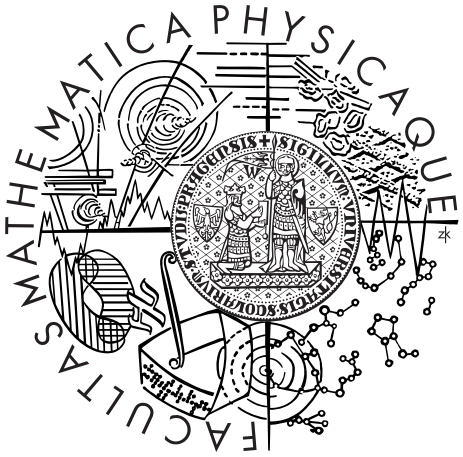
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Martin Vejman

Pogamut and USARSim integration

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Michal Bída
Study programme: Informatics
Specialization: General Informatics

Prague 2012

I would like to thank my supervisor Mgr. Michal Bída for his patience, support and kind critical comments that directed my effort while working on this thesis.

I would also like to thank my mother for her continual support.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Integrace Pogamutu a platformy USARSim

Autor: Martin Vejman

e-mail autora: vejmanm@ssakhk.cz

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Michal Bída

e-mail vedoucího: Michal.Bida@mff.cuni.cz

Abstrakt:

Tato práce se bude zabývat integrací robotického simulátoru USARSim s platformou Pogamut, která umožňuje snadné prototypování virtuálních agentů. Jako doklad korektnosti poskytneme několik robotů, kteří řeší jednoduché úkoly. Dílčím cílem práce je zhodnotit vhodnost reaktivního plánovače POSH pro programování robotů USARSimu. Z tohoto důvodu budou vytvořeny dvě verze létajícího robota řešícího složitější úkol. První verze bude naprogramována v Javě a druhá pomocí nástroje POSH. V práci pak provedeme srovnání těchto dvou verzí.

Klíčová slova: USARSim, Pogamut, POSH, virtuální roboti, umělá inteligence.

Title: Pogamut and USARSim integration

Author: Martin Vejman

Authors's e-mail: vejmanm@ssakhk.cz

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Michal Bída

Supervisor's e-mail: Michal.Bida@mff.cuni.cz

Abstract:

In this work we integrate the USARSim simulator with the Pogamut platform in order to facilitate the development of USARSim virtual robots. We present a set of robots solving simple task as a proof-of-concept. We also provide an aerial robot solving more complicated task. This robot will be used to evaluate the suitability of reactive planner POSH for controlling robots in USARSim. Two versions of the robot will be created and compared qualitatively. The first version will use pure Java, the second version will use reactive planner POSH.

Keywords: USARSim, Pogamut, POSH, virtual robots, artificial intelligence.

Contents

Introduction **3**

1 Technologies used **4**
1.1 Unreal Tournament . . . 4
1.2 USARSim . . . 4
1.3 POSH . . . 5
1.4 Pogamut . . . 6

2 Problem Analysis **7**
2.1 Related work . . . 7
2.1.1 MOAST . . . 7
2.1.2 Player . . . 7
2.1.3 Pyro . . . 7
2.2 Technical specification . . . 8
2.2.1 USARSim communication . . . 8
2.2.2 Info messages . . . 9
2.2.3 Commands . . . 9
2.2.4 Robot Types . . . 10
2.2.5 More complicated Problem . . . 11

3 Implementation **12**
3.1 Containers for info messages . . . 12
3.2 TCP/IP message parser . . . 12
3.3 Modules . . . 14
3.4 Commands . . . 15
3.5 Set of sample robots . . . 16
3.5.1 Submarine - Nautical vehicle . . . 16
3.5.2 ERS - Legged robot . . . 18
3.5.3 P2DX - Ground vehicle . . . 19

4 POSH vs. Java **21**
4.1 Solution analysis . . . 21
4.2 Scan preview . . . 23
4.3 POSH implementation . . . 24
4.4 Java implementation . . . 27
4.5 Comparison of both . . . 28
4.6 Results . . . 28

5 Discussion **31**
5.1 Errors encountered . . . 31
5.2 Lessons learned . . . 31
5.3 Future work . . . 32

Conclusion **33**

Bibliography **34**

CD contents **36**

Introduction

The field of robotics is becoming increasingly relevant in our everyday lives. There are new technologies allowing engineers to design complex robot mechanisms, but the artificial intelligence (AI) is on low levels or there is none at all. Even the Mars Exploration Rover had to have operators navigating it through terrains to obtain images and samples of ground. Robust and autonomous AI systems for robots would help us utilize robots more effectively. However, the research faces numerous obstacles, one of which is having the robot itself. Robots are expensive to build and to maintain and we have to travel to particular area sites for testing and validating algorithms. And since we do not know if our robot designs are any good, we may be spending time and effort pointlessly.

