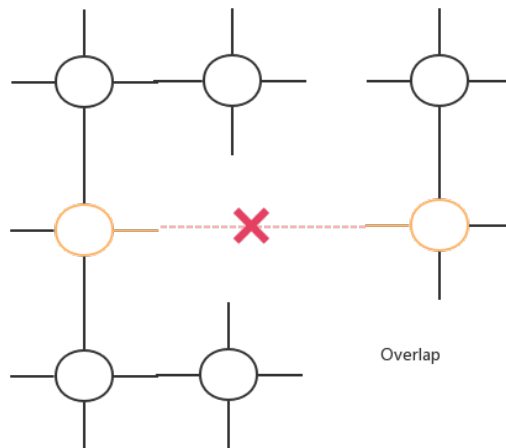


Figure 7: Illegal configuration: Unit distance violation

The figure 7 presents a configuration for highlighted nodes and edges. This kind of interaction

should be rejected. To resolve this, it can be assumed that the interaction had already finished and see whether there is at least one pair of nodes owning a same coordinate. If it does, the interaction should be rejected because it causes an overlap otherwise it should accept the connection.

Note that after populating the coordinate, the coordinates of nodes in clique that the receiver belonging to is based on the receiver nodes' perspective. Hence, these coordinates of nodes require to do a centralized rotation with receiver as the center to confirm to the coordinates from initializer's perspective. The rotation degree is given by

$$180 - ||Angle(port_{receiver}) - Angle(port_{initializer})||$$

which is the "port difference without node rotation involves", because the internal shape of two cliques is no related to their rotation degree).

Some cases might be more complicated. The two cliques may overlap with each other but their axisymmetric one will not. An example could be:

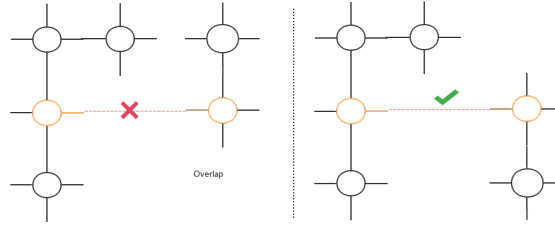


Figure 8: Axisymmetric shape eliminate overlap

The two cliques will be overlapped with each other after interaction in the left-hand of the figure 8 but if one of them flip about the "x-axis" of another clique the overlap will be eliminated. For interaction happens with $p_x(\text{RIGHT})$ and $p_{-x}(\text{LEFT})$ of initializer node, it may need to check the clique that the receiver node belongs to and its x-axisymmetric shape. For interaction happens with $p_y(\text{UP})$ and $p_{-y}(\text{DOWN})$ of initializer node, it may need to check the clique that the receiver node belongs to and its y-axisymmetric shape.

Modification against original design This section is the solution towards the legal configuration checking problem in original design document. The original version proposed such a problem without answers. For now, the solution have been finished.

Algorithm 2 Algorithm for detecting overlapping

```
1: procedure DETECT
2:    $((nodeA, portA), (nodeA, portA)) \leftarrow scheduler.select()$ 
3:    $isOverlap \leftarrow false$ 
4:   if  $nodeA.belongToGroup = nodeB.belongToGroup$  then return  $isOverlap$ 
5:   else
6:      $nodeA.coordinate \leftarrow (0, 0)$ 
7:      $connectedSetA \leftarrow bfsForPopulatingCoordinate(nodeA)$ 
8:      $nodeB.coordinate \leftarrow getCoordinateReceiver(nodeA, portA)$ 
9:      $connectedSetB \leftarrow bfsForPopulatingCoordinate(nodeB)$ 
10:     $connectedSetB \leftarrow bfsForRotation(connectedSetB, 180 - ||portB.angle - portA.angle||)$ 
11:     $symmetricSetB \leftarrow symmetric(connectedSetB, portA)$ 
12:    if  $(connectedSetA \cap connectedSetB \neq \emptyset) \wedge (connectedSetA \cap symmetricSetB \neq \emptyset)$  then
13:       $isOverlap \leftarrow true$ 
14:    else
15:      if  $(connectedSetA \cap connectedSetB \neq \emptyset)$  then
16:         $changeToSymmtricStructure(nodeB, portA)$ 
17:    return  $isOverlap$ 
```

