Write Robotic Programs using StarL

Yixiao Lin

Agenda

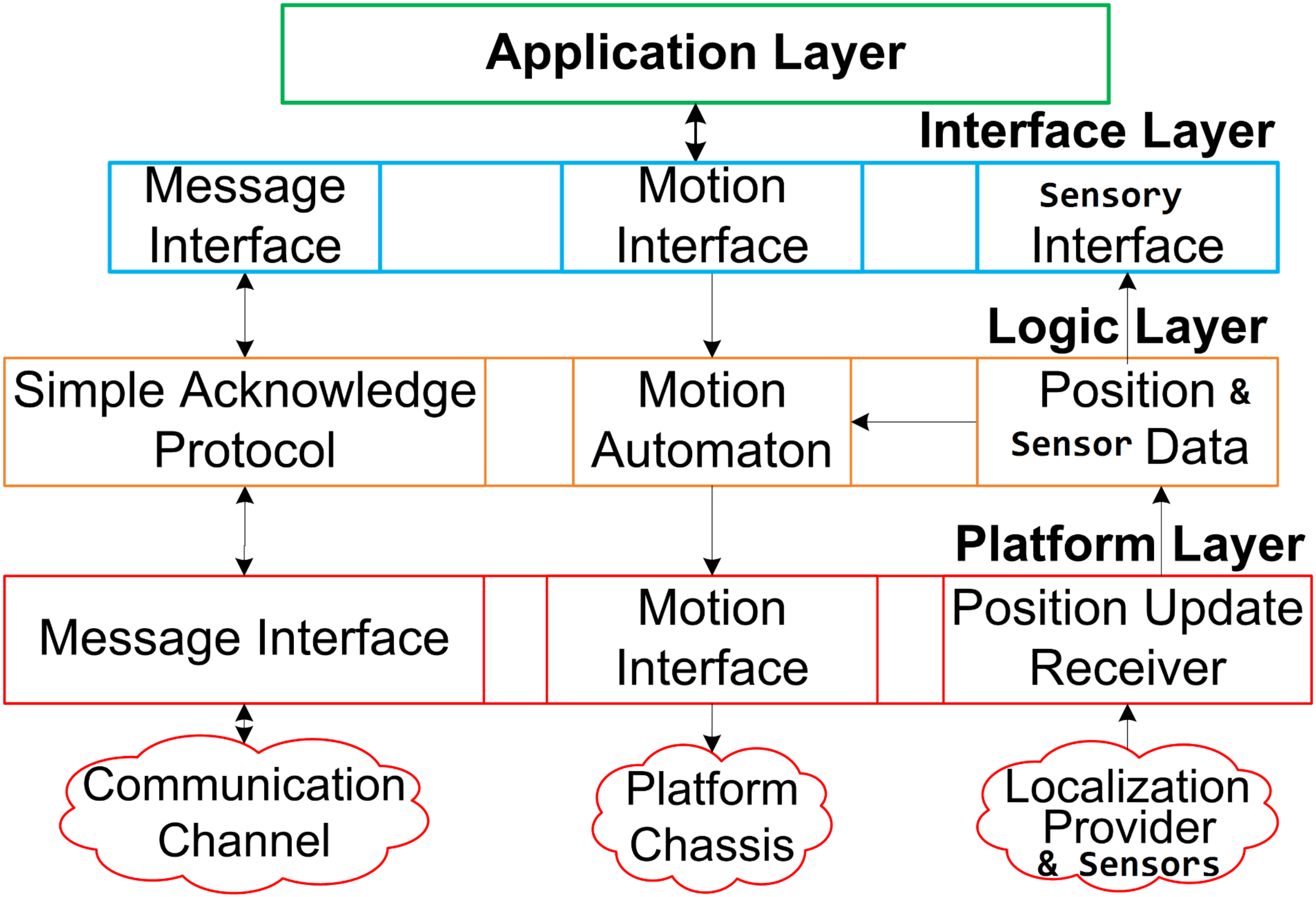
- Background on StarL
- Application structure in StarL
- Traffic Sign Application
- Algorithm and Implementation

What is StarL?

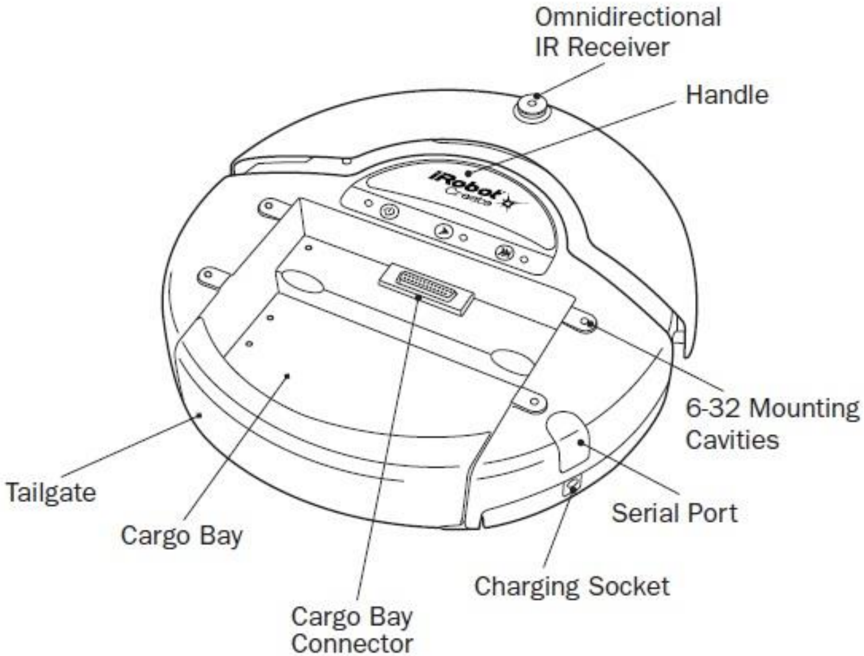
- Stabilizing Robotics Programming Language
- Java based platform developed for high level distributed robotic programming

