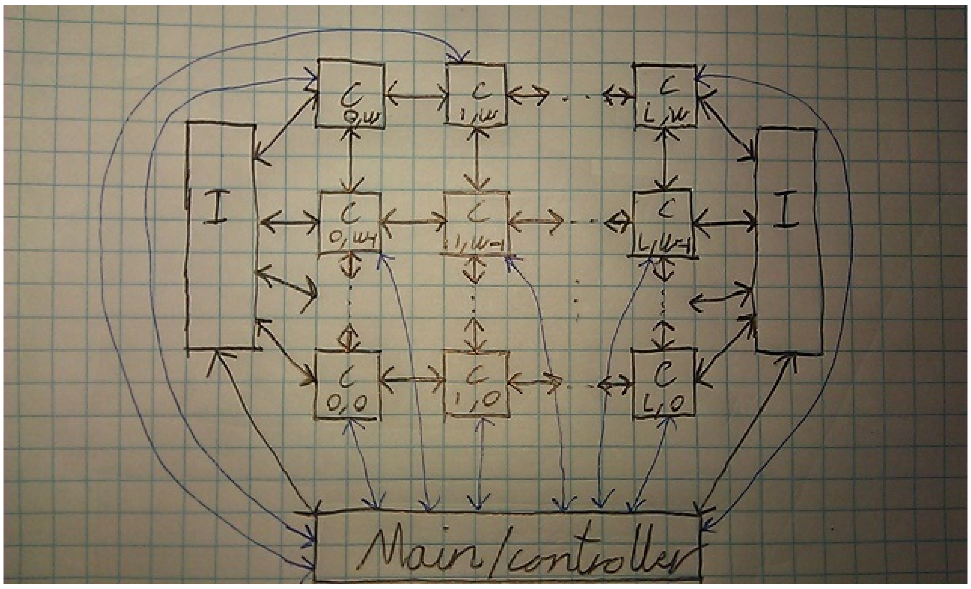
XMP 2013/14 Exam (Programming)

Henrik Bendt (gwk553)

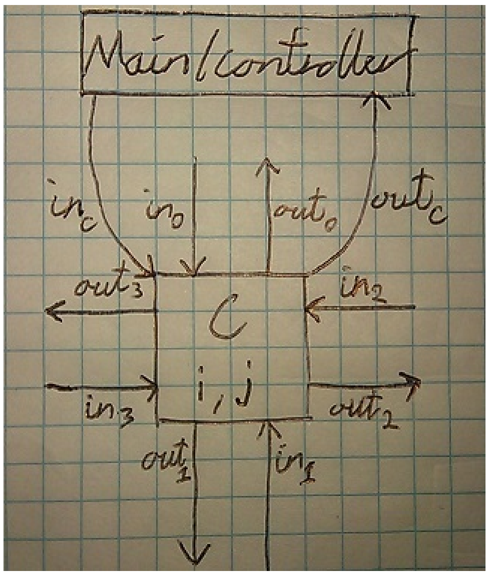
24. januar 2014

1 Design

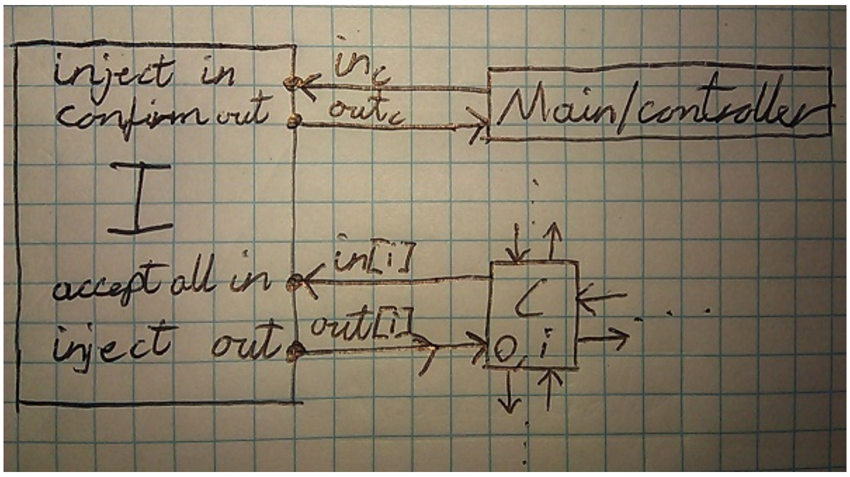
The automaton is designed with *cell*s as processes (I shall call processes threads from here on out)and pedestrian as data exchanged on channels between theads. Each cell has channels to its 4 cartesian neighbors (which can be nil or to an injector), and also maintains a channel to the main thread; the *controller*. At each end of the automaton is an *injector*, handling injections of pedestrians going opposide directions, that is, towards the other injector. All threads communicate by 0-buffered, one-directional, 1-1 channels and share no data.