4.3 Interaction (Sequence) Design

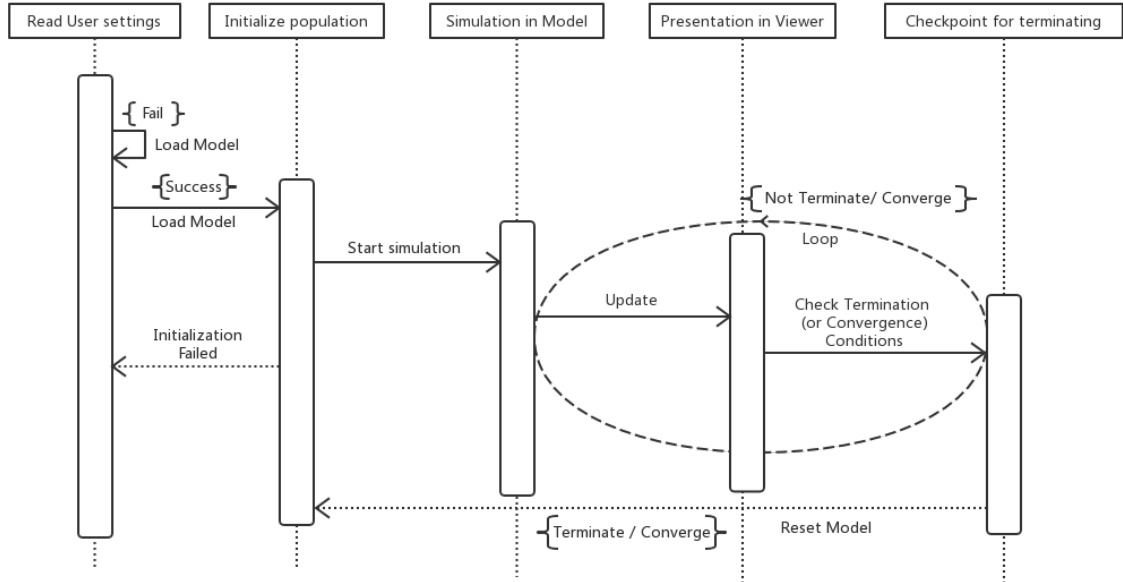


Figure 9: Interaction Design Diagram

The figure 9 illustrates the entire interaction process of the simulator. Initially, the user select one protocol from a set of well-defined protocols and then select a set of parameters required such

as how many nodes to be included in the simulation, the initial status distribution for nodes and so forth. Then the user click the "Apply settings button" to apply the model settings. before applying the model, the program itself would check whether all parameters are well-defined, i.e. locating in the value range they should be. If they are, the software would load the model otherwise it will reject this particular request from user and allow the users to attempt their (possibly) another setting.

After the program successfully loaded a model, the user can press the "Start simulation" button to start a simulation. A simulation contains multiple rounds. For each round, the model executes interaction simulation under the control of scheduler specified in one of parameters mentioned before and then the model would update the states of elements in viewer to show the what happened in the model itself. Following that, the model would check whether it reaches a configuration that should stop the simulation. A stop can be caused by user's temporarily interruption (pause) or the fact that the number of consistent accumulation of inefficient interactions overwhelms a pre-configured parameters defined in the model (which indicates a terminating or convergence with a high possibility). If the model does not detect a stop, it will continue the "interact - update - checking stop" process as a loop. Once it detect a stop, the software will return to state of waiting for starting of a simulation process (while the model would keep its internal states and user can build an another new model under this situation).

Modification against original design The current design removed a step of sequence in original sequence design called "Initialize scheduler", which located after "Initialize population" step. the "Initialize scheduler" process had been integrated into "Initialize population" because the scheduler partition does not necessarily separated from the population and can be treated as a partition of a "population" in concept.