StarL Components

- Common Library
  - Communication protocol
  - Motion control
  - Sensors
- Simulator
  - Engine
  - Visualizer
    - 2D
    - 3D


Application Layer
Interface Layer
Message Interface
Motion Interface
Sensory Interface
Logic Layer
Simple Acknowledge Protocol
Motion Automaton
Position & Sensor Data
Platform Layer
Message Interface
Motion Interface
Position Update Receiver
Communication Channel
Platform Chassis
Localization Provider & Sensors

iRobot Create



Create an Application

- Define states and variables
- Create state machine
  - Functions
  - Motion
  - Messages
  - Sensor information
- Handle communication

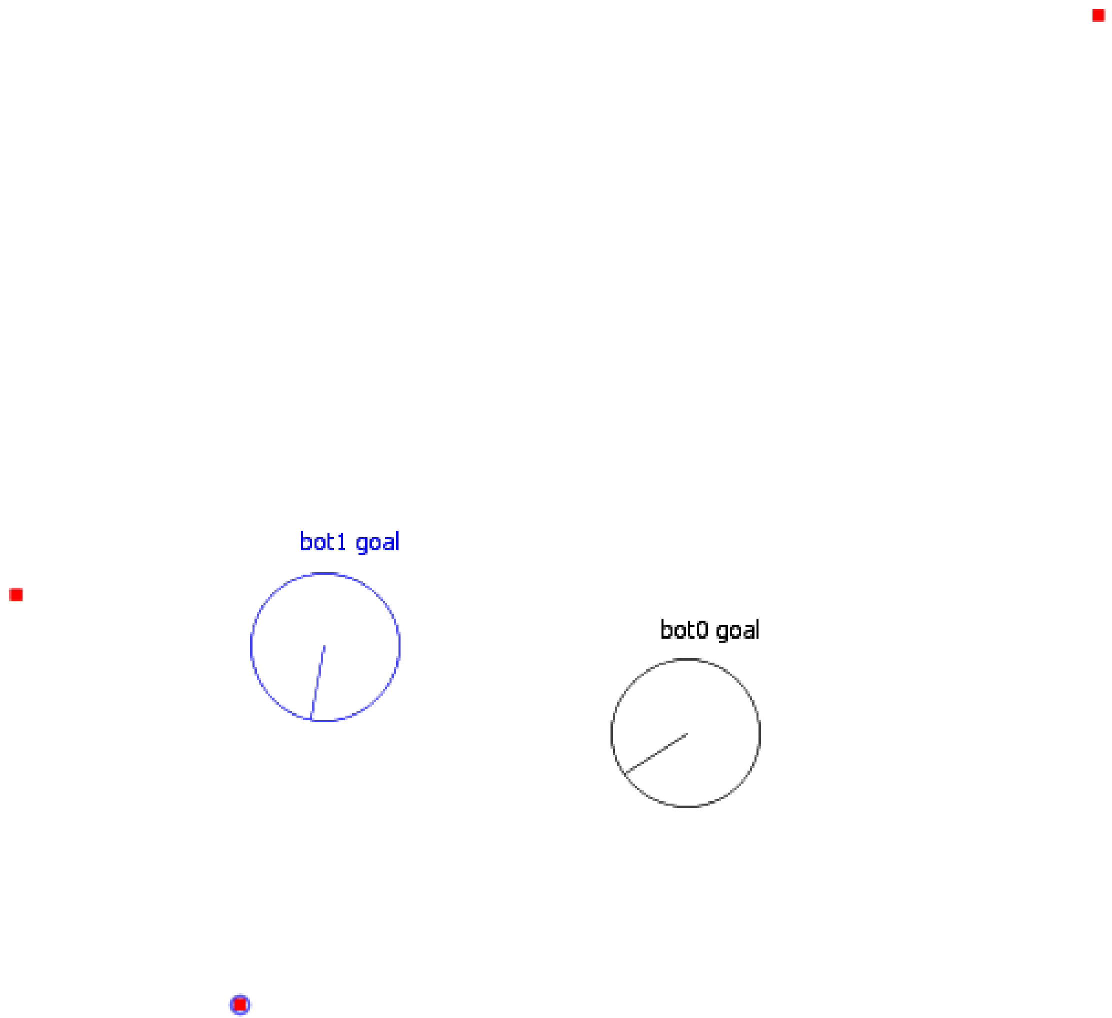
```java
public class RaceApp extends LogicThread {
    private static final boolean RANDOM_DESTINATION = false;
    public static final int ARRIVED_MSG = 22;

    final Map<String, ItemPosition> destinations = new HashMap<String, ItemPosition>();
    ItemPosition currentDestination;

    private enum Stage {
        PICK, GO, DONE
    };

    private Stage stage = Stage.PICK;

    public RaceApp(GlobalVarHolder gvh) {
        super(gvh);
        MotionParameters.Builder settings = new MotionParameters.Builder();
//      settings.ROBOT_RADIUS(400);
        settings.COLAVOID_MODE(COLAVOID_MODE_TYPE.USE_COLAVOID);
        MotionParameters param = settings.build();
        gvh.plat.moat.setParameters(param);
        for(ItemPosition i : gvh.gps.getWaypointPositions())
            destinations.put(i.getName(), i);
        gvh.comms.addMsgListener(this, ARRIVED_MSG);
    }
```

```java
    public List<Object> callStarL() {
        while(true) {
            switch(stage) {
            case PICK:
                if(destinations.isEmpty()) {
                    stage = Stage.DONE;
                } else {
                    currentDestination = getRandomElement(destinations);
                    gvh.plat.moat.goTo(currentDestination);
                    stage = Stage.GO;
                }
                break;
            case GO:
                if(!gvh.plat.moat.inMotion) {
                    if(currentDestination != null)
                        destinations.remove(currentDestination.getName());
                    RobotMessage inform = new RobotMessage("ALL", name, ARRIVED_MSG, currentDestination.getName());
                    gvh.comms.addOutgoingMessage(inform);
                    stage = Stage.PICK;
                }
                break;
            case DONE:
                return null;
            }
            sleep(100);
        }
    }
```