Figur 1: The architecture of the automaton. All channels are 0-buffered, one-directional, 1-1 channels.



Figur 2: The design of a cell in the automaton. The cell has some in and out channels to other cells (or an injector), named after the direction it points (0 is north, 1 is south, 2 is east and 3 is west). If cell is at the border, the channels pointing out of the automaton is nil. A cell also has an in and out channel to the controller.



Figur 3: The design of the west (left-most) injector. The east (right-most) injector has the same design. The injector has an inject channel from the controller, telling the injector to inject. After injection to an adjecent cell, the injector confirms the inject with the controller. If no adjecent cell is free, the injector tells the controller of the congestion and the inject is lost. The injector accepts all input from the list of cells adjecent to it (like a drunk moving out of the automaton or a move by a pedestrian).

The automaton can be divided into the following phases of a step:

1. Initial phase: injecting and moving pedestrians and drunks.
2. User phase: registering and acting upon user input.
3. Drunk phase: spreading the drunk effect upon cells within the range of 3 cells from a cell containing a drunk.
4. End phase: reporting all current positions of pedestrians and drunks.

The controller handles the step progression. A step includes synchronization of the cells with the controller two times. A synchronization means that the controller recieves a message from a cell and sends one back after recieving a message from all cells in the automaton. Synchronization happens at the end/initial phase and at the drunk phase. This is to make sure drunk effect is applied correctly and that the effect does not change/move while pedestrians are moving, but unfortunately this also limits the concurrency of the automaton compared to just synchronizing once at each step iteration.

All cells do non-blocking communication at almost all time, i.e. both sending and recieving is non-blocking, which limits the possibilities of deadlocks. The only times a cell does a blocking communication is when it responds to a move request of a drunk or a pedestrian and when responding to a spread-drunk-effect-request. This is because the cell moving is (non-blockingly) waiting for the response, but the cell does not want to accept any other requests from other cells. The injector always blocks when responding on requests from cells or controller, and it always accepts all input. The controller blocks on all communication with the automaton, as it does not need to respond on any input except when synchronizing the automaton, which is also done blockingly. User input is on a seperate thread, and the controller accepts input from this thread once every step.

When a pedestrian tries to make a move to a cell, it awaits the response from that cell before trying to move to another cell. The cell with the waiting pedestrian will still accept input from other cells which can be move-requests too. These requests will be declined as long as a pedestrian is still active on cell. This is because it cannot guarantee that the current pedestrian will move this step (because all possible moves can be blocked), so it cannot accept other pedestrians and risk ending up with two pedestrians when the step ends. This logic was necessary to not end up with a deadlocked cirkle of requests between the cells. When two pedestrians tries to make a move to the same cell, the *winner* is the one successfully sending a request to the cell, which is pseudo-random depending on the order the cell looks at the channels and if it looks at the right time (remember all sending is non-blocking, so a sender will at some point time out and listen for input before trying to send again).

To awoid spreading drunk effect while pedestrians and drunks are moving, the drunk effect is spread in its own phase. At this phase it is ensured that both pedestrians and drunks are standing still. Drunk effect is spread by the following algoritm:
Sending request:

1. Send drunk-effect-request to a neighbor with a counter set to 3. Await the response while also checking possible incomming requests. When response is registered, send same request to other neighbors.

Recieving request:

1. Recieve drunk-effect-request from neighbor a.
2. If counter equal 1, no more spreading required. Respond with confirmation to a.
3. If counter greater than 1, do sending request with decremented counter to all neighbors except a. When all neighbors have confirmed, send confirmation to a.
4. If already spreading drunk effect given by a cell b and recieving, from cell c, a drunk effect with greater counter than the one already accepted, free b by responding with a confirmation to it and spread the new, higher value drunk effect to all but c. When done, send confirmation to c.

1.1 Likelihood of deadlocks

An injector always await input on all channels and it always accept all input. The only time an injector do not await input is when it is injecting or confirms a request from a cell or controller. As all cells always check input from all neighbors, injection should not deadlock. As the controller will (blockingly) await responses from the injectors after they are requested to inject, this blocking response should not deadlock. As the cell awaits a response from a request and does not do anything else but accept input while waiting for a response, this should not deadlock unless an intermediate request to the cell deadlocks. The controller is the only always-blocking communicator, by which all cells in the automaton synchronizes. As the controller always waits for input from all cells before sending output to any of them (like when starting a new step), the only way the controller can deadlock and indeed the whole automaton is if a cell at some point is not sending to or recieving from the controller. Thus it all comes down to the cells.