4.4 UI Interface Design

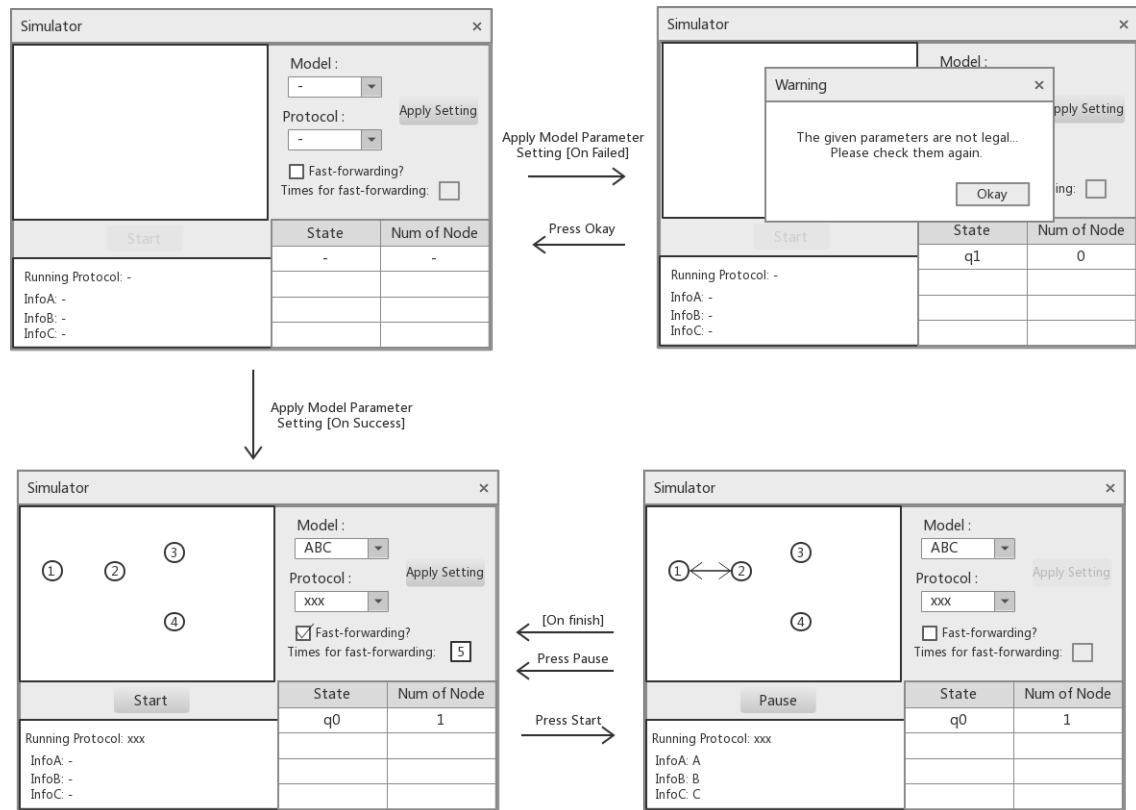


Figure 10: Interaction Design Diagram

Modification against original design The UI was redesigned entirely. This is because the original design built on the assumption that the users load their settings for some population from domain specification language (DSL) codes residing in some files. The DSL is infeasible to be finished due to the limited time of implementation. Hence, the design would allow user to set their population through the UI containing a set of well-defined protocols and would allow the users to write their own extensional protocol through few lines of codes. As the functionality changes, the UI removed the functionality entrance for loading files from disk and added some options to allow user setting the parameters of a protocol.