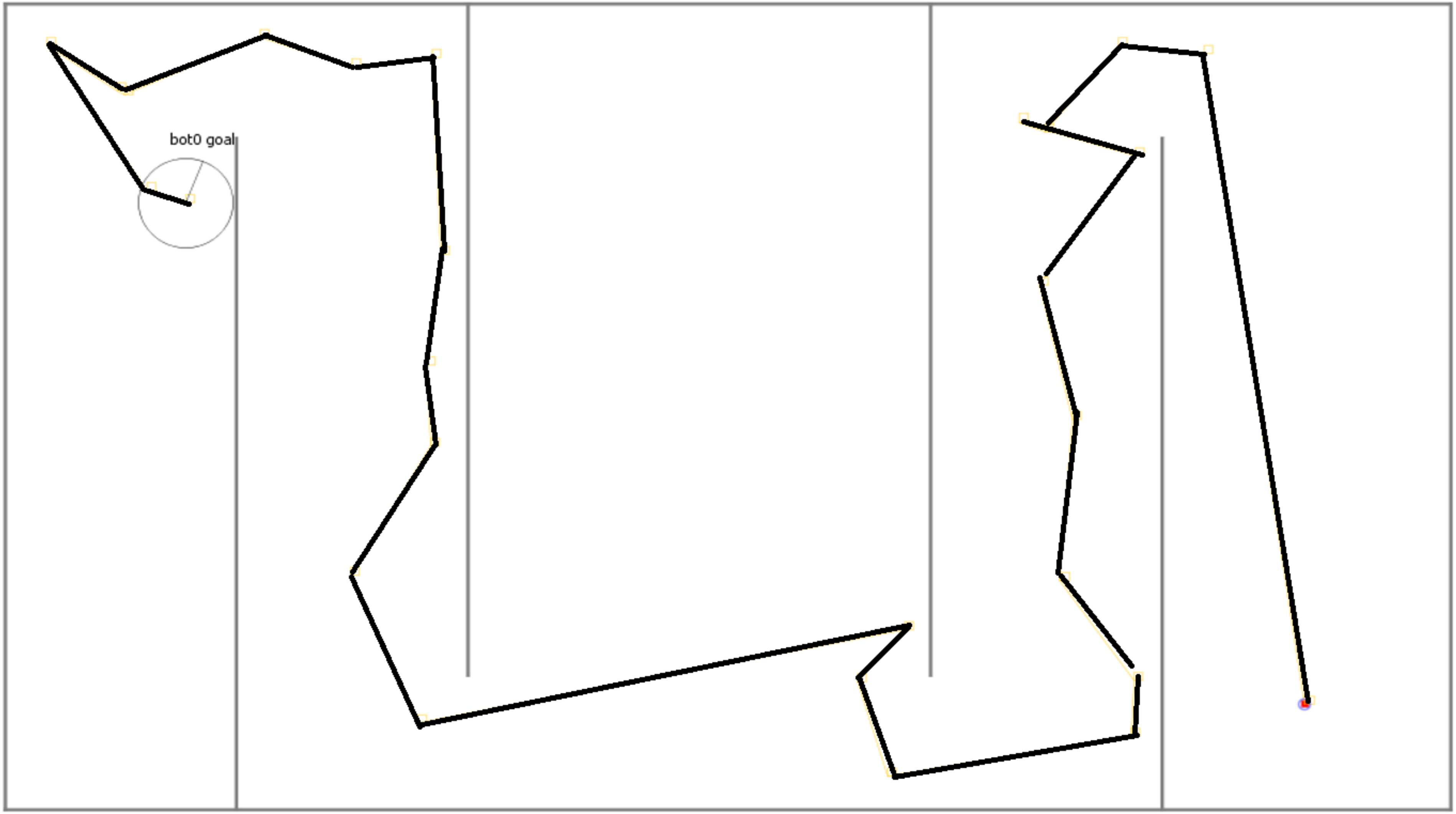
```java
    }

    @Override
    protected void receive(RobotMessage m) {
        String posName = m.getContents(0);
        if(destinations.containsKey(posName))
            destinations.remove(posName);

        if(currentDestination.getName().equals(posName)) {
            gvh.plat.moat.cancel();
            stage = Stage.PICK;
        }
    }