Cells accepts input from all other cells at all time except when responding to a request. As a request cannot be accepted unless the sender knows, the sender will always expect a response from the cell accepting its request. Thus this should not deadlock. Also, as mentioned above with the pedestrians, we cannot get a circle deadlock, as a move of a pedestrian does not block a cell.

Spreading drunk effect at its own phase limits the number of ways to deadlock, as a cell can only be accepting or spreading drunk effect in this phase and at no other time. While recieving a spread-drunk-effect-request, the sender will wait for a response, but still accept input from other cells while waiting. If recieving a spread-drunk-effect-request, the cell will blockingly respond immidiately, but the sender should be waiting for this response. If the sender, while waiting for a response, gets a request, it will blockingly respond to that. Then again if that sender gets a request before recieving, it will also be blockingly responding. Thus this block-response can create a chain of blocking-response-sending cells, but it can never make a circle (and thereby deadlock), as once a cell has send a request it cannot send another and thus the last sender must have no one to recieve requests from, and thereby cannot deadlock.

2 Control Interface

The following commands can be used to manipulate the simulation while running:

- “pause” pauses the simulation. Other commands can be executed while paused. Pausing simply means not initiating the next step before told to by the user.
- “resume” resumes the simulation if it was paused, otherwise just keeps running the simulation.

- “rate x” changes the inject rate of the simulation to x, if $0 < x \leq 10$.
- “addDrunk x y” adds a drunk at the position (x,y), if this is within the boarders of the automaton, that is, if $0 \leq x < length$ and $0 \leq y < width$, as the board is 0-indexed.
- “exit” terminates the simulation.

Note that all commands takes effect in the middle of a step, that is, after a step has begun and before outputting the status of the current board.

3 Initialization

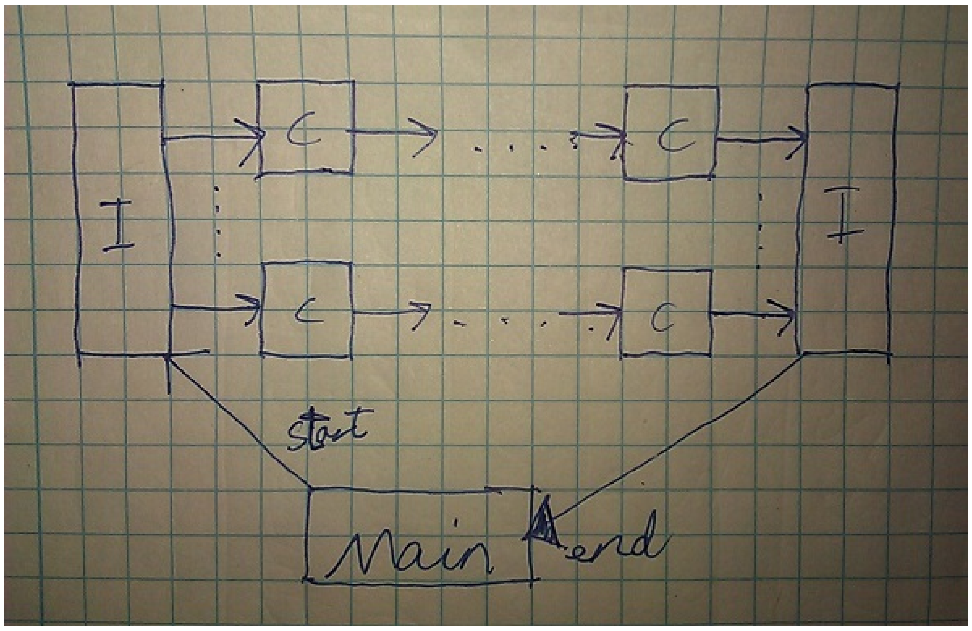
The board dimensions are defined as a length and a width, where the length corresponds to the x-axis, and the width to the y-axis. The length and width can be set in the main function along with the initial inject rate (which can be changed via the command interface of the running simulation) and the maximum number of steps.

The current solution is hard coded to have the two inject directions east and west. Also, the implementation can not accept other ways to inject than two opposite inject directions and the board must be rectangular. Even though these limitation could easily be extended, time did not allow it.