4.5 Object used in the system

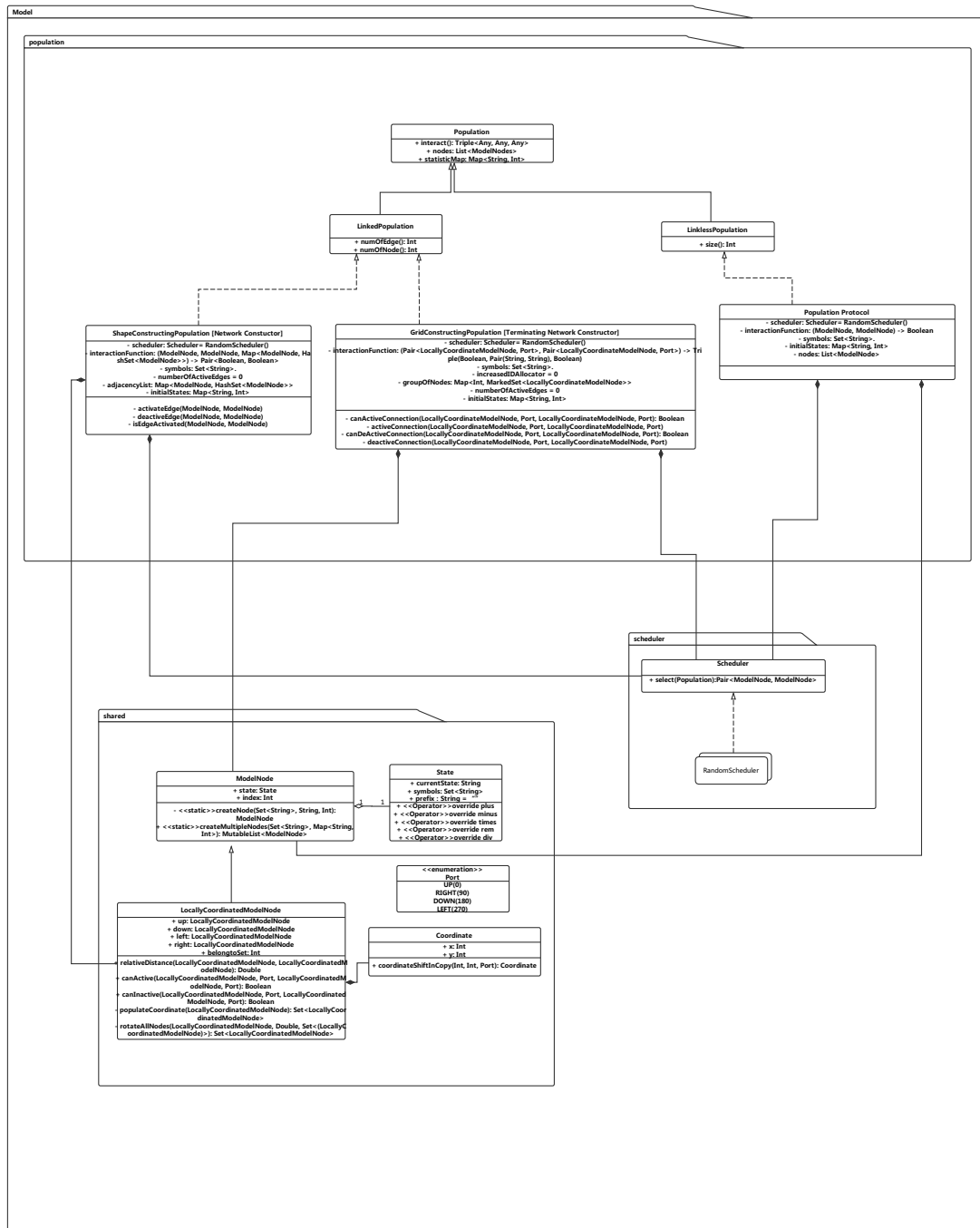


Figure 11: Object-oriented Design: Model Part

5 Realisation

5.1 From Start of Realisation to 3rd March, 2018: implement first two models

At very beginning, the first step was to attempt to implement the model of the models including general population protocol [1], the network constructor [2] and the terminating grid network constructor[3]. The first two is relatively easy because they are much easier to be represented than the third one using widely used data structure like adjacency list. For terminating grid network constructor, it is more complex in conception compared with other two models. Hence it is failed to be done during this initial development period.

Construct software architecture During the development period, the most interfaces of model partition are finished according to the UML object diagram mentioned in the *design* section. An example could be the scheduler interface

Listing 2: Interface design

```
//commit on Feb. 24th
interface Scheduler {
    fun select(population: Population): Pair<Node, Node>
}
```

and grid network class without implementation.

Listing 3: Grid network interface (early screenshot)

```
//commit on Feb. 24th
class NetworkConstructingPopulation constructor(scheduler: Scheduler
) : LinkedPopulation {
    override val nodes: List<Node>
        get() = TODO("not implemented") //To change initializer of
        created properties use File | Settings | File Templates.

    override fun numOfEdge(): Int {
        TODO("not implemented") //To change body of created
        functions use File | Settings | File Templates.
    }

    override fun numOfNode(): Int {
        TODO("not implemented") //To change body of created
        functions use File | Settings | File Templates.
    }

    override fun interact(): Boolean {
        TODO("not implemented") //To change body of created
        functions use File | Settings | File Templates.
    }
}
```

Though the method body not finished, these codes provides guidelines to the author for further development. The language adopted in the project is Kotlin [9]. This is A JVM language with support for pattern matching, high-order function abstraction and operator overloading [9], which were essential characteristics for the development of this simulator.

Basic method for Handle centered rotation In the early stage, the author did aware that it would be necessary to handle centered rotation issues for terminating grid network model, hence the author started to implement the rotation matrix, that is the equation 1.

Listing 4: Grid network interface (early screenshot)

```
import koma.end
import koma.extensions.get
import koma.extensions.map
import koma.mat
import koma.matrix.Matrix
import kotlin.math.roundToInt

//commit on Feb. 24th
data class LocallyCoordinatedNode(val x: Int,
                                   val y: Int,
                                   override val state: State, private
                                   val index: Int) : Node(state =
                                   state, index = index) {

    // IN-CLOCK direction rotation
    fun getLocallyRotatedNode(degree: Int): LocallyCoordinatedNode {
        val rad = degree * Math.PI / 180
        val transferMat = mat[
            Math.cos(rad), Math.sin(rad) end
            0 - Math.sin(rad), Math.cos(rad)
        ]
        val transferred = (transferMat * mat[this.x, this.y].T).
            removeNegativeZeros()
        return LocallyCoordinatedNode(transferred[0, 0].roundToInt(),
            transferred[1, 0].roundToInt(), state, index)
    }

    private fun Matrix<Double>.removeNegativeZeros(): Matrix<Double>
    {
        return this.map { it -> if (it == -0.0) Math.abs(it) else it
        }
    }
}
```

This is somehow a acceptable but naïve implementation for the only case that the rotation centre is the original point (0, 0) and does not efficient for centred rotation involves multiple nodes needed to be rotated (it might be optimised if taking the affine transformation form mentioned as in equation 3). The koma library, a scientific computing library written in Kotlin [10], is used here to simply the matrix calculation.

Because in Kotlin, the -0.0 is defined as a different with 0.0 for floating number and the value -0.0 is considered less than 0.0 [11], which causes issues in this calculation. Hence it is required to uniform the kind of different "0" using *removeNegativeZeros* method.

Test-driven development At very first of beginning, the test codes for simple global line protocol (for testing network constructor model) and dancing protocol (for testing general population protocol model) were written before any actual code is written. The test also included a testing for node rotation (used later on for terminating grid network constructor). These test code written under JUnit framework [12].

Here is an example testing for rotation.