```

Simulate

```java
package edu.illinois.mitra.demo.race;

import edu.illinois.mitra.starlSim.main.SimSettings;

public class Main {

    public static void main(String[] args) {
        SimSettings.Builder settings = new SimSettings.Builder();

        settings.N_BOTS(2);
        settings.TIC_TIME_RATE(1.5);
        settings.WAYPOINT_FILE("waypoints/four.wpt");

        settings.DRAW_WAYPOINTS(false);
        settings.DRAW_WAYPOINT_NAMES(false);
        settings.DRAWER(new RaceDrawer());

        Simulation sim = new Simulation(RaceApp.class, settings.build());
        sim.start();
    }

}
```

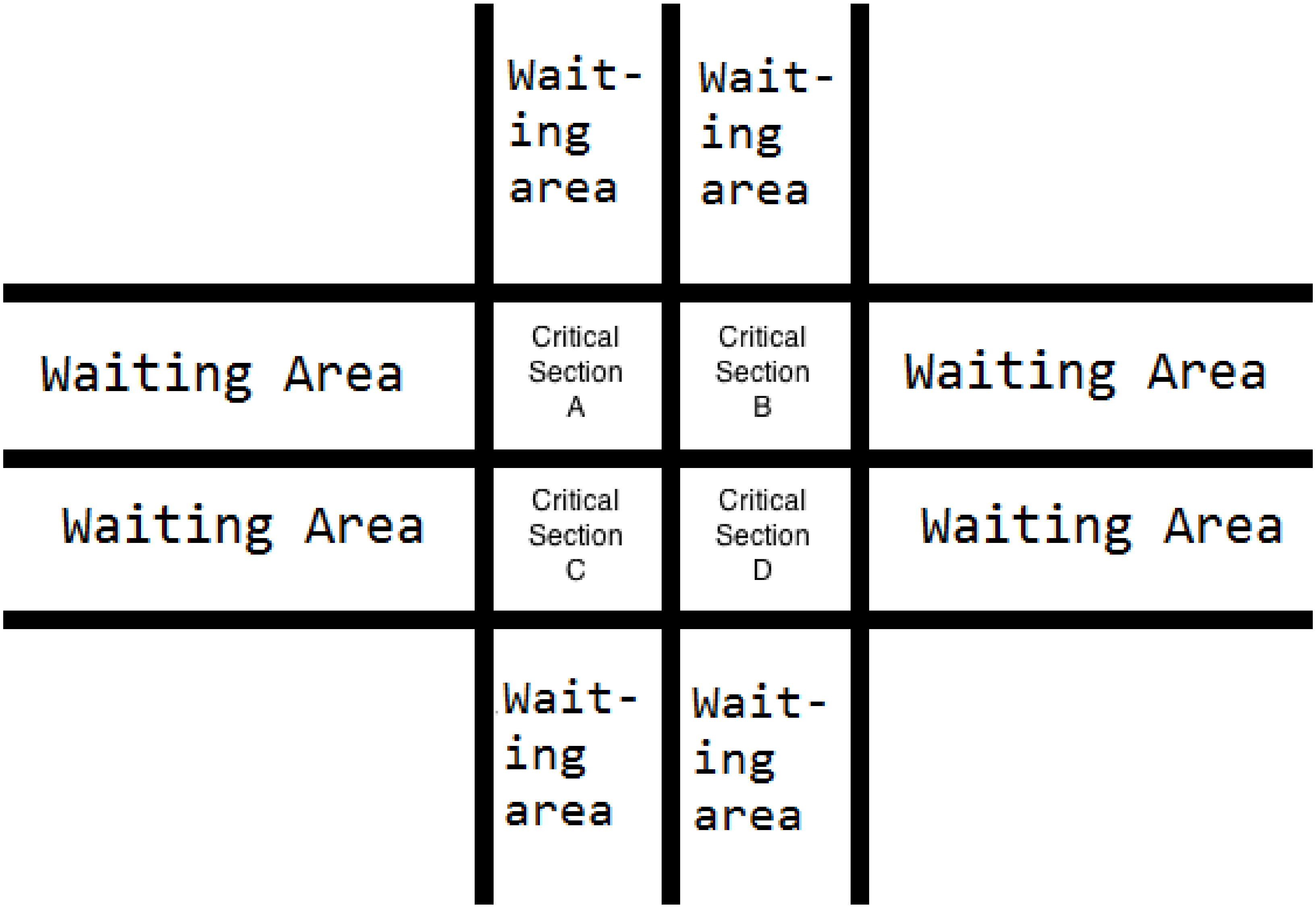

bot1 goal
bot0 goal

Form StarL



Stop Sign

Traffic Sign Application



Properties

- Safety: if some robot is in a critical section, then no other robots can be in the same critical section.
- Liveness: a robot interested in entering the critical section will enter the critical section will eventually succeed.
