RoboCup Rescue Simulator and Agent Development Framework Manual

October 15, 2018

Contents

1 Introduction

A *disaster* may be defined as a crisis situation causing widespread human, material, economical and environmental damages that surpasses the system's capacity to recover. There are two main ways of treating a disaster:

1. Preventing it from happening, which can be done through identifying its causes and preventing them from taking place.
2. Managing its impacts once they happen through the application of previously established governmental policies and strategies.

The former option is usually more desirable, since it is very effective and very little disrupting. However, this is not always possible, since some events are too intense to be avoided even if forecasted with great antecedence, and others are unpredictable. Thus, the management option becomes necessary for the mitigation of the disaster's impacts.

After the 1995 Kobe earthquake events, Japanese researchers came up with the idea of developing a simulator that reproduces conditions similar to a urban post-earthquake in order to test mitigation strategies for this kind of disaster. They also organized a competition, the RoboCup Rescue Agent Simulation League, in which people from all over the world would be able to demonstrate their advancements in the coordination of rescuing agents teams running in this simulator and to help in the development of policies and strategies to mitigate the impact of such disasters.

The purpose of this manual is to facilitate the understanding of the first contact with the RoboCup Rescue Simulation server and to help people interested in participating in RoboCup Rescue Agent Simulation competitions.

2 Installation

This manual assumes the simulator and agents will run in a Linux machine even though it is possible to run them in Microsoft Windows or Apple macOS. We recommend to use Linux because it is open-source and most of the distributions have a good support from the users' community. If you have never used Linux before and intend to, we recommend starting with a user-friendly distribution, such as Ubuntu[1] or Fedora[2].

2.1 Software Requirements

- Java OpenJDK 8+[3]
- Git
- Gradle
- Utilities like *wget*, *bash*, *xterm*, *tar*, *gzip*, etc.
  If you are using Ubuntu, all of these software are present in the default software repositories.

2.2 Installing RoboCup Rescue Simulation (RCRS) Server

1. Clone the simulation server from `https://github.com/roborescue/rcrs-server`.
2. Change to the directory "rcrs-server".
   (a) If you use macOS, patch the file "boot/functions.sh" like

   "`sed -i -e "/readlink/s/^/#/" boot/functions.sh`".
3. Compile the simulator using the commands `gradle clean` and `gradle completeBuild`.
4. Check the message at the end of the installation. If the installation is successfully completed, you get the message `BUILD SUCCESSFUL`; otherwise you get `BUILD FAILED`.

If you are using Ubuntu, the installation proceeds according to the commands below:

[1]`https://www.ubuntu.com/`
[2]`https://getfedora.org`
[3]`https://openjdk.java.net/`

Installation on Ubuntu

```
$ git clone https://github.com/roborescue/rcrs-server.git
$ cd rcrs-server
($ sed -i -e "/readlink/s/^/#/" boot/functions.sh # Only required for macOS)
$ gradle clean
$ gradle completeBuild
```

The following message will be appeared if the installation is successfully completed.

Install Completion

```
BUILD SUCCESSFUL in 2s
1 actionable task:  1 executed
```

2.3 Compiling the Agent Development Framework (ADF) Sample Agents

Download the sample agents with ADF by cloning the `https://github.com/roborescue/rcrs-adf-sample.git` repository. Then, you move to `rcrs-adf-sample` directory and compile the sample agents using the script `compile.sh`.

If you are using Ubuntu, you can get and compile the ADF with the following commands:

Download ADF on Ubuntu

```
$ git clone https://github.com/roborescue/rcrs-adf-sample.git
$ cd rcrs-adf-sample
$ ./compile.sh
```

3 Running the ADF Sample Agents on RCRS Server

3.1 Running without Precomputation

To run the sample agents, you must open two terminal windows. One is used to run the simulation server (i.e., the simulator) and the other is used to run the agents.

3.1.1 Running the Simulation Server

Use one terminal window and move to the boot directory inside the simulator's folder (`rcrs-server`). Then, type `bash start-comprun.sh`. The sequence of commands are:

Running Simulation Server

```
$ cd rcrs-server
$ cd boot
$ bash start-comprun.sh
```

When the simulation server runs correctly, the window in Fig. 1 will appear.