4 Termination

Upon termination, one of the injectors is told to kill the cells and then die by the main thread. The cells adjecent to the injector is told to die by the injector and when all adjecent cells are told, the current injector terminates. Each cell tells the next cell in the direction of the last living injector to die. Thus each row of cells dies indipendently of other rows of cells and does so without affecting eachother, so all cells are guaranteed to die. An alternative could have been to use the controller, which has channels to all cells, to tell each cell to die. This would make a slower termination, as it is sequential, and also can only be done at synchronization steps. Thus I chose not to do this. The implemented solution would not work if the automaton would not be a filled rectangular, but for example have holes (i.e missing cells) or have uneven boarders. Then the alternative solution would be prefered or a whole other solution, for example one where each cell tells all neighboring cells to die, not just the next in line (which is also very concurrent).

When the last injector recieves a die-notification from one of its adjecent cells, it goes into a die-mode, whereby it expects to receive die-notifications from the rest of its adjecent cell. When this has happened, it tells the main thread that it is terminating and then terminates. The main thread now knows that the automaton have terminated correctly and now only needs to kill the input thread. To do this, the user is asked to input any text in the console, as the input thread blocks on stdin (there is as far as I could find no non-blocking way for Golang to recieve input from stdin). By recieving input on stdin, the thread unblocks and is told to terminate. Now the main thread, as the last thread, can terminate.



Figur 4: The design of the termination order of the automaton.

Note that each timeout thread of the cells are also terminated by a recieving input on a special quit-channel, which the individual cell calls before terminating. This call is blocking, so the cell does not terminate before the timeout thread is told to terminate.

5 Implementation

The solution is written in Golang. This means that in order to do non-blocking in/out-put, I had to use a timout-thread for each cell in the cellular automaton, because in Golang, non-blocking threads only read/write on a channel, if it is blocked in the other end. So in order to give a chance for this to happen, a construct with a loop around a select-statement with two cases, one being the actual case the thread is trying to communicate on and the other being for input from the timeout thread channel. If no communication has occured on the first case, then at some point the timeout channel (which gets a message with a random interval between 0 and 49 miliseconds) will recieve a message and the thread can look at input channels. After it has looked at all other input channels, it loops back to repeat the above. This is, as far as I know, the only way for channels to not block on sending/recieving on a list of channels in golang. Unfortunately this also takes up an overhead of space and CPU-cycles per thread.

Golang channels are bi-directional, but should only be used as a unidirectional (which is also more csp). Thus I have made the struct Bichannel, which contains two channels, one to use on input and one on output. This struct is used in most situations throughout the program, as most situations require two-way communication. Also, channels are typed in Golang. Most channels in this implementation uses the custom defined message struct type, which consists of multiple types of data along with a tag, a string, defining the message send. This is always set when communicating.

The snapshot of the current state of the automaton is dumped as text to the console. The text consists of the current step and then a line for each pedestrian (it has a unique id, the inject number) and drunk with their current coordinates.

I have also made all threads explicitly write out when dying, so it is easy to see that all cells and injectors die.