- Concurrent Entering: if robot-A is interesed in a path that does not intersect with robot-B's path, then robot-A and robot-B can enter the critical section concurrently.

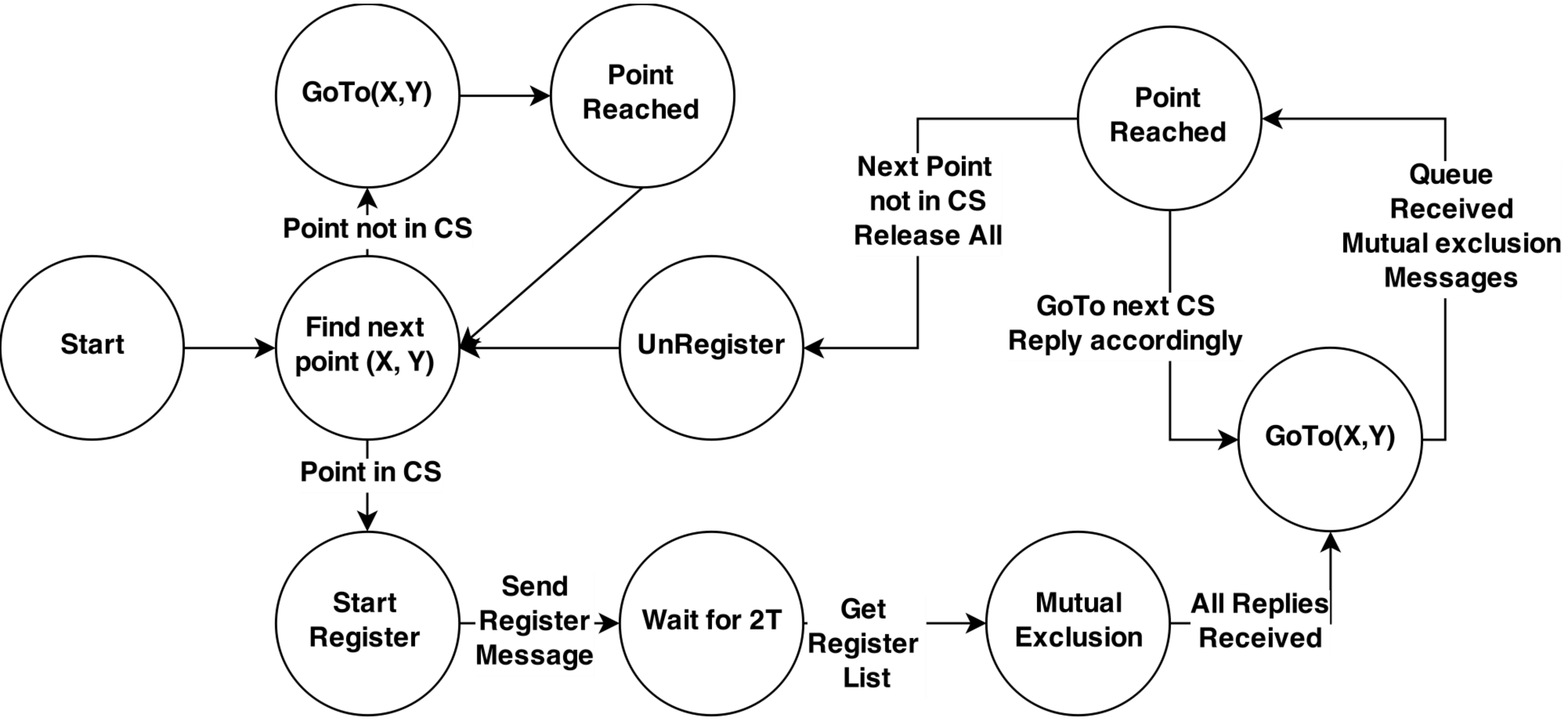
Algorithm 1: Mutual exclusion

```
On initialization:
   state := RELEASED;
To enter the section
   state := WANTED;
   sections := array of critical sections wanted
   T :=  timestamp obtained at registration;
   Multicast (sections, T, ID) to all interested robots;
   Wait until all interested robots reply;
   state := HELD;
On receipt of a request <sections_i, Ti, pi> at pj (i != j)
   if  (sections_j intersects sections_i and
        (state = HELD  or (state = WANTED and (T, pj) < (Ti, pi)))
   then
      queue request from pi without replying;
   else
      reply immediately to pi;
   end if
To exit a critical section s1
   sections -= s1;
   if(sections == empty)
      state := RELEASED;
      reply to any queued requests;
   else
      check queued requests and reply accordingly
```