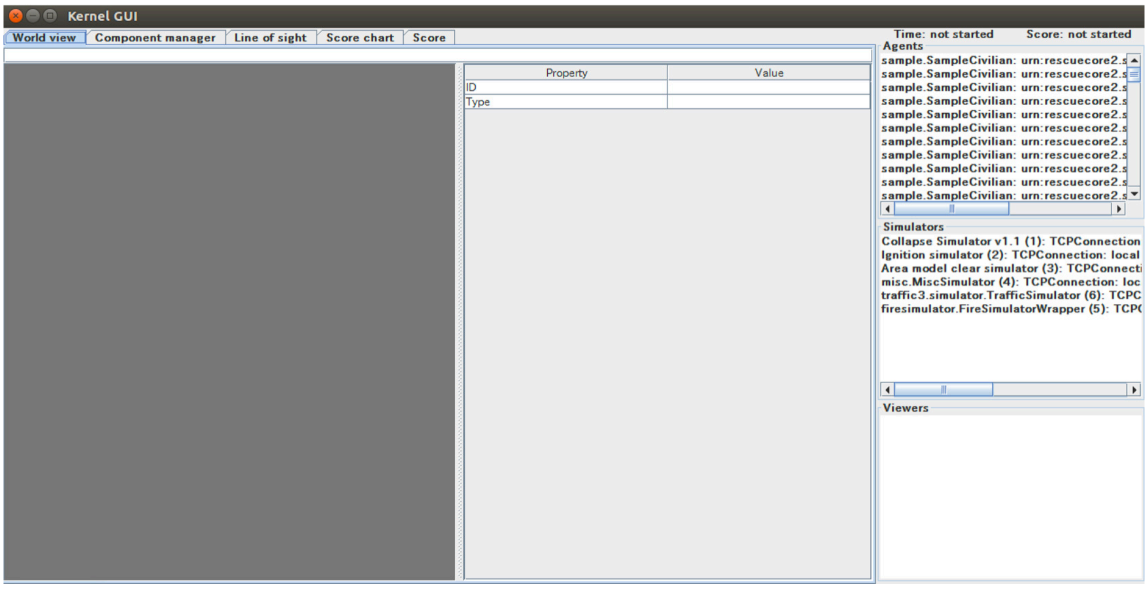
3.1.2 Running the ADF Sample Agents

After running the simulation server, move to `rcrs-adf-sample` directory on the other terminal window and run the agents by the following commands:

Running Sample Agents

```
$ sh launch.sh -all
[FINISH] Done connecting to server (3 agents)
```

If the agents can connect with the simulator, the state of the agents and a city are shown on the left-hand side in the window shown in Fig. 1. Then, the simulation is started as shown in Fig. 2.



```
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
```

```
Collapse Simulator v1.1 (1): TCPConnection
Ignition simulator (2): TCPConnection: local
Area model clear simulator (3): TCPConnecti
misc.MiscSimulator (4): TCPConnection: loc
traffic3.simulator.TrafficSimulator (6): TCPC
firesimulator.FireSimulatorWrapper (5): TCP(
```

Figure 1: Running the Simulation Server

3.2 Running with Precomputation

Agents can examine a simulation scenario by performing a precomputation of it before starting the simulation. The length of the precomputation is predefined to 2 minutes in the competition. The precomputation needs two terminal windows likewise.

3.2.1 Running the Simulation Server for Precomputation

Use one terminal window and move to the boot directory inside the simulator's folder (`rcrs-server`). Then, type `bash start-precompute.sh`. These commands are:

Running Simulation Server

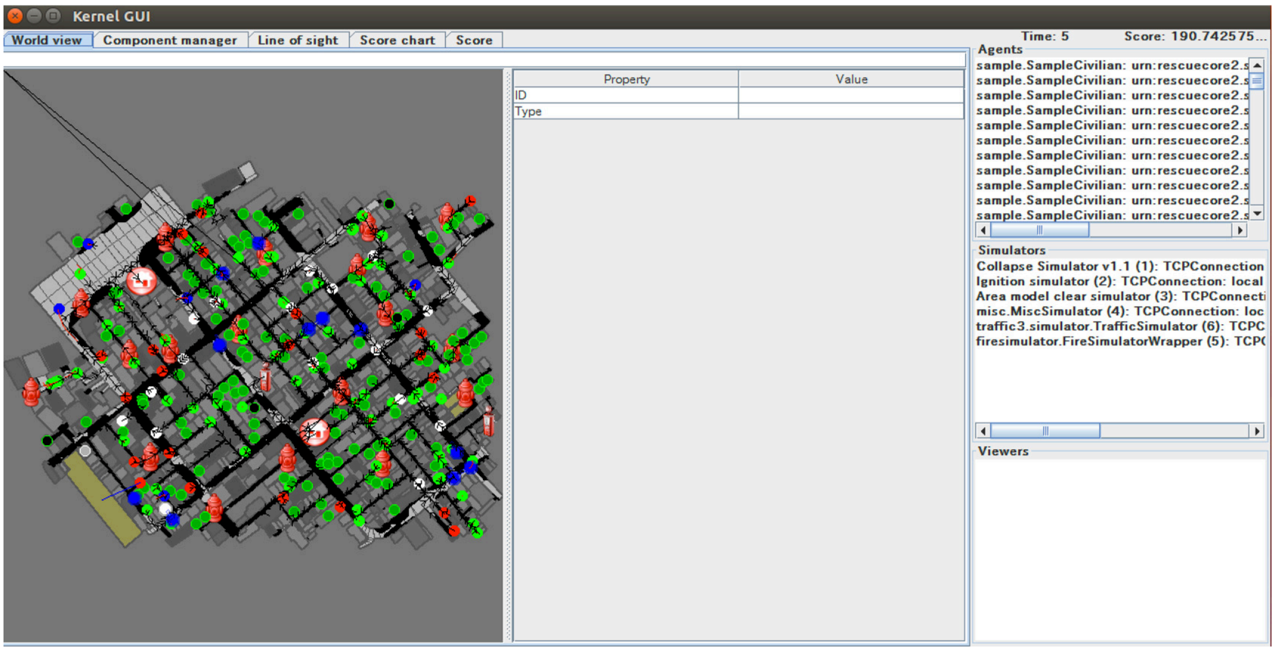
```
$ cd rcrs-server
$ cd boot
$ bash start-precompute.sh
```

3.2.2 Running the ADF Sample Agents for Precomputation

After running the simulation server for the precomputation, move to the ADF directory on the other terminal window and run the agents executing the commands:

Running Sample Agents

```
$ sh launch.sh -t 1,0,1,0,1,0 -h localhost -pre 1 & APID=$!  ; sleep 120 ; kill
$APID
[START ] Connect to server (host:localhost, port:7000)
[INFO ] Connected - adf.agent.platoon.PlatoonFire@756ec19c (PRECOMPUTATION)
[INFO ] Connected - adf.agent.platoon.PlatoonPolice@366bbbe (PRECOMPUTATION)
[INFO ] Connected - adf.agent.platoon.PlatoonAmbulance@2a453513 (PRECOMPUTATION)
*******************
[FINISH] Connect PoliceForce (success:1)
[FINISH] Connect AmbulanceTeam (success:1)
[FINISH] Connect FireBrigade (success:1)
[FINISH] Done connecting to server (3 agents)
```



```
Time: 5        Score: 190.742575...
Agents
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
sample.SampleCivilian: urn:rescuecore2.s
Simulators
Collapse Simulator v1.1 (1): TCPConnection
Ignition simulator (2): TCPConnection: local
Area model clear simulator (3): TCPConnecti
misc.MiscSimulator (4): TCPConnection: loc
traffic3.simulator.TrafficSimulator (6): TCPC
firesimulator.FireSimulatorWrapper (5): TCP(
Viewers
```

Figure 2: Starting the Simulation

3.2.3 Running the Simulation

When the precomputation is completed, push `Control-C` and type `sh kill.sh` to stop the simulation server of running. Then, type `bash start-comprun.sh` to start the simulation server again to run the simulation scenario. The commands are:

Running Simulation Server

```
Control-C
$ sh kill.sh
$ bash start-comprun.sh
```

3.2.4 Running the ADF Sample Agents

After running the simulation server, move to the ADF directory on the other terminal window and run the agents using the commands:

Running Sample Agents

```
$ sh lauch.sh -all
[FINISH] Done connecting to server (3 agents)
```

4 Simulation Server Options

4.1 Directories

Important directories of the simulation server are:

- `boot/`: scripts to run the simulation server
  - `boot/config/`: configuration files of the simulation server.
  - `boot/logs/`: log files.
- `build/`: the simulation server's Java classes.

- `jars/`: the simulation server's JAR files.
- `lib/`: libraries used by the simulation server.
- `maps/`: maps that can be ran in the simulation server.
- `modules/`: the simulation server's source code.
- `oldsims/`: source code of some of the simulation server's older versions.

4.2 Parameters

The following parameters can be used to run the simulation server:

- -m MAPDIR or --map MAPDIR, where MAPDIR is the path to the directory containing the map you want to run (default is ../maps/gml/Kobe2013/map).
- -c CONFIGDIR or --config CONFIGDIR, where CONFIGDIR is the directory containing the configuration associated with a map (default is ./config).
- -l LOGDIR or --log LOGDIR, where LOGDIR is the directory where the log files will be stored (default is ./logs).

These parameters can be used at running a precomputaion and a simulation. You must use the same parameters regarding MAPDIR and CONFIGDIR to run a simulation server with or without precomputation. An example of how to run the simulation server using these parameters is:

Running Simulation Server with Options

```
$ bash start-precompute.sh -m ../maps/gml/berlin -l logs2
(After completing agents' precomputation)
Control-C
$ sh kill.sh
$ bash start-comprun.sh -m ../maps/gml/berlin -l logs2
```

5 How to Create Your Own Agents with ADF

This section explain how to implement your agents using ADF samples.

5.1 Important Directories

Important directories of ADF (`rcrs-adf-sample`) are:

- `config/`: configuration file of agents.
- `src/`: agents' source codes.
- `precomp_data/`: results of a precomputation for each type of agents.
- `build/`: agents' Java classes.
- `library/`: libraries used by agents.

5.2 Files to Create Your Agents

You can develop your own agents codes using only the files in the directories:

- `src/adf/sample/centralized`: source codes for *central agents*. This is the type of agents whose only interaction with the world is through radio communication. There are three types of central agents: *Ambulance Centers*, *Fire Stations* and *Police Office*, and they are represented as buildings in the simulation server.
- `src/adf/sample/extraction`: codes of combining actions described in the directory below.
- `src/adf/sample/module`: concrete codes of algorithms, e.g. path planning, clustering, target detection, etc. The directory contains two directories:

– `src/adf/sample/module/algorithm`
– `src/adf/sample/module/complex`

Note

You must not make any changes of files in `src/adf/sample/tactics`. This is the restriction for our current competition rule.

You should fundamentally copy the sample codes, not edit them. The reason is that the sample codes would be used if ADF could not find your own codes. You can easily change reference to your modules by modifying `src/adf/config/module.cfg`. The usage of the file is described below.

5.3 Work Flow of Coding Your Agents

The steps necessary to code your own agents are:

1. Copy sample codes related to agents which you want to create,
2. Edit the copied files.
3. Edit `src/adf/config/module.cfg` according to the edited files.
4. Compile and run.

5.4 Modules Configuration File

The modules configuration file `src/adf/config/module.cfg` indicates which codes would be used as agents' module. Fig. 3 shows part of the modules configuration file. The left-hand side of the colon indicates the module name, the right-hand side is the class name. In most cases, modules of which targets' problems are the same should refer to an identical class for all agent types. The example in Fig. 3 is in `TacticsAmbulanceTeam.Search` and `TacticsFireBrigade.Search` indicates that both modules refer to `adf.sample.module.complex.SampleSearch`. An usage example is shown in Section 5.5.3.

```
TacticsAmbulanceTeam.HumanDetector :  adf.sample.module.complex.SampleHumanDetector
TacticsAmbulanceTeam.Search :  adf.sample.module.complex.SampleSearch

TacticsAmbulanceTeam.ActionTransport :  adf.sample.extaction.ActionTransport
TacticsAmbulanceTeam.ActionExtMove :  adf.sample.extaction.ActionExtMove

TacticsAmbulanceTeam.CommandExecutorAmbulance :  adf.sample.centralized.CommandExecutorAmbulance
TacticsAmbulanceTeam.CommandExecutorScout :  adf.sample.centralized.CommandExecutorScout

TacticsFireBrigade.BuildingDetector :  adf.sample.module.complex.SampleBuildingDetector
TacticsFireBrigade.Search :  adf.sample.module.complex.SampleSearch

TacticsFireBrigade.ActionFireFighting :  adf.sample.extaction.ActionFireFighting
TacticsFireBrigade.ActionExtMove :  adf.sample.extaction.ActionExtMove
⋮
```

Figure 3: Modules Configuration File

5.5 Example of Implementing A* Algorithm for Path Planning Algorithm

5.5.1 Copy A Sample Code

First of all, you should copy the sample code for path planning which is `SamplePathPlanning.java`. The example is described below. Note that the second command is split into two lines because of space limitations, but it should be entered as a single line.