6 Source Code

```go
package main

import (
    "fmt"
    "math/rand"
    "time"
    "strconv"
    "runtime"
    "os"
)

//Directions with enum type
type Direction int
const (
        N Direction = iota //0
        S //1
        E //2
        W //3
)

type Position struct {
    X, Y int
}

type Pedestrian struct {
    Id int
    Dir Direction
}

//Struct for messages send over channels
type Message struct {
    Tag string
    Pos Position
    Test bool
    Id int
    Pedestrian Pedestrian
    Drunk bool
    DrunkArea int
}
//Struct for a bidirectional channel, where a process only sends on Out and recieves on another In.
type Bichannel struct {
    In chan Message
    Out chan Message
}
//make a new Bichannel, return both versions of it
func makeNewBichannel() (Bichannel, Bichannel) {
    ch1 := make(chan Message)
    ch2 := make(chan Message)
    return Bichannel{In: ch1, Out: ch2}, Bichannel{In: ch2, Out: ch1}
}

func oppositeDirection(d Direction) Direction {
    switch d {
        case N:
            return S
        case S:
            return N
        case E:
            return W
        case W:
            return E
    }
    return d //should never happen
}

func getDirString(d Direction) string {
    switch d {
        case N:
            return "N"
        case S:
            return "S"
        case E:
            return "E"
        case W:
            return "W"
    }
    return "" //should never happen
}

/*
    A cell is an active part of the automaton.
    Listens on neighbors and controller channels
    If occupied, tries to push pedestrian to next cell at each step.
    Same with drunk.
*/
func cell(pos Position, neighbors [4]Bichannel, controlCh Bichannel){
    rand.Seed(time.Now().UTC().UnixNano()) //seed random generator
    var msg Message
    var pedestrian Pedestrian
    var orthoDirections [2]Direction
    drunk := false
    drunkEffect := false
    occupied := false


    numNeighbors := 0
    for i := range neighbors {
        if neighbors[i] != (Bichannel{}){
            numNeighbors++
        }
    }

    timeout := make(chan bool)
    timeoutQuitCh := make(chan bool)
    go func() {
        for {
            select {
            case <-timeoutQuitCh:
                runtime.Goexit()
            case timeout <- true:
                time.Sleep(time.Duration(rand.Intn(50)) * time.Millisecond)
            }
        }
    }()
```

```
occupy := func(ped Pedestrian){
    occupied = true;
    pedestrian = ped
    direction := pedestrian.Dir
    if direction == N || direction == S {
        orthoDirections = [2]Direction{E, W}
    } else if direction == E || direction == W {
        orthoDirections = [2]Direction{N, S}
    }
}
leave := func(){
    orthoDirections = [2]Direction{}
    pedestrian = Pedestrian{}
    occupied = false
}

killNext := func(i int){ //kills cell opposite of i
    var d Direction
    switch i {
    case 0:
        d = S
    case 1:
        d = N
    case 2:
        d = W
    case 3:
        d = E
    default: //should not happen
        fmt.Println("Cell: error when killing")
        return
    }
    if neighbors[d] != (Bichannel{}) {
        neighbors[d].Out<-Message{Tag: "die"}
    }
    timeoutQuitCh<-true
    runtime.Goexit()
}

spreadDrunk := func(fromIndex, area int){
    neighborSend := [4]bool{false, false, false, false}
    reset := false
    var count int
    if fromIndex < 0 {
        count = numNeighbors
    } else {
        count = numNeighbors-1
    }

    listenNeighborsDrunk := func(i int){
        for j, ch := range neighbors {
            if j != i {
                select {
                case msg =<-ch.In:
                    switch msg.Tag {
                    case "die":
                        neighbors[fromIndex].Out<-Message{Tag: "drunkArea", Test: true} //free f
                        killNext(j)
                    case "drunkArea":
                        if !msg.Test && msg.DrunkArea-1 > area {
                            neighbors[fromIndex].Out<-Message{Tag: "drunkArea", Test: true} //fr
                            fromIndex = j //spread new, higher value drunk area
                            area = msg.DrunkArea-1 //restart
                            count = numNeighbors-1
                            neighborSend = [4]bool{false, false, false, false}
                            reset = true
                        } else { //confirm
                            neighbors[j].Out<-Message{Tag: "drunkArea", Test: true}
                        }
                    default: //should not happen
                        fmt.Println("Cell: error in msg tag when spreading drunkness", msg)
                    }
                default: //skip
                }
            }
        }
    }

    for count > 0 {
        for j, ch := range neighbors {
            if j != fromIndex && ch != (Bichannel{}) && !neighborSend[j] {
                select {
                case ch.Out<-Message{Tag: "drunkArea", DrunkArea: area, Test: false}:
                    success := false
                    for !success {
                        select {
                        case msg =<-ch.In: //get confirmation
                            success = true
                            switch msg.Tag {
                            case "die":
                                neighbors[fromIndex].Out<-Message{Tag: "drunkArea", Test: true}
                                killNext(j)
                            default:
                                if !reset {
                                    neighborSend[j] = true