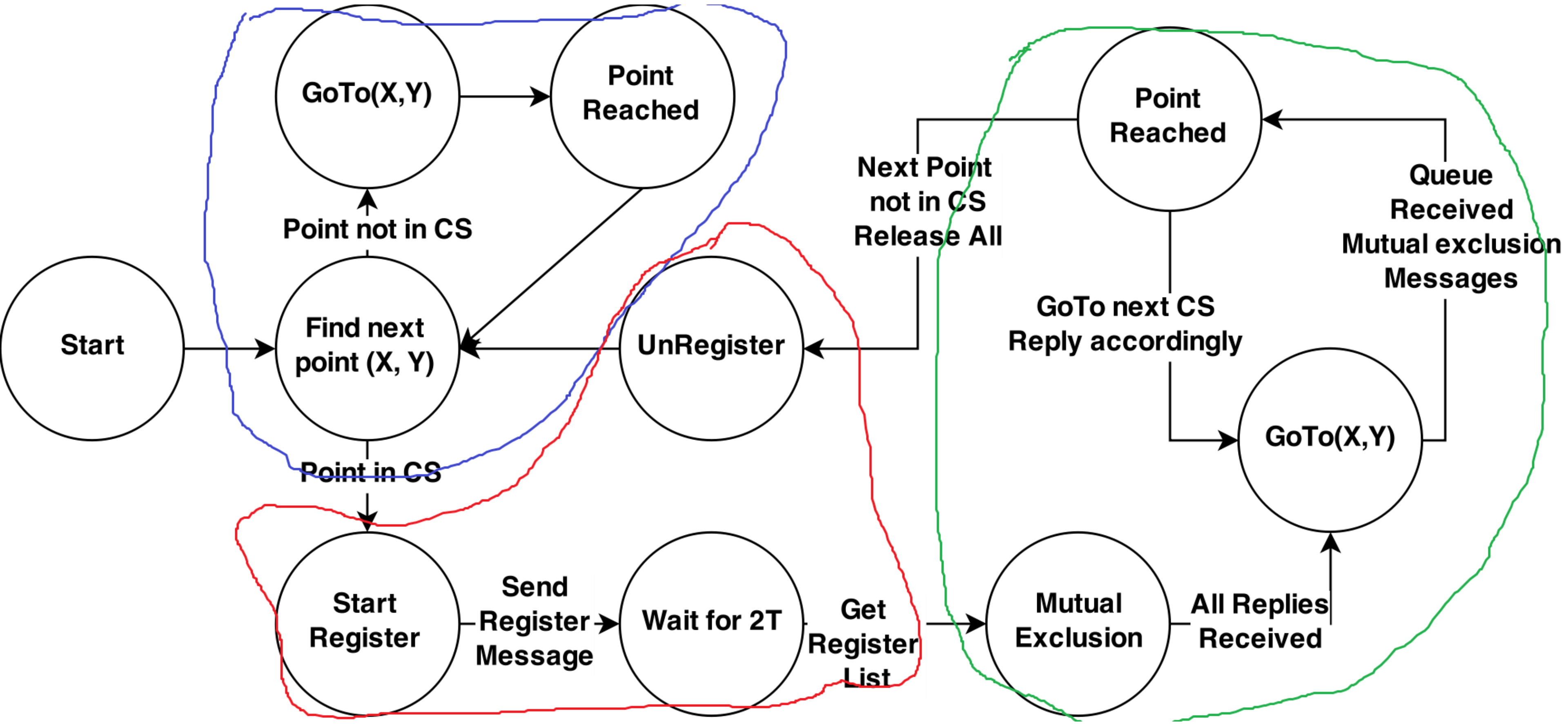
Algorithm2: Registration

```
Register:
   start listening;
   state := Register;
   Geocast(Register, ID);
   wait for 2*T;
   Construct RegisterList using queued messages;
      Order the list using the timestamp,
      if the timestamp is null, order message according to ID
   timeStamp = position in RegisterList + maximum time stamp from reply;
   state := Registered;
UnRegister:
   Geocast(UnRegister, ID);
On Receipt(Register message):
   if(state = Register)
      reply(ID, null);
   if(state = Registered)
      reply(ID, timeStamp);
   end if;
On Receipt(reply message):
   put message in queue;
On Receipt(UnRegister message)
   if(state = Register)
      remove all queued message from sender;
   if(state = Registered)
      remove sender from RegisterList;
```