Fortunately there are platforms such as USARSim, which offer robots for free in a form of simulation. Development of robots in simulated environments has multiple benefits. It allows users to model versatile robotic systems in a similar manner as in reality and to put them into specific fabricated scenarios. Such a platform is convenient for AI development.

However, the USARSim usage is still not as easy to use as we would like. Therefore, we decided to create a Java superstructure as a plug-in for Pogamut platform. Pogamut is established platform for intelligent virtual agent development. By integrating USARSim with Pogamut, we expect to facilitate the usage of USARSim and to enable development of USARSim robots in Pogamut.

This thesis deals with the connection of robotic simulator USARSim to the Pogamut platform via Java programming language. Having a support of all USARSim features in Pogamut should enable us to assemble simplified set of modules for the intelligent virtual agent (IVA) development. Also a set of example robots should be developed as a proof of concept prototypes. These prototypes shall expose different kinds of robot control. We evaluate the suitability of a reactive planner POSH for controlling robots in USARSim as a partial goal by solving more complex problem using one of USARSim's robots implemented in POSH. This is compared with another solution implemented in Java.

The outcomes of this thesis include a) PogamutUSAR2004 Java library - a plugin for Pogamut platform enabling the development of USARSim robots, b) three simple example USARSim robots created with PogamutUSAR2004 and c) aerial robot solving more complex problem with customly designed sensors. Two versions of a robot exists, one of them is using POSH reactive planner and the second one uses plain Java.

This paper is organized as follows. In the first chapter we will look at used technologies. In the second chapter we will discuss the assignment in detail and we will mention similar software dealing with this matter. The third chapter will cover the implementation, and description of the set of sample robots. The fourth chapter will describe more complex problem solved by an Aerial vehicle controlled by reactive planer POSH. A comparison with a solution of this problem in Java will be covered by this chapter also. In the fifth chapter we will draw conclusions and summarize faults encountered during the development.

1. Technologies used

In this chapter we provide a general overview of technologies and tools used throughout this thesis.

1.1 Unreal Tournament

Unreal Tournament (UT) is a popular 3D game made by Epic Games in the first person shooter genre. The game is displayed in figure 1.1. UT is based on Unreal Engine, which is a powerful 3D engine used in some of the modern computer games. It provides a complete physics engine. There is a modification of UT called GameBots that provides information about the game such as position of agent through simple text based protocol. Essentially, it creates a TCP/IP socket to exchange data between Unreal engine and the controller. The USARSim simulator utilizes Gamebots and extends its protocol.

Figure 1.1: Unreal Tournament 2004 game insight.

1.2 USARSim

Unified System for Automation and Robot Simulation (USARSim) [1] is an open source high-fidelity simulation tool that can be used to simulate various robots and environments. As shown in figure 1.2, it offers different types of robots as

well as wide range of sensors that can be easily mounted on any robot just by changing the configuration file. Sample worlds are offered as additional packages. USARSim is based on the Unreal Tournament game engine that provides high performance physics simulation. This tool has proven to have a broad scope of application as it has been used for the RoboCup Virtual Robot competition [2] as well as for the IEEE Virtual Manufacturing Automation Competition (VMAC) [3].

Figure 1.2: Various robots supported by USARSim for UT2004, from the left Talon, StereoP2AT and HOAP.

There are currently three versions of USARSim [1] all based on Unreal engines that provide high-fidelity physics simulation. Unreal Tournament 2004 (UT2004), Unreal Tournament 3 (UT3) and the newest version is for Unreal Development Kit (UDK). At the point of deciding which version to use in our project, the UDK version had not been released yet — the UT3 version was the newest. Despite the fact that the UT3 version is based on the newer engine, we decided to use an older UT2004 version for the following reasons. The UT3 version offered just a few robots, which was not compatible with our aspiration of being able to perform fast prototyping of variety types of IVA in Pogamut. Ultimately there were some minor issues related to physics (namely with P3AT robot) [4]. This constituted a certain risk of not being able to use the tool properly. From now on we will discuss only the UT2004 version of USARSim unless specified otherwise.