Listing 5: Testing for rotation

```
//commit on Feb.24th
class LocallyCoordinatedNodeTest {

    @org.junit.Test
    fun rotationDegree() {
        // Create a node with locally coordinate (0, 0)
        val firstCase = LocallyCoordinatedNode(0, 0, State.
            createState(setOf("a", "b"), "a"), 1)
        // Make rotation
        val t90 = firstCase.getLocallyRotatedNode(90)
        val t180 = firstCase.getLocallyRotatedNode(180)
        val t270 = firstCase.getLocallyRotatedNode(270)
        val t360 = firstCase.getLocallyRotatedNode(360)

        assert(firstCase == t90)
        assert(firstCase == t180)
        assert(firstCase == t270)
        assert(firstCase == t360)
        // Create a node with locally coordinate (1, 1)
        val secCase = LocallyCoordinatedNode(1, 1, State.
            createState(setOf("a", "b"), "a"), 1)
        //Make rotation
        val tt90 = secCase.getLocallyRotatedNode(90)
        val tt180 = secCase.getLocallyRotatedNode(180)
        val tt270 = secCase.getLocallyRotatedNode(270)
        val tt360 = secCase.getLocallyRotatedNode(360)
        assert(LocallyCoordinatedNode(1, -1, State.createState(
            setOf("a", "b"), "a"), 1) == tt90)
        assert(LocallyCoordinatedNode(-1, -1, State.createState(
            setOf("a", "b"), "a"), 1) == tt180)
        assert(LocallyCoordinatedNode(-1, 1, State.createState(
            setOf("a", "b"), "a"), 1) == tt270)
        assert(secCase == tt360)
    }
}
```

5.2 From 3rd March to 26th March, 2018: Further progression

Abstraction of interaction function && implementation for some protocols During this stage, the three different interaction function concept appearing in the definition of the three models were abstracted and represented using a Kotlin method (or say, a Kotlin function).

- For general population protocol, it abstracted as

Listing 6: Abstraction for population protocol interaction function

```
fun protocolFunc(initializer: ModelNode, receiver:
    ModelNode): Boolean
```

where the 1st and 2nd parameters are the two nodes selected to interact, the return value indicates whether the interaction happens (*true* means it happens, *false* means it not).

- For network constructor, it abstracted as

Listing 7: Abstraction for network constructor interaction function

```
fun protocolFunc(initializer: ModelNode, receiver:
    ModelNode, adjacencyList: Map<ModelNode, HashSet<
    ModelNode>>): Pair<Boolean, Boolean>
```

where the 1st and 2nd parameters are the two nodes selected to interact, the 3rd parameter are the adjacency list (of which actual structure is a map) for the population that the first nodes located indicating the status of connection for the population. The first value in return pair indicates whether it is "effective" interaction (i.e. there is at least one changed state among 2 nodes and edge during the interaction, so *true* means it is "effective", *false* means not) while the second value in return pair indicates whether it is *active* state for connection in between the two nodes after interaction(*true* means there it is active, *false* means it is inactive).

- For terminating grid network, because the model is not well abstracted so it is not been abstracted successfully during this period.

Start of viewer implementation Consider the limited time for implementation, it reasonably considered as "infeasible" if building entire viewer system from sketch. The author used a dynamic graph library called "GraphStream" [13] to reduce the work amount to ensure a delivery on time. GraphStream provides a set of basic tools for modelling dynamic graphs, such as representing elements of nodes and edges, which exactly suit for the scenario of the simulator.

Failed attempt to Domain Specific Language As initial design proposed, the author attempted to implement the functionality that allow user load their protocol from a file. This involves loading a piece of user written code (in some Language) and then parse it into executable Kotlin code. For time limited, it would be hard to implement a parser. A few difficulties could be identified:

- The computability of different models differs from each other. It is hard to be defined a standard set of calculation required to be supported.
- The form of language defined in papers requires further semantic interpretation to be transformed as executable code. A simple protocol sum (modulo 4) [1] can illustrate this point. This protocol gathers the sum (modulo) of all agents to a single agent and the output comes from the unique agent with a non-null value. The detailed rule of the protocol could be found in appendix. Notice the rule $(v_1, v_2) \rightarrow (v_1 + v_2, \perp_{v_1+v_2})$ involves an addition operation according to the two inputs, an symbol substitution for one of output state (the second one) and a subscript addition for the second output state. It would be a requirement to a set of powerful language grammar to define these operation if the language is required to express these operations.

A compromised but feasible solution is to utilise some language characteristics provided by Kotlin language such as operator overloading and infix expression [9]. Though defining some semantic meaning of some operations on the *State* class, the transition function of sum (modulo 4) could be achieved. Here is an implementation using this method.

Listing 8: Implementation for transition function of sum (mod 4) protocol


```

fun sumModeFourFunc(initializer: ModelNode, receiver: ModelNode):
Boolean {
    var isChanged = false
    val transferred = when{
        Pair(initializer.state,receiver.state) match Pair("[0123]", "[0123]") ->{
            Pair((initializer.state + receiver.state)%4, "N" and ((
                initializer.state + receiver.state)%4))

        }
        Pair(initializer.state, receiver.state) match Pair("[0123]", "N[0123]") -> {
            Pair(initializer.state,"N" and initializer.state)
        }

        else -> null
    }

    if (transferred!=null){
        initializer.state = transferred.first
        receiver.state = transferred.second
        isChanged = true
    }

    return isChanged
}

```

Note that in the implementation, there is some "new" keyword "match" and "and". The "match" keyword is used for regular expression matching, where the first parameter before "match" keyword is the input state pair and the second parameter after this keyword is the pattern to be followed. The "and" keyword concatenates a string and a state to produce proper results according to different situation.