Copy the Sample Path Planning

```
$ mkdir -p src/myteam/module/algorithm
$ cp src/adf/sample/module/algorithm/SamplePathPlanning.java
src/myteam/module/algorithm/AStarPathPlanning.java
```

5.5.2 Editing the Sample Code

Listing 1 is the code of `SamplePathPlanning.java`, which has Dijkstra's algorithm. You should edit 1st line, 18th line and 27th line (these lines are indicated with red comments). You would implement your own code in the method `calc()`, and remove the method `isGoal()` that is only used by `calc()`. Listing 2 shows the results of editing these lines.

You must implement the method `calc()` to get its calculation result by the method `getResult()`. The type of `getResult()` returning is `List<EntityID>`.

Listing 3 indicates the contents of the method `calc()`. In addition you should write the new private class `Node` which is used by the method `calc()`. The code is shown in listing 4. It must be put in the file `AStarPathPlanning.java`.

Listing 1: SamplePathPlanning.java

```
1  package adf.sample.module.algorithm; // Edit this line
2  
3  import adf.agent.communication.MessageManager;
4  import adf.agent.develop.DevelopData;
5  import adf.agent.info.AgentInfo;
6  import adf.agent.info.ScenarioInfo;
7  import adf.agent.info.WorldInfo;
8  import adf.agent.module.ModuleManager;
9  import adf.agent.precompute.PrecomputeData;
10 import adf.component.module.algorithm.PathPlanning;
11 import rescuecore2.misc.collections.LazyMap;
12 import rescuecore2.standard.entities.Area;
13 import rescuecore2.worldmodel.Entity;
14 import rescuecore2.worldmodel.EntityID;
15 
16 import java.util.*;
17 
18 public class SamplePathPlanning extends PathPlanning { // Edit this line
19 
20     private Map<EntityID, Set<EntityID>> graph;
21 
22     private EntityID from;
23     private Collection<EntityID> targets;
24     private List<EntityID> result;
25     // Edit the following line
26     public SamplePathPlanning(AgentInfo ai, WorldInfo wi, ScenarioInfo si, ModuleManager moduleManager, DevelopData developData) {
27         super(ai, wi, si, moduleManager, developData);
28         this.init();
29     }
30 
31     private void init() {
32         Map<EntityID, Set<EntityID>> neighbours = new LazyMap<EntityID, Set<EntityID>>() {
33             @Override
34             public Set<EntityID> createValue() {
35                 return new HashSet<>();
36             }
37         };
38         for (Entity next : this.worldInfo) {
39             if (next instanceof Area) {
40                 Collection<EntityID> areaNeighbours = ((Area) next).getNeighbours();
41                 neighbours.get(next.getID()).addAll(areaNeighbours);
42             }
43         }
44         this.graph = neighbours;
45     }
46 
47     @Override
48     public List<EntityID> getResult() {
49         return this.result;
50     }
51 
52     @Override
53     public PathPlanning setFrom(EntityID id) {
54         this.from = id;
55         return this;
56     }
57 
58     @Override
59     public PathPlanning setDestination(Collection<EntityID> targets) {
60         this.targets = targets;
61         return this;
```

```java
    }

    @Override
    public PathPlanning updateInfo(MessageManager messageManager) {
        super.updateInfo(messageManager);
        return this;
    }

    @Override
    public PathPlanning precompute(PrecomputeData precomputeData) {
        super.precompute(precomputeData);
        return this;
    }

    @Override
    public PathPlanning resume(PrecomputeData precomputeData) {
        super.resume(precomputeData);
        return this;
    }

    @Override
    public PathPlanning preparate() {
        super.preparate();
        return this;
    }

    @Override
    public PathPlanning calc() {  // Renew this method (implement your algrithm here)
        List<EntityID> open = new LinkedList<>();
        Map<EntityID, EntityID> ancestors = new HashMap<>();
        open.add(this.from);
        EntityID next;
        boolean found = false;
        ancestors.put(this.from, this.from);
        do {
            next = open.remove(0);
            if (isGoal(next, targets)) {
                found = true;
                break;
            }
            Collection<EntityID> neighbours = graph.get(next);
            if (neighbours.isEmpty()) {
                continue;
            }
            for (EntityID neighbour : neighbours) {
                if (isGoal(neighbour, targets)) {
                    ancestors.put(neighbour, next);
                    next = neighbour;
                    found = true;
                    break;
                }
                else {
                    if (!ancestors.containsKey(neighbour)) {
                        open.add(neighbour);
                        ancestors.put(neighbour, next);
                    }
                }
            }
        } while (!found && !open.isEmpty());
        if (!found) {
            // No path
            this.result = null;
        }
        // Walk back from goal to this.from
        EntityID current = next;
        LinkedList<EntityID> path = new LinkedList<>();
        do {
            path.add(0, current);
            current = ancestors.get(current);
            if (current == null) {
                throw new RuntimeException("Found a node with no ancestor! Something is broken.");
            }
        } while (current != this.from);
        this.result = path;
        return this;
    }
    // Remove the method (it is only used by calc()).
    private boolean isGoal(EntityID e, Collection<EntityID> test) {
        return test.contains(e);
    }
}
```