                                    count--
                                } else {
                                    reset = false
                                }
                            }
                        case <-timeout:
                            listenNeighborsDrunk(j)
                        }
                    }

                case <-timeout:
                    listenNeighborsDrunk(-1)
                }
            }
        }
    }
    if fromIndex >= 0 { //confirm
        neighbors[fromIndex].Out<-Message{Tag: "drunkArea", Test: true}
    }
}
```

```go
listenNeighbors := func(){
    for i, c := range neighbors {
        select {
        case msg =<-c.In:
            switch msg.Tag {
                case "occupy":
                    if occupied {
                        c.Out<-Message{Tag: "occupy", Test: false}
                    } else {
                        c.Out<-Message{Tag: "occupy", Test: true}
                        occupy(msg.Pedestrian)
                    }
                case "die"://Must die. Kill next
                    fmt.Println("cell: die")
                    killNext(i)
                case "drunkMove":
                    if !drunk && msg.Drunk {
                        drunk = true
                        drunkEffect = true
                        c.Out<-Message{Tag: "drunkMove", Test: true}
                    } else {
                        c.Out<-Message{Tag: "drunkMove", Test: false}
                    }
                case "drunkArea": //spread drunk area
                    drunkEffect = true
                    k := msg.DrunkArea
                    if k > 1 {
                        spreadDrunk(i, k-1)
                    } else {
                        c.Out<-Message{Tag: "drunkArea", Test: true} //confirm
                    }
                default: //errorfull message tag, should not happen
                     fmt.Println("cell: Error in tag", msg.Tag, "pos:", pos)
            }
        default://skip
        }
    }
}


drunkMove := func(){
    i := rand.Intn(5)
    if i < len(neighbors) {
        ch := neighbors[i]
        if ch == (Bichannel{}) { //Check if ch is valid
            drunk = false //drunk goes away
            return
        }
        success := false
        for !success {
            select {
            case ch.Out<-Message{Tag: "drunkMove", Test: false, Drunk: true}:
                for !success {
                    select{
                    case msg =<-ch.In:
                        if msg.Tag == "drunkMove" && msg.Test {
                            drunk = false
                        }
                        success = true
                    case <-timeout:
                        listenNeighbors()
                    }
                }
            case <-timeout: //force to listen to neighbors for a while
                listenNeighbors()
            }
        }
    }
}

pedMove := func(ch Bichannel) {
    if ch == (Bichannel{}) { //Check if ch is valid
        return
    }
    success := false
    for !success {
        select {
        case ch.Out<-Message{Tag: "occupy", Pedestrian: pedestrian}: //try to move ped to next c
            for !success {
                select{
                case msg =<-ch.In:
                    if msg.Tag == "occupy" && msg.Test {
                        leave()
                    }
                    success = true
                case <-timeout:
                    listenNeighbors()
                }
            }
        case <-timeout: //force to listen to neighbors for a while
            //skip
        }
        listenNeighbors()
    }
}

waitNextStep := func(){
    drunkEffect = false //reset drunkeffect. May be added again after done-check if drunk still
    //synchronize cells, drunk effect spread phase
    step := false
    for !step {
        select{
        case controlCh.Out<-Message{Tag:"status", Pos: pos}:
        case msg =<- controlCh.In: //step 1
            if msg.Tag == "drunkMove" {
                drunk = true
            } else {
                step = true
            }
        case <-timeout: //Force to listen for eventual die requests
            listenNeighbors()
        }
    }

    if drunk { //spread drunk effect
```

```go
            drunkEffect = true
            spreadDrunk(-1, 3)
        }

        //synchronize cells, next step
        step = false
        for !step {
            select{
            case controlCh.Out<-Message{Tag:"status", Pos: pos, Test: occupied, Pedestrian: pedestri
            case msg =<- controlCh.In: //step
                step = true
            case <-timeout: //Force to listen for eventual die requests
                listenNeighbors()
            }
        }
    }

    <- controlCh.In //initial step
    for {
        waitNextStep()
        if occupied { //only listen on control channel if occupied
            if !drunkEffect {
                pedMove(neighbors[pedestrian.Dir])
            }
            if occupied { //prime direction not possible, try orthogonal directions
                i := rand.Intn(2)
                pedMove(neighbors[orthoDirections[i]])
                if occupied { //try other orthogonal direction
                    i++
                    pedMove(neighbors[orthoDirections[i % 2]])
                }
            }
        }
        if drunk {
            drunkMove()
        }
    }
}




/*
    Used at the ends of the side walk.
    Injects new pedestrians when told by the controller.
    Also kills cells on termination (and terminate itself).
    Dies if all closest cells tells it to.