The *initializer.state* and *receiver.state* are instance of State. The state class finally overrides (i.e. defines) 5 different binary operations including plus, minus, multiplication, division and module. The defined operations can be modified or extended on request.

5.3 From 27th March to 8th April, 2018: Implement terminating grid network constructor

Model representation of terminating grid network constructor The model representation of grid network constructor has been covered in detail in the *design* section. The quad-direction linked structure was not initially proposed and was modified accordingly during this period of time. The structure allow the model be modelled mathematically to follow geometrical restrictions and limit the number of nodes connected to one nodes, which is also a limitation for this model.

View representation of terminating grid network constructor The basic concepts in the dynamic graph library, GraphStream [13], solely contains nodes, which may link to the "node (agent)" in the three types of models to be simulated, and the edges, which may link to the "(active) connection between two nodes" for network simulator and terminating grid network simulator. Nonetheless, there is no direct correspondence entity of "port" or "node

with port” in the viewer library GraphStream. It was required to define a node with ”port” and specify what the outlook (or representation) of port.

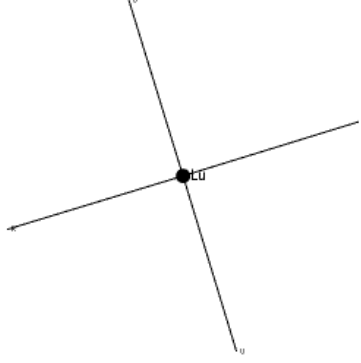


Figure 12: Screenshot for a ”node with port” in the viewer

The figure 12 is a typical representation of ”a node with 4 ports” in terminating grid network constructor model. The centred node, which is marked as L_u is represented through a node representation in GraphStream. Here, to avoid any confusions, any elements including nodes and edges in viewer part will be called as a representation while the nodes and edges in model part are still called its original name. A port p is actually another invisible node representation around the centred node L_u , and has visible line connected with L_u . Hence, a node in terminating grid network constructor will be represented as a connected graph with 5 nodes and 4 edges in a star-like topology.

Since the node representation is a graph in general, the basic manipulation of node representation in other will not suit for the node representation of terminating grid network. Suppose there is a node representation has a real coordinate (x_{np}, y_{np}) (It is called ”real coordinate” because the coordinate represents the true location of the node representation and is distinct to the concept ”relative coordinate” appearing in the model part.) and rotation degree θ_{np} , and it intends to another real coordinate (x_{nq}, y_{nq}) , the movement would follows steps:

- Move the centred node representation to coordinate (x_{nq}, y_{nq})
- Move the up port representation p_u to coordinate to $(x_{nq}, y_{nq} + 1)$, right port p_r representation to coordinate to $(x_{nq} + 1, y_{nq})$, down port representation p_d to coordinate to $(x_{nq}, y_{nq} - 1)$ and left port p_l representation to coordinate to $(x_{nq} - 1, y_{nq})$
- Do a centred rotation for p_u, p_l, p_r, p_d regarding the centre (x_{nq}, y_{nq}) with rotation degree θ_{np}

For rotation operation, after the centred node representation rotates, the 4 port representations are required to rotate the same degree that the centred node representation rotates.

Abstraction of interaction function of terminating grid network constructor After clarifying the model and view representation for the model, it finally could deduce the method signature for its transition function.

Listing 9: Abstraction for terminating grid network constructor interaction function

```
fun protocolFunc(
    firstPair: Pair<LocallyCoordinatedModelNode, Port>,
    secondPair: Pair<LocallyCoordinatedModelNode, Port>
): Triple<Boolean, Pair<String, String>, Boolean>
```

where the *firstPair* refers the initializer node and port selected to interact and the *secondPair* is the receiver. There are three variables in the returned triple, where the first one indicates whether it is an effective interaction (*true* if it is while *false* for it is not), the second pair is the states for the initializer node and receiver node respectively after the interaction, the third one indicates whether there is a "active" connection in between two ports selected after the interaction (*true* for there is while *false* for there is not).

5.4 From 9th April to 15th April, 2018: User Interface implementation and experiment

Pause during simulation process and Fast-forward a simulation at initial The pausing function was implemented during the entire process but finishes during the final stage of implementation. The viewer is following a "generator" pattern, which means its simulation is step based. At any time, it can stay in an intermediately state and then resume at any time hence naturally results such a functionality.

The fast-forwarding function allows user to specify a number for pre-executed selections (not number of interactions because some interactions may ineffective interactions), and then show the results on the viewer directly after such that numbers of selections. This is normally fast then simulate in a step by step way. The key of this pattern is that it simulates only in the model partition but does not demonstrate changes in the viewer part for initial steps, so this function will save a large amount of time from communication and synchronisation in between model and viewer, and so from visualization animation.

Protocol expansibility Despite of the DSL was failed to be implemented, it was still possible to find some other ways to simply the process to define users' own protocols. A possible solution might be reflection, which means "the ability of a program to manipulate as data something representing the state of the program during its own execution" [14].