Listing 2: Part of AStarPlanning.java

```java
package myteam.module.algorithm; // Position of the file

import adf.agent.communication.MessageManager;
import adf.agent.develop.DevelopData;
import adf.agent.info.AgentInfo;
import adf.agent.info.ScenarioInfo;
import adf.agent.info.WorldInfo;
import adf.agent.module.ModuleManager;
import adf.agent.precompute.PrecomputeData;
import adf.component.module.algorithm.PathPlanning;
import rescuecore2.misc.collections.LazyMap;
import rescuecore2.standard.entities.Area;
import rescuecore2.worldmodel.Entity;
import rescuecore2.worldmodel.EntityID;

import java.util.*;

public class AStarPathPlanning extends PathPlanning { // Same as the file name

    private Map<EntityID, Set<EntityID>> graph;

    private EntityID from;
    private Collection<EntityID> targets;
    private List<EntityID> result;
    // Same as the file name
    public AStarPathPlanning(AgentInfo ai, WorldInfo wi, ScenarioInfo si, ModuleManager moduleManager, DevelopData developData) {
        super(ai, wi, si, moduleManager, developData);
        this.init();
    }
```

Listing 3: calc()

```java
    @Override
    public PathPlanning calc() {
        List<EntityID> open = new LinkedList<>();
        List<EntityID> close = new LinkedList<>();
        Map<EntityID, Node> nodeMap = new HashMap<>();
        open.add(this.from);
        nodeMap.put(this.from, new Node(null, this.from));
        close.clear();

        while (true) {
            if (open.size() < 0) {
                this.result = null;
                return this;
            }
            Node n = null;
            for (EntityID id : open) {
                Node node = nodeMap.get(id);
                if (n == null) {
                    n = node;
                } else if (node.estimate() < n.estimate()) {
                    n = node;
                }
            }
            if (targets.contains(n.getID())) {
                List<EntityID> path = new LinkedList<>();
                while (n != null) {
                    path.add(0, n.getID());
                    n = nodeMap.get(n.getParent());
                }
                this.result = path;
                return this;
            }
            open.remove(n.getID());
            close.add(n.getID());

            Collection<EntityID> neighbours = this.graph.get(n.getID());
            for (EntityID neighbour : neighbours) {
                Node m = new Node(n, neighbour);
                if (!open.contains(neighbour) && !close.contains(neighbour)) {
                    open.add(m.getID());
                    nodeMap.put(neighbour, m);
                }
                else if (open.contains(neighbour) && m.estimate() < nodeMap.get(neighbour).estimate()) {
                    nodeMap.put(neighbour, m);
                }
                else if (!close.contains(neighbour) && m.estimate() < nodeMap.get(neighbour).estimate()) {
                    nodeMap.put(neighbour, m);
                }
            }
        }
    }
```

Listing 4: Node Class

```java
private class Node {
    EntityID id;
    EntityID parent;
    double cost;
    double heuristic;

    public Node(Node from, EntityID id) {
        this.id = id;

        if (from == null) {
            this.cost = 0;
        } else {
            this.parent = from.getID();
            this.cost = from.getCost() + worldInfo.getDistance(from.getID(), id);
        }

        this.heuristic = worldInfo.getDistance(id, targets.toArray(new EntityID[targets.size()])[0]);
    }

    public EntityID getID() {
        return id;
    }

    public double getCost() {
        return cost;
    }

    public double estimate() {
        return cost + heuristic;
    }

    public EntityID getParent() {
        return this.parent;
    }
}
```

5.5.3 Editing the Module Configuration File

You must edit the module configuration file `src/adf/config/module.cfg` related to a path planning to use your code. The Figs. 4 and 5 show the part of the default module.cfg and the part of the edited module.cfg where the lines related to a path planning are changed. In this case, all `adf.sample.module.algorithm.SamplePathPlanning` in the file are replaced with `myteam.module.algorithm.AStarPathPlanning`. If you would like to use the code in some modules, you can indicate that the only modules refer to it.

```
SampleRoadDetector.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
SampleSearch.PathPlanning.Ambulance :  adf.sample.module.algorithm.SamplePathPlanning
SampleSearch.PathPlanning.Fire :  adf.sample.module.algorithm.SamplePathPlanning
SampleSearch.PathPlanning.Police :  adf.sample.module.algorithm.SamplePathPlanning
ActionExtClear.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
ActionExtMove.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
ActionFireFighting.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
ActionTransport.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
CommandExecutorAmbulance.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
CommandExecutorFire.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
CommandExecutorPolice.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
CommandExecutorScout.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
CommandExecutorScoutPolice.PathPlanning :  adf.sample.module.algorithm.SamplePathPlanning
```

Figure 4: Default module.cfg

6 RoboCup Rescue Simulation Server

6.1 Architecture

The simulator is in fact divided into many other simulators:

- the `clear` simulator, responsible for blockade removal;
- the `collapse` simulator, responsible for managing buildings' structural damage and blockade creation;

```
SampleRoadDetector.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
SampleSearch.PathPlanning.Ambulance :  myteam.module.algorithm.AStarPathPlanning
SampleSearch.PathPlanning.Fire :  myteam.module.algorithm.AStarPathPlanning
SampleSearch.PathPlanning.Police :  myteam.module.algorithm.AStarPathPlanning
ActionExtClear.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
ActionExtMove.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
ActionFireFighting.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
ActionTransport.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
CommandExecutorAmbulance.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
CommandExecutorFire.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
CommandExecutorPolice.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
CommandExecutorScout.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
CommandExecutorScoutPolice.PathPlanning :  myteam.module.algorithm.AStarPathPlanning
```

Figure 5: Edited module.cfg

- the `ignition` simulator, responsible for firing up random buildings during the simulation;
- the `fire` simulator, responsible for the fire spread between buildings;
- the `traffic` simulator, responsible for humans' movement;
- the `misc` simulator, responsible for human damage and buriedness.