1.3 POSH

Reactive planning is a way of selecting actions for agent by a system that selects his next action by a quick look-up in a dynamic data structure.

Parallel-rooted, Ordered Slip-stack Hierarchical (POSH) planner is a reactive planner for controlling behaviour of virtual agents. POSH was created by J.J.Bryson at University of Bath [5] and is based on Behaviour Oriented Design (BOD) [6], which is a methodology for creating intelligence of complete complex agents. That covers autonomous robots. POSH plans are structures for selecting actions in BOD. What makes the system reactive is that the decision making of what to do next, is based on the present state of the world perceived by agent's sensors.

POSH primitives are *Acts* and *Senses*. *Acts* are used for triggering operations and *Senses* are used for perceiving data. Both are leaves in POSH hierarchical plans and both depend on the common programming language (Java in our case).

Combination of multiple primitives requires aggregations. There are three ways how to do that.

- `Drive collection`: It is the root of the POSH hierarchy and it is what each cycle decides which goal should the agent be working on.
- `Competences`: They represent basic reactive plans.
- `Action patterns`: Simple series of actions are gathered in these.

POSH script files have a syntax designed in a way that is easy to parse in LISP. It can be edited manually, but we use a GUI editor [7]. Thanks to GUI, editing POSH plans is easier and faster. We do not need to deal with tedious process of learning POSH syntax.

1.4 Pogamut

Pogamut [8] is an open source platform for fast development of virtual agents embodied in 3D environments such as Unreal Tournament 2004. This toolkit provides low-level management of basic operations such as spawning, controlling and even debugging agents, allowing developers to focus on higher principles. Pogamut is convenient for educational purposes and for research. It allows rapid reactive agent development and it facilitates development of agents with advanced artificial intelligence. There is, among other plug-ins, a support of reactive planner POSH and visual editor for POSH plans. Therefore developers can implement an agent's behaviour directly in Java or use simply the high-level API to create decision trees as the algorithm for selecting an action to execute.

2. Problem Analysis

This chapter presents related work and discusses the technical aspects, communication with the server, capabilities of USARSim and its modifications.

2.1 Related work

We will briefly review most common tools used as user side applications called *Controllers*. The controller usually works in following way. It connects to Unreal server and sends a command to spawn an USARSim robot. After the robot is created in the simulator, the controller listens to sensor data and sends commands to control the robot.

2.1.1 MOAST

Mobility Open Architecture Simulation and Tools (MOAST) [9] is a general purpose research tool used for low-level robotic control as well as for controlling heterogeneous robot group behaviour. It is a framework providing infrastructure for development, testing and analysis of autonomous systems. There is also a mechanism to migrate algorithms from the virtual world to entirely real implementation. MOAST has been utilized by the RoboCup and VMAC competitions. Compared to MOAST, our work does not need to migrate developed systems to the real implementation.

2.1.2 Player

Player [10] is a network server, and client program talks to it over a TCP socket reading information from sensor and writing commands to actuators. It gives user simple and complete control over the sensors and actuators on the robot. Player serves as a hardware abstraction layer for robotic devices. It defines a set of specifications describing ways to interact with some class of device. It communicates with specific hardware via device driver and provides standard interface to its clients. Therefore it is not important which brand of sensor is used, it simply provides an unified interface to users. This allows portability to other robots even from virtual to a real one. Player is primarily used on Linux system and other POSIX platforms, whereas our work is multiplatform.

2.1.3 Pyro

Python Robotics (Pyro) [11] is a Python library that abstracts all of the underlying hardware details. It can be used both for robot simulators and for real robotics research. Not all USARSim robots are supported. Its goal is to make topics in robotics and AI easy to explore. Pyro is out of date and no longer supported.

2.2 Technical specification

Every robot in USARSim is composed of a chassis, mechanical parts such as wheels, propellers, rudders and mission package mechanisms. Apart from these parts that are fixed, there are also configurable auxiliary parts such as joints, cameras, sensors, effecters, grippers, headlights, etc. Configurable parts are easy to modify via *USARBot.ini* file located in the system folder of UT2004. Detailed description of configuring this file is described in [12].