The simulator will check three particular classes with specific name and check the functions and fields inside these classes that complies with some naming convention and load them if it possible when been launched each time. More specifically, it has some classes called "protocol container", which of all implement *InteractionFunctions* interface and have their own companion object inside it. A well defined protocol *i* contains three class static members with a same prefix, say *prefix_i*, in a protocol container. The three class static members are named as "*prefix_iInitialState*", "*prefix_iSymbol*" and "*prefix_iFunc*", which are a map with String key and Int value defining the initial configuration of the protocol, a set of String defining possible states that appearing in nodes and a function defines the transition function of a protocol.

- For population protocol, the container is called *PopulationProtocolFunctions*
- For network constructor, the container is called *ShapeConstructionFunctions*
- For terminating grid network constructor, the container is called *GridNetworkConstructingFunctions*

The $prefix_i$ of a protocol i is the "camel case" naming for the protocol and will be used in naming the protocol in user interface. For example, a prefix "squareGridNetwork" will be presented as Square Grid Network in the simulator user interface.

Experiment to find ineffective protocols After finished the implementation of all main functionalities and built some protocols, the author examined some protocols to check their performance and found that the dancing protocol may hard to converge under some settings. This is an evidence that the simulator is able dig into what happens inside a protocol. A further details would be discussed later in *evaluation* section.

6 Learning Points

This section will discussed some points that the author learned throughout the entire final year project process.

6.1 Academic skills

Basic understanding towards population protocols It was entirely an accidental for the author to select this topic as my final year project theme. Tracking back to one years' ago, the author saw the project description and find that it is possibly an attractive idea. This is an area but few researchers focusing on compared with current research hotspot such as deep learning. Though like that, the population protocol and network constructor provides a different perspective for what is computing and how computing works.

Applying mathematics into real project The research process involves some mathematics such as rotation matrix and trigonometric functions. Though they are very basic, it is still a good practice to apply these knowledge learnt in book to solving the practical problems. It is truly deepen my understanding towards linear algebra and trigonometric functions through working on this project.

Paper reading and information gathering Even for now, the author can only understand some partition of paper listed in bibliography but not all of them because of his very limited knowledge, especially for some related mathematical knowledge, such as probability theory and computational complexity theory. However, this is also good train for the author to quick handle papers using his limited knowledge and to quick gather useful information from related papers.

Academic writing and use of \LaTeX Though this is neither the first time that author write academic pieces nor first time to use \LaTeX , this is first time that the author using \LaTeX to finish a document that more than 20 pages.

Learning direction: Foundation knowledge The author found that he had some essential mathematic basics is not covered in his previous studies and may start to his self-study in the near future.

6.2 Technical (Development) skills

Learnt a new language Kotlin and gained more understanding on programming language It was a proper choice for the author to take a risk for learning and using Kotlin rather than Java to finish the project. Many language characteristics in Kotlin assisted the development process and simply the implementation. The pattern matching, operator overloading and infix keyword deepen the author's understanding towards programming language.

Had a better understanding on the relationship in between interface and model The application is all about how to cooperate the interface and model to enable them to work correctly and coherently. Throughout the project, it took the author a large amount of time to construct the architecture of model and user interface and bridging them together.

6.3 Soft skills

Problem reduction and problem solving There were many problems that encountered in designed and development process finally resolved through reduce problem to simpler case. For instance, the overlapping problem for terminating grid network constructor could be resolved via discussion solution under different situations. These solutions under different situations finally become the answer to resolve the original problem.

Time management This entire development process teaches the author a lesson that practice plan vitals. The author was failed to do so and led to a bad experience during last period of development and dissertation writing.

Oral presentation The failure during requirement presentation enforced the author rethinking how to properly deliver his context to his audience and optimise his strategy in his final presentation through starting from easy concepts rather than presenting all core concepts at initial.