These simulators establish connections to the *kernel simulator*, responsible for coordinating the simulators' processes and centralizing the data they generate.

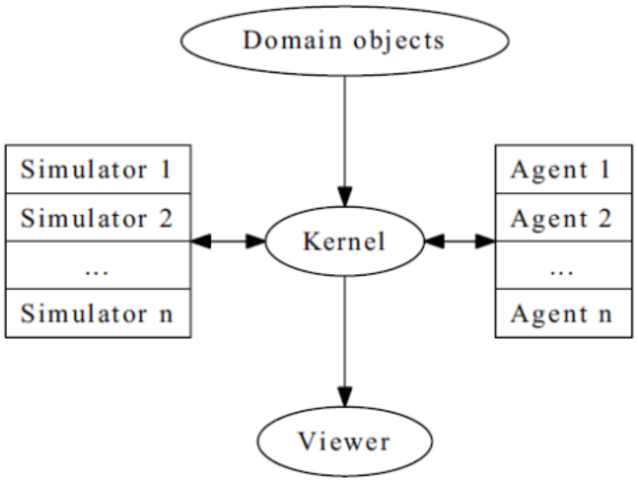


Figure 6: RoboCup Rescue Agent Simulation platform architecture Skinner and Ramchurn (2010)

The RoboCup Rescue simulator was designed to create a *partially observable*, *discrete-time*, *dynamic*, *stochastic*, *multiagent* environment. In other words, in this environment:

- the complete world current state cannot be known through a single agent's perception (even if the agent has an infinite range of sight, it still will not be able to see through a building's walls);
- time is divided in intervals, as opposed to continuous time;
- there are random elements that affect its state transition;
- there is more than one agent present, and one's actions may interfere with the others' performance.

Time is divided in *time-steps*; during each time-step, the agent perceives the environment and reasons about what action it will perform. In each time-step, the following happens:

1. The kernel updates all agents' perception (visual and communication) and waits for the agents' commands.

2. The agents updates their world model and make their decisions, sending their commands to the kernel.
3. The kernel sends the agents' commands to the simulators.
4. The simulators process the agents' commands and send the changes suffered by the environment back to the kernel.
5. The kernel sends the environment changes to the viewers.

6.2 Entities

Several objects are represented in the simulator as depicted in Figure 7.

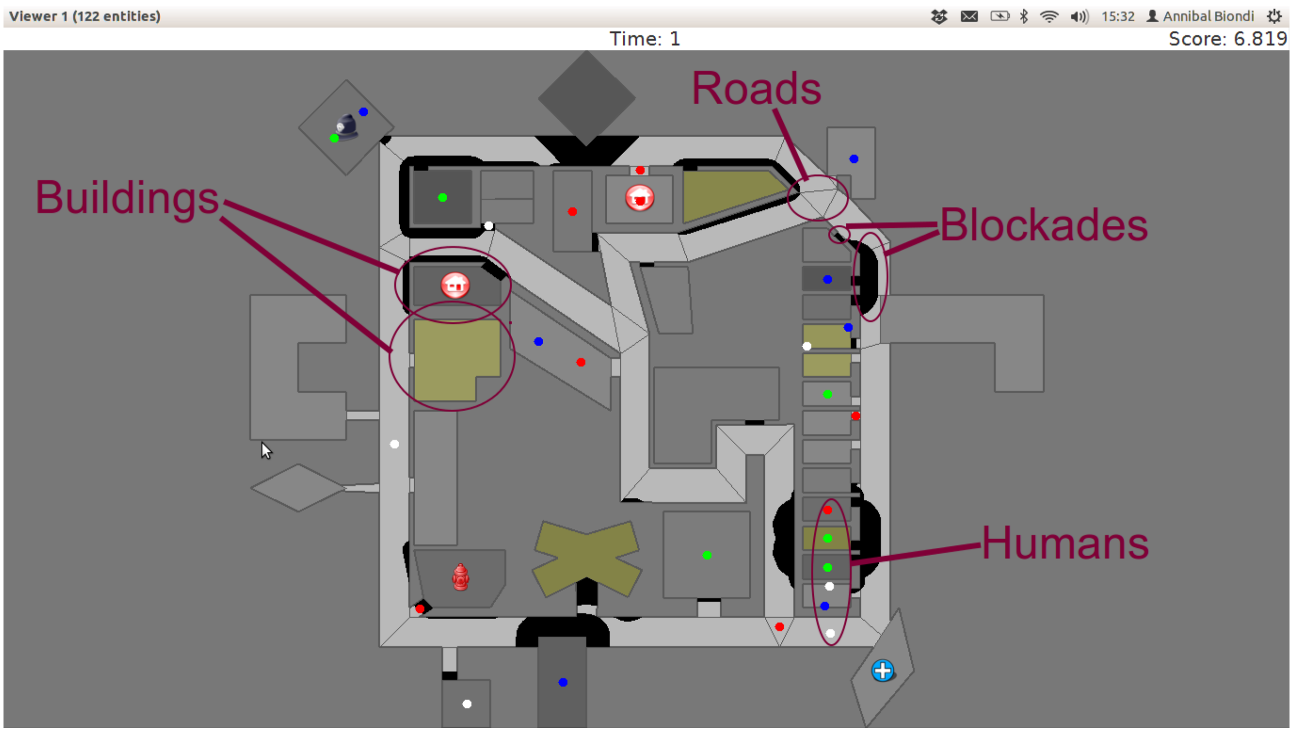


Figure 7: Entities of the simulator

6.2.1 Blockades

Blockades obstruct the path of agents and civilians; they are represented as black polygons in roads. Blockades appear in the beginning of the simulation and are not produced after this. They must be removed by Police Forces.