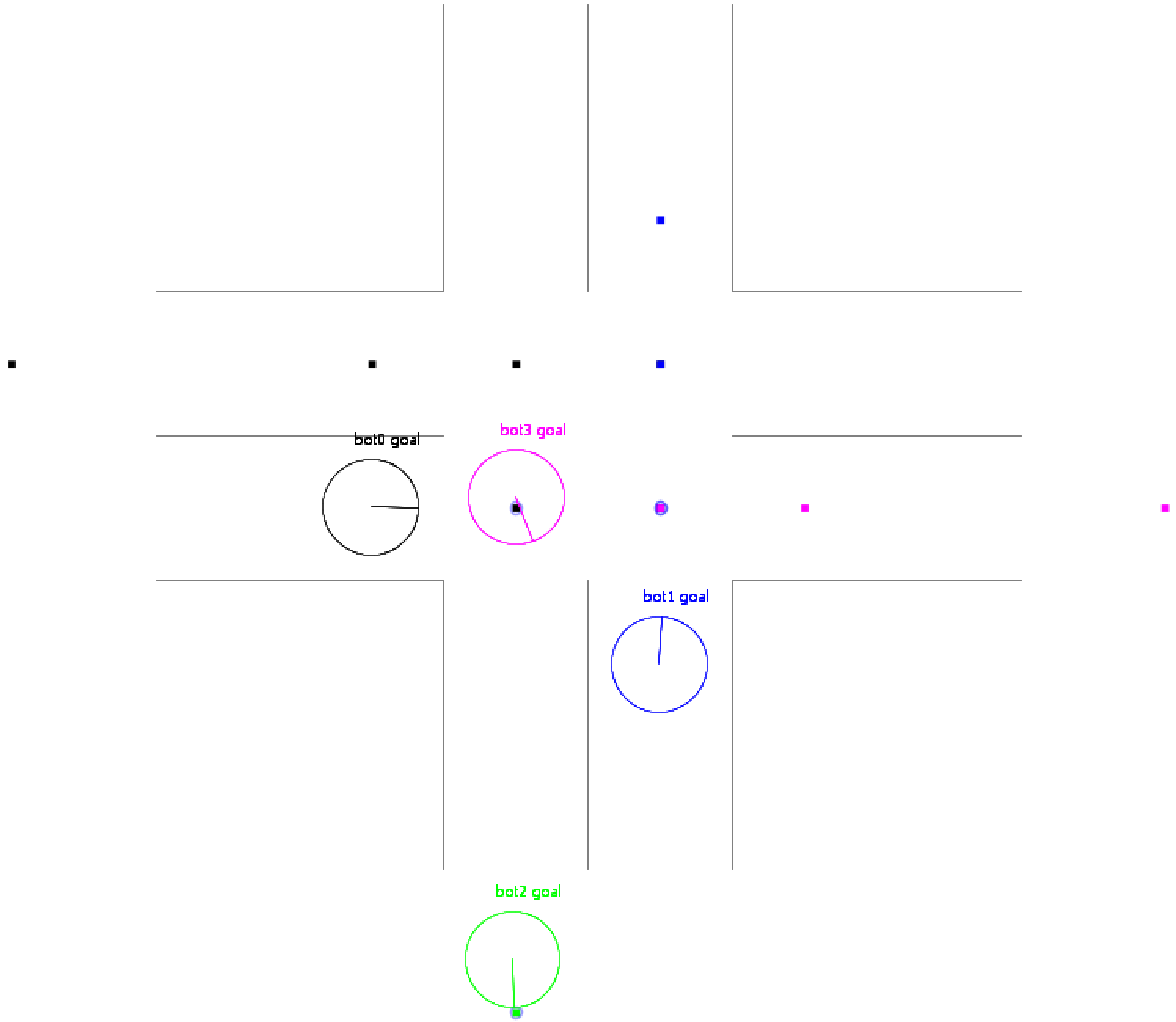
State Machine



State Machine



Simulation



Guarantees

- Safety:
  - Assume that robot A and robot B are both in the critical section S, they must both gained access to S.
  - Assume A gained access before B.
  - B must have received A's reply to gain the access. However, according to the algorithm, robot A would have queued B's request since A is in state HELD. Therefore, there is a contradiction, A and B can not both have access to S.

Guarantees

- Liveness:
  - It is ensured because a robot will get all it's required critical sections before proceeding to it's first critical section. Therefore, there would be no deadlocks.
  - A robot A would not wait forever because the timestamps generated for newly registered robots are bigger than A's timestamp.

Guarantees

- Concurrent Entering: Consider only two robots, if their path does not intersect, they will both reply to each other according the algorithm. Therefore, both would gained access and enter different critical sections concurrently.