Bibliography

- [1] J. Aspnes and E. Ruppert, “An introduction to population protocols,” *Bulletin of the European Association for Theoretical Computer Science*, vol. 93, pp. 98–117, Oct. 2007.
- [2] O. Michail and P. G. Spirakis, “Simple and efficient local codes for distributed stable network construction,” *Distributed Computing*, vol. 29, no. 3, pp. 207–237, 2016. [Online]. Available: <https://link.springer.com/article/10.1007/s00446-015-0257-4>
- [3] O. Michail, “Terminating distributed construction of shapes and patterns in a fair solution of automata,” *Distributed Computing*, pp. 1–23, 2017. [Online]. Available: <http://link.springer.com/article/10.1007/s00446-017-0309-z>
- [4] O. Michail, I. Chatzigiannakis, and P. G. Spirakis, *New Models for Population Protocols*, ser. N. A. Lynch (Ed), Synthesis Lectures on Distributed Computing Theory, N. A. Lynch, Ed. Morgan & Claypool, 2011.
- [5] D. Amaxilatis, M. Logaras, O. Michail, and P. G. Spirakis, “NETCS: A new simulator of population protocols and network constructors,” *CoRR*, vol. abs/1508.06731, 2015. [Online]. Available: <http://arxiv.org/abs/1508.06731>
- [6] S. Widnall, “Lecture 13 - vectors, matrices and coordinate transformations,” Fall 2009. [Online]. Available: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-07-dynamics-fall-2009/lecture-notes/MIT16_07F09_Lec03.pdf Accessed: 2018-04-30.
- [7] E. Weisstein, “Affine transformation.” [Online]. Available: <http://mathworld.wolfram.com/AffineTransformation.html> Accessed: 2018-05-01.
- [8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.
- [9] JetBrains, “Kotlin language.” [Online]. Available: <https://kotlinlang.org/api/latest/jvm/stdlib/org.w3c.fetch/-m-a-n-u-a-l.html> Accessed: 2018-05-01.
- [10] K. Kauffman, “Koma.” [Online]. Available: <http://koma.kyonifer.com/> Accessed: 2018-05-01.
- [11] JetBrains, “Floating point numbers comparison in kotlin.” [Online]. Available: <http://kotlinlang.org/docs/reference/basic-types.html#floating-point-numbers-comparison> Accessed: 2018-05-01.
- [12] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, and C. Stein, “Junit.” [Online]. Available: <https://junit.org/junit4/> Accessed: 2018-05-01.
- [13] GraphStream Team, “Graphstream, a dynamic graph library.” [Online]. Available: <http://graphstream-project.org/> Accessed: 2018-05-01.
- [14] J. Malenfant, M. Jacques, and F. Demers, “A tutorial on behavioral reflection and its implementation.” [Online]. Available: <http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/malenfant.pdf>

A Brief Derivation for equation (1)

For convenience, here name $\angle CAB$ as α in figure 3. Then it has

$$|AD| = |\vec{r}'| \cos(\alpha - \theta)$$

$$|B'D| = |\vec{r}'| \sin(\alpha - \theta)$$

By applying angle sum identities,

$$|AD| = |\vec{r}'|(\cos(\alpha) \cos(-\theta) - \sin(\alpha) \sin(-\theta))$$

$$|B'D| = |\vec{r}'|(\sin(\alpha) \cos(-\theta) + \sin(-\theta) \cos(\alpha))$$

Because $|\vec{r}'| = |\vec{r}|$, $\sin(\theta) = -\sin(-\theta)$ for $\theta \in [0, \pi/2)$ and $\cos(\theta) = \cos(-\theta)$ for $\theta \in [0, \pi/2)$,

$$|AD| = |\vec{r}|(\cos(\alpha) \cos(\theta) + \sin(\alpha) \sin(\theta))$$

$$|B'D| = |\vec{r}|(\sin(\alpha) \cos(\theta) - \sin(\theta) \cos(\alpha))$$

Then,

$$|AD| = (|\vec{r}| \cos(\alpha)) \cos(\theta) + (|\vec{r}| \sin(\alpha)) \sin(\theta)$$

$$|B'D| = (|\vec{r}| \sin(\alpha)) \cos(\theta) - (|\vec{r}| \cos(\alpha)) \sin(\theta)$$

Hence,

$$|AD| = |AC| \cos(\theta) + |BC| \sin(\theta)$$

$$|B'D| = |BC| \cos(\theta) - |AC| \sin(\theta)$$

Finally,

$$|x'| = |x| \cos(\theta) + |y| \sin(\theta)$$

$$|y'| = |x|(-\sin(\theta)) + |y| \cos(\theta)$$

For $\theta \in [0, \pi/2)$, the equations can be written simply as

$$x' = x \cos \theta + y \sin \theta$$

$$y' = x(-\sin \theta) + y \cos \theta$$

which can be easily extended to $\theta \in [0, 2\pi + 2k\pi)$ via discussion under different cases, given that $k \in \text{Integer}$.

The two equations can be written in matrix form :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

B Brief Derivation for equation (3)

The equation 2 can be transformed to:

$$\begin{aligned}\begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} c_x \\ c_y \end{bmatrix} \right) + \begin{bmatrix} c_x \\ c_y \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} c_x \\ c_y \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} c_x \cos \theta + c_y \sin \theta \\ -c_x \sin \theta + c_y \cos \theta \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \end{bmatrix} \\ &= \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} c_x(1 - \cos \theta) - c_y \sin \theta \\ c_y(1 - \cos \theta) + c_x \sin \theta \end{bmatrix}\end{aligned}$$

C List of Implemented Protocol

D Open source libraries used in the simulator

Credits to:

- Koma, A scientific computing library, under Apache 2.0, <http://koma.kyonifer.com/>
- TornadoFX, JavaFX Framework for Kotlin, under Apache 2.0, <https://github.com/edvin/tornadofx>
- GraphStream, A Dynamic Graph Library, under LGPLv3, <http://graphstream-project.org/>

Additionally, credit to NETCS, A Java implementation for non-grid dynamic network, under 3-Clause BSD, <https://github.com/amaxilat/netcs>, this is not used in the simulator but give an some inspiration to author of the simulator.