Properties

- `position`: ID of the road to which the blockade belongs
- `repair cost`: cost to completely remove the blockade from the road
- `shape`: a rectangle which surrounds the whole blockade
- `X` & `Y`: coordinates of the blockade's centroid
- `apexes`: vector containing the apexes of the blockade

6.2.2 Area

Area entities represent buildings and roads.

Properties

- `blockades`: a list with the blockades in that area
- `edges`: a list with the edges that limit the area
- `neighbours`: a list of the areas that can be accessed from this area
- `X` & `Y`: coordinates representing the area in the map

While both buildings and roads have the *blockades* attribute, blockades appear only in roads.

6.2.3 Buildings

Buildings group all kinds of buildings in the simulator: besides the regular ones, described below, there are special kinds of buildings (*refuges*, *ambulance centres*, *fire stations* and *police offices*, shown in Figure 8) which cannot catch on fire. They will be described in later sections of this document.

(a) Refuge (b) Ambulance Central (c) Fire Station (d) Police Office

Figure 8: Special buildings

Properties

- `brokenness`: how structurally damaged the building is; does not change during the simulation
- `fieryness`: the intensity of the fire and fire-related damage in the building
  - `UNBURNT` (not burnt at all)
  - `WATER_DAMAGE` (not burnt at all, but has water damage)
  - `HEATING` (on fire a bit)
  - `BURNING` (on fire a bit more)
  - `INFERNO` (on fire a lot)
  - `MINOR_DAMAGE` (extinguished but minor damage)
  - `MODERATE_DAMAGE` (extinguished but moderate damage)
  - `SEVERE_DAMAGE` (extinguished but major damage)
  - `BURNT_OUT` (completely burnt out)
- `floors`: the number of floors the building has
- `ground area`: the area of each floor
- `ignition`: indicates if the simulator has lit this building on fire[4]
- `importance`: (unknown function; has equal values to all buildings)
- `temperature`: temperature of the building; if it crosses a certain threshold, the building catches on fire
- `total area`: the total area of the building (`floors X ground area`)

Regular buildings are represented as polygons of various colors, depending of their status, as shown in Figure 9; the darker the color, the greater the structural fire or water damage.

In the beginning of the simulation, broken buildings trap humans inside it under debris; these debris must be removed by Ambulance Teams, who then proceeds to rescue the human.

A *refuge* is a special kind of building: it represents a place destined to support the rescue activity, providing medical care for the wounded and water to the Fire Brigades. In the simulator, humans inside a refuge have their damage zeroed, which means they do not lose health while they stay there; damage will, however, resume when the human entity leaves the refuge.

Also, Fire Brigades have their water supply replenished by a certain amount during each cycle while they are inside the refuge.

[4]A building can catch on fire by being ignited by the simulator or by being close to a burning building; ignition will be set to "1" if the building was, at some point of the simulation, ignited by the simulator

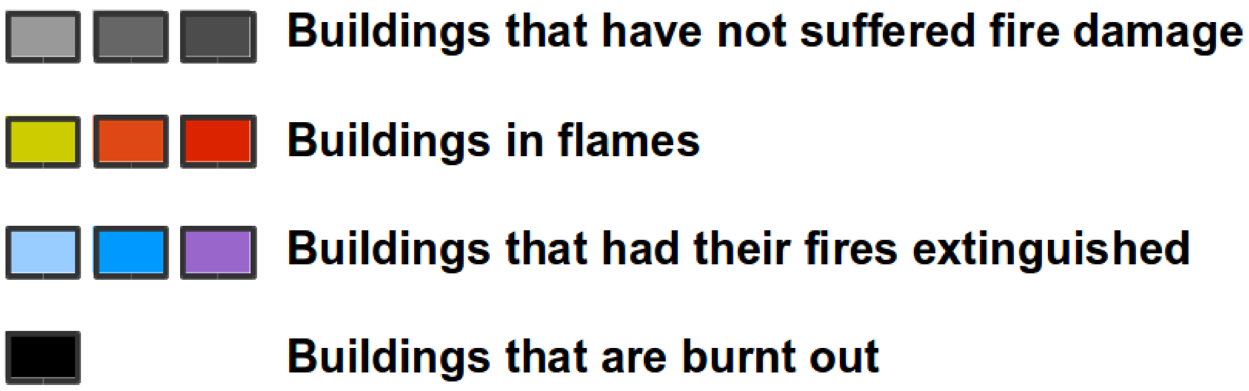

Buildings that have not suffered fire damage
Buildings in flames
Buildings that had their fires extinguished
Buildings that are burnt out

Figure 9: Possible status of regular buildings

6.2.4 Roads

Area entities representing roads have no new attributes besides those of *area* entities.

6.2.5 Humans

These are the entities representing humans. In the simulator, they can be *Civilians*, *Ambulance Teams*, *Fire Brigades* or *Police Forces*. They are all represented by circles of different colors, and cannot move by themselves if they are dead or buried.