Hybrid Automata

1. **Automaton**Mutual Exclusion(RListOfCars, wantedSections, ID, timeStamp)

    state: $\{REQUEST, WAIT, ENTRY, CS, EXIT\} := REQUEST$

    variables: ListOfCars := RListOfCars, Sections := wantedSections

    curpoint:= Sections.pop, MessageQueue := $\emptyset$;

    signature input, output input: message m1

    output: message m2, goTo(point)

    transitions:

    request

    pre state = REQUEST

    eff m2 :=(Sections, ListOfCars, ID, timeStamp) , state := WAIT

    enter

    pre state = WAIT $\cap$ ListOfCars = $\emptyset$;

    eff goTo(curpoint), state := ENTRY;

    nextpoint

    pre state = ENTRY $\cap$ goTo(curpoint) is finished

    eff curpoint := Sections.pop, check queue

Hybrid Automata

check queue

pre $\exists$message m$\in$MessageQueue, (Sections $\cup$ curpoint) $\cap$ m.Sections = $\emptyset$;

eff m2 :=("reply", m.from, ID);

release all

pre curpoint = NULL

eff state:=EXIT,$\forall$ message m$\in$ MessageQueue, m2 :=("reply", m.from, ID);

receive

pre (m1.Sections$\cap$Sections=$\emptyset$)$\cup$ (m1.timeStamp, m1.from)<(timeStamp, ID)

eff m2 :=("reply", m1.from, ID); otherwise, put m1 to messageQueue

trajectories:

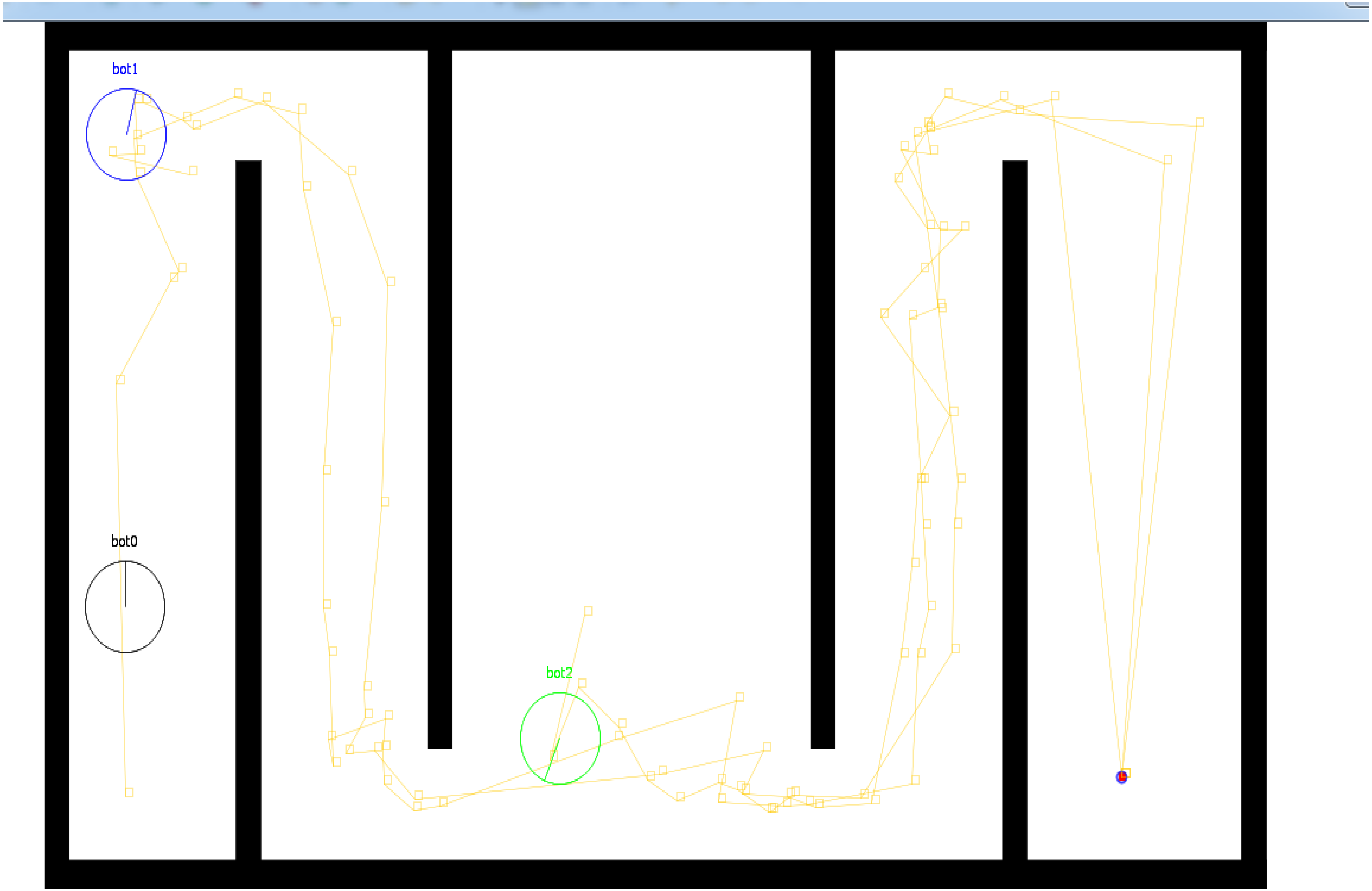
evolve:

invariant: Bot always go along the section directions

No more than one bot ever be in the same critical section

Other Applications

- Race
- Maze
- Light Painting
- Distributed Search


bot1
bot0
bot2

Acknowledgements

- Sayan Mitra
- Adam Zimmerman
- Chengyang Wang
- Adel Ahmadyan
- Taylor Johnson
- All the other previous members of the project

Thank you !

- Any Questions?