    direction is the direction to put new pedestrians.
*/
func injector(cellChs []Bichannel, controlCh Bichannel, direction Direction){
    var msg Message

    die := func(index int){
        //expect "die" messages from all other adjecent cells but the initial one
        for i, c := range cellChs {
            if i != index {
                found := false
                for !found {
```

```
                msg=<-c.In
                if msg.Tag == "die" { //TODO can be removed, should not happen.
                    found = true
                }
            }
        }
    }
    controlCh.Out<-Message{Tag: "die"}
    fmt.Println("injector: die end")
    runtime.Goexit()
}

for {
    select {
    case msg =<-controlCh.In: //wait for time step
        switch msg.Tag {
        case "inject":
            //Inject at first free cell, chosen in random order.
            pedestrian := Pedestrian{Dir: direction, Id: msg.Id}
            found := false
            shuffledList := rand.Perm(len(cellChs))
            for _, i := range shuffledList {
                ch := cellChs[i]
                if !found {
                    ch.Out<-Message{Tag: "occupy", Pedestrian: pedestrian}
                    msg =<-ch.In
                    if msg.Tag == "occupy" {
                        found = msg.Test
                    }
                }
            }
            if !found {
                controlCh.Out<-Message{Tag: "congestion", Test: true}
            } else {
                controlCh.Out<-Message{Tag: "congestion", Test: false}
            }
        case "die":
            fmt.Println("injector: kill-die")
            for _, ch := range cellChs {
                ch.Out<-Message{Tag: "die"}
            }
            fmt.Println("injector: kill-die end")
            runtime.Goexit()
        default:
        }
    default://check inputs from cells, accept everything
        for i, ch := range cellChs {
            select {
            case msg =<-ch.In:
                switch msg.Tag {
                case "occupy": //accept all pedestrians. They are now done.
                    ch.Out<-Message{Tag: "occupy", Test: true}
                case "die":
                    fmt.Println("injector: die")
                    die(i)
                case "done":
                    ch.Out<-Message{Tag: "done"}
                case "drunkArea":
```

```go
                        ch.Out<-Message{Tag: "drunkArea", Test: true}
                    case "drunkMove":
                        ch.Out<-Message{Tag: "drunkMove", Test: true}
                    default://error, should not happen
                        fmt.Println("injector eror")
                    }
                default://skip
                }
            }
        }
    }
}

/*
    Controller of the automaton.
    Interacts with the user.
    Controls the inject rate and tells injector when to inject.
    Injects drunks at given locations.
    Synchronizes cells (making steps).
*/
func control(injectChs []Bichannel, cellChs [][]Bichannel, initInjectRate, maxTimeSteps int){
    var s string
    var msg Message
    inputCh := make(chan string)
    inputQuitCh := make(chan string)
    injectRate := initInjectRate //Init should be 4
    injectCounter := 0
    pedestrainCount := 0
    pause := false

    userInput := func(c, quit chan string) {//TODO how to close, fmt.Scan blocks....
        var s string
        for {
            fmt.Scan(&s)
            select {
            case <-quit:
                runtime.Goexit()
            case c<-s:
            }
        }
    }

    userInputHandler := func(s string) bool {
        switch s {
        case "exit":
            fmt.Println("Terminating")
            injectChs[0].Out<-Message{Tag: "die"} //tell first injector to die.
            for i, ch := range injectChs { //wait for injectors to die.
                if i>0 { //first injector die silent
                    msg =<-ch.In
                }
            }
            //when last injector is dead, all cells are already dead too.
            fmt.Println("Enter any text to exit")
            inputQuitCh<-"" //this is the ONLY way to kill a thread reading stdin in go (making a us
            return true //terminate
        case "pause":
            fmt.Println("pausing")
            pause = true
        case "resume":
            fmt.Println("resuming")
            pause = false
        case "rate":
            s =<-inputCh
            i, _ := strconv.Atoi(s)
            if 1 <= i && i <= 10 {
                injectRate = i
                fmt.Println("Inject rate changed to", i)
            } else {
                fmt.Println("Given inject rate not between 1 and 10")
            }
        case "addDrunk": //at drunk at given position
            s =<-inputCh
            x, _ := strconv.Atoi(s)
            s =<-inputCh
            y, _ := strconv.Atoi(s)
            xMax := len(cellChs)
            yMax := len(cellChs[0])
            if 0 <= x && x < xMax && 0 <= y && y < yMax {
                //add drunk at position.
                cellChs[x][y].Out<-Message{Tag: "drunkMove"}
                //<-cellChs[x][y].In //block until drunk is placed.
            } else {
                fmt.Println("Invalid position given! Must be within automaton, dimension:", xMax, "X
            }
            fmt.Println("Drunk added")
        default:
            fmt.Println("Unknown command")
        }
        return false
    }

    checkUser := func() {
        select {
        case s = <-inputCh: //TODO: this is only time user input is registered!