- `buriedness`: how deep the human is buried
- `damage`: how much HP the human loses per cycle; zeroes when a refuge is reached
- `direction`: direction to where the human is moving (inferred); the Y-axis positive half is zero, and the value increases until 129599 (360*60*60 - 1) seconds anti-clockwise
- `HP`: health points of the human; if it reaches 0, the human dies
- `position`: ID of the entity where the human is; may be an area entity or a human entity (if it is inside an ambulance)
- `position history`: a list of the entities the human has passed during the last cycle, in chronological order
- `stamina`: not implemented; would decrease each time the agent took an action and would be partially replenished at the beginning of each cycle
- `travel distance`: (unknown)
- `X` & `Y`: coordinates representing the human in the map

The color of each human in the simulator is defined by its type and its health: the lower its health, the darker it is. Dead humans are represented by the black color.

6.2.6 Civilians

Civilians are human entities and they are not part of a rescue team; they are represented by the color green. Their standard behavior is to walk to the closest refuge on their on if they are not wounded or buried; otherwise, they will have to be transported by an Ambulance Team.

6.3 Agents

These are the entities that will compose your rescue team; in other words, this is what you will program. Agents are divided in two types: *platoon agents* and *central agents*.

6.3.1 Platoon agents

Platoon agents are able to interact with the simulated environment through perception and executing actions on it. They can also exchange messages with other agents by vocal or radio communication. They are comprised of three different categories: the *Ambulance Team*, *Fire Brigade* and *Police Force*.

Ambulance Team *Ambulance Teams* are responsible for rescuing humans (agents and civilians) and take them to a refuge. They are able to unbury victims and carry one person.

Fire Brigade *Fire brigades* are responsible for extinguish fires on buildings. Moreover, they carry a certain amount of water in their tanks and they can replenish it in a refuge.

Police Force *Police forces* are responsible for removing blockades from the roads. When ordered to do so, they will clean a certain amount, specified in the repair cost parameter, from the target blockade at each cycle. However, differently from Ambulance Teams and Fire Brigades, having two Police Forces acting on the same blockade brings no advantage to the process: it will be as though there was only one Police Force acting on it.

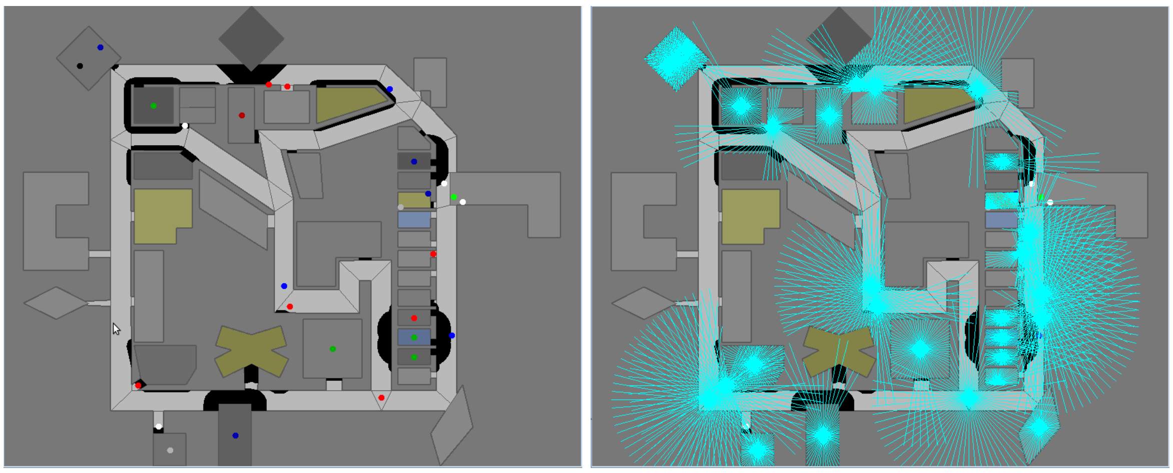
6.3.2 Central agents

Central agents are a type of agents whose only interaction with the world is through radio communication. There are three types of central agents: *Ambulance Centers*, *Fire Stations* and *Police Offices*, and they are represented as buildings.

6.4 Perception and Commands

The simulator has two perception modes: *standard* and *line of sight*. We will discuss just the latter, since it is the one used in competitions.

Line of sight perception simulates visual perception of the agent: a vision range and a number of rays are defined and the agent percepts anything that is reached by these rays.



(a) No LoS (b) LoS

Figure 10: Normal viewer and LoS perception

The set of currently visible entities for an agent is stored in a structure named *ChangeSet*; entities present in it are automatically updated in its world model; that is, if an agent perceives a blockade it did not know that was there before, this blockade is automatically added to its world model. The opposite, though does not happen: if the agent does not perceive a blockade any more, nothing in its world model changes, even if it knew that there was a blockade there before. In that case, the agent will still think that there is a blockade in that road, even though such blockade has already been cleared. Thus, it is up to the agent to figure this out and modify its world model accordingly.

6.5 Communication

There are two forms of communication available in the simulator: *direct communication* and *radio communication*. Direct communication, done with the command *speak*, is communication audible to humans within a radius from the emitter agent, as if the emitter shouted something.

Radio communication is done with the command *tell*, and transmits information to all agents that are signed up to the channel on which it was broadcasted. Radio communication channels are present in limited number, each one with a limited bandwidth.

In both types of communication, the message has to be coded into a string of bytes before being sent; the receptor must decode it once it receives the message. Both types might be susceptible to message *drop out*, where the message is not received by its targets; radio communication is also susceptible to message *failure*, where the message is received empty.

References

Skinner, C. and Ramchurn, S. (2010). The robocup rescue simulation platform. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1*, AAMAS '10, pages 1647–1648, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.