Mission package constitutes a set of parts linked together by controllable joints. Mission packages are used for constructing robotic arms, camera mechanisms for panning and tilting the camera and for other arrangements of movable segments that are not a part of robot's motion gear.

Sensors are essential part of this simulator as they embody a perception of the world. There are many different types of sensor that can measure speed, acceleration, range from objects, establish position, distinguish sound, ball location or human motion. It is worth mentioning that the lengths are measured in meters and angles in radians.

Ultimately there is a possibility of creating our own sensors and effecters, even our own robots and mission packages. We can even add our own command. These capabilities indicate the opportunities we are given by using this tool, all modifications are explained in [12]. We did not need to use any of these specific modifications in our work.

2.2.1 USARSim communication

In order to create a proper client application to exchange data with the USARSim server, we need to understand its protocol and communication structures. Controller has to be able to connect with the server, send commands and receive messages. Messages received from the server are called *info messages*.

USARSim follows simple GameBots communication text-based protocol. It opens TCP/IP socket for the purposes of information exchange. IP address of this socket is the IP address of the computer that runs the server and the default port number is 3000. Both commands and info messages have the following form:

`TYPE {Segment1} {Segment2} ...`

- *TYPE* indicates the type of command or info message and suggests what segments will follow. It is presented in upper case characters.
- *Segment* is one or more name-value pairs. String of non-white characters is considered to be a valid name or value figure. Names and values are separated by one white space.

TYPE and the first *Segment* as well as the other following *Segments* are also separated by one white space. The message is terminated by *carriage return* and *line feed* character ("\r\n").

Example of an info message received from the server and a command the controller could send might look like this:

- `SEN {Type Sonar} {Name L0 Range 1.8396} {Name R0 Range 4.9954}` - Sensor info message reporting left and right sonar ranges.

- `INIT {ClassName USARBot.P2DX} {Name Robot1} {Start PlayerStart}` - Command that will spawn P2DX robot at a start pose called PlayerStart.

2.2.2 Info messages

It is important to have a knowledge about various types of info messages to be able to select a suitable data structure when treating incoming information. This is the summary of info messages we need to be prepared to receive:

- `STA`: *State message* contains the state of a robot. It is a periodical info message and it begins to appear with spawning of a robot to the world.
- `MISSTA`: If a robot is equipped with a mission package, *Mission package state message* will be sent from the server along with the *State message*. It reports the state of a mission package.
- `SEN`: Depending on the sensor equipment, *Sensor messages* will be received with this header as often as individual sensor's scan interval is adjusted.
- `GEO`: *Geometry message* reports geometry information about mission packages, sensors, effecters or robots.
- `CONF`: *Configuration message* is used to carry configuration information about the same subjects as *Geometry messages*.
- `RES`: *Response message* provides a response status of sensor or effecter setting command.
- `NFO`: *Game info message* appears as soon as a controller connects to the server and it holds a type of the game, name of the level and a time limit. This header is also used for info messages that bear information about possible spawning point names, locations and rotations.

Some info messages such as *State messages* and *Sensor messages* are issued regularly and the rest of them is processed in a response to a specific initiative. It is important to keep in mind that each type of info message may well tell us what segments will follow, but it mostly depends on secondary facts such as type of vehicle, type of sensor or type of geometry or configuration data.

2.2.3 Commands

There are also various types of commands we should know about to properly select a suitable data structure when processing the control tasks. This is the summary of commands we need to be able to dispatch:

- `DRIVE`: Drive command makes the robot move. Different type of robot is controlled by a different set of parameters. For example, Ground vehicle might demand spin speed for left and right side wheels to start moving while Nautical vehicle requires rudder angle and propeller speed defined.
- `MULTIDRIVE`: To fulfil the needs of a complicated movement such a Legged robot represents, there is a command to move multiple joints at once.

- `INIT`: Command for spawning robots into the world. Items such as the world controller[1] or communication station[2] can be spawned by this command, too. We did not need to use either of these items in our project.

- `GETGEO`: Queries the geometry information about specified type of sensor, effecter or robot, alternatively mission package. As a consequence, *Geometry message* is sent by the server.

- `GETCONF`: This command can query the configuration information about the same subjects as `GETGEO` command. Causes *Configuration message* to emerge.

- `SET`: Set command is used for complementary control such as