            if userInputHandler(s) {
                fmt.Println("Controller exiting")
                os.Exit(0)
            }
            for pause { //wait until unpause
                fmt.Println("System paused")
                s =<-inputCh
                if userInputHandler(s) {
                    fmt.Println("Controller exiting")
                    os.Exit(0)
                }
            }
        default: //skip
        }
    }

    go userInput(inputCh, inputQuitCh) //thread to handle user input

    for i:=0; i < maxTimeSteps; i++ {

        fmt.Println("Controller: step number", i)
        //next timestep
        for _, l := range cellChs {
            for _, c := range l { //then nextstep cells
                c.Out<-Message{Tag: "next"}
            }
        }
        //inject phase
        injectCounter++
        if injectCounter >= injectRate { //inject time!
             //one new pedestrian per injector
            for _, c := range injectChs {
                c.Out<-Message{Tag: "inject", Id: pedestrainCount}
                pedestrainCount++
            }
            //Check for congestion
            for _, c := range injectChs {
                msg =<-c.In
                if msg.Tag == "congestion" && msg.Test{
                    fmt.Println("Congestion! Unable to inject new pedestrian at one end.")
                }
            }
            injectCounter = 0
        }
        //listen on inputs from user
        checkUser()
        //Synchronize cells for drunk management
        for _, l := range cellChs {
            for _, c := range l {
                msg=<-c.In
            }
        }
        for _, l := range cellChs {
            for _, c := range l {
                c.Out<-Message{Tag: "syncDrunk"}
            }
        }
        //synchronize cells for report of automaton, before next step
        for _, l := range cellChs {
            for _, c := range l {
                msg=<-c.In
                if msg.Test && msg.Pedestrian != (Pedestrian{}) {
                    dir := getDirString(msg.Pedestrian.Dir)
                    fmt.Println("Pedestrian", msg.Pedestrian.Id, "is at position:", msg.Pos, "going"
                }
                if msg.Drunk {
                    fmt.Println("Drunk is at position:", msg.Pos)
                }
            }
        }
    }
    userInputHandler("exit") //we are done.
}

/*
    Hardcoded to create a minimum board, as described in assignment.
    Returns bichannels to injecters (end of sidewalk)
*/
```

```go
func setupMinSideWalk(length, width int) ([]Bichannel, [][]Bichannel) {
    var cellToInjectorChs [][]Bichannel
    numInjectors := 2
    injectorControlChs := make([]Bichannel, numInjectors)

    controlCellChs := make([][]Bichannel, length)
    for i := range controlCellChs {
        controlCellChs[i] = make([]Bichannel, width)
    }

    //make injectors, hardcoded to 2.
    injectorChs := make([]Bichannel, width)
    cellToInjectorChs = make([][]Bichannel, numInjectors)
    cellToInjectorChs[0] = make([]Bichannel, width)
    cellToInjectorChs[1] = make([]Bichannel, width)
    for j:= range cellToInjectorChs[0] {
        ch1, ch2 := makeNewBichannel()
        cellToInjectorChs[0][j] = ch1
        injectorChs[j] = ch2
    }
    ch1, ch2 := makeNewBichannel() //control channels
    go injector(injectorChs[:], ch1, E)
    injectorControlChs[0] = ch2

    injectorChs = make([]Bichannel, width)
    for j:= range cellToInjectorChs[1] {
        ch1, ch2 := makeNewBichannel()
        cellToInjectorChs[1][j] = ch1
        injectorChs[j] = ch2
    }
    ch1, ch2 = makeNewBichannel()
    go injector(injectorChs[:], ch1, W)
    injectorControlChs[1] = ch2

    //make cells
    westCellRowChs := make([]Bichannel, width)
    copy(westCellRowChs, cellToInjectorChs[0]) //TODO might be cellToInjectorChs[0][:]
    var southCh, northCh, eastCh, westCh Bichannel
    var neighbors [4]Bichannel
    for x:=0; x < length-1; x++ { //do all rows but the last
        for y:=0; y < width; y++ {
            neighbors[S] = southCh
            if y < width-1 {
                northCh, southCh = makeNewBichannel()
            } else {
                northCh = Bichannel{}
                southCh = Bichannel{}
            }
            neighbors[N] = northCh
            neighbors[W] = westCellRowChs[y]
            eastCh, westCh = makeNewBichannel()
            westCellRowChs[y] = westCh
            neighbors[E] = eastCh
            pos := Position{x,y}
            cellControlch1, ch2 := makeNewBichannel()
            controlCellChs[x][y] = ch2
            go cell(pos, neighbors, cellControlch1)
            neighbors = [4]Bichannel{}
        }
    }
    for y:=0; y < width; y++ { //do last row with special west channel to injector
        neighbors[S] = southCh
        if y < width-1 {
            northCh, southCh = makeNewBichannel()
        } else {
            northCh = Bichannel{}
            southCh = Bichannel{}
        }
        neighbors[N] = northCh
        neighbors[W] = westCellRowChs[y]
        neighbors[E] = cellToInjectorChs[1][y]
        pos := Position{length-1,y}
        cellControlch1, ch2 := makeNewBichannel()
        controlCellChs[length-1][y] = ch2
        go cell(pos, neighbors, cellControlch1)
    }

    return injectorControlChs[:], controlCellChs[:][:]
}

func main(){
    length := 10
    width := 5
    initInjectRate := 4
    maxTimeSteps := 200
    injectChs, controlCellChs := setupMinSideWalk(length, width)

    control(injectChs, controlCellChs, initInjectRate, maxTimeSteps)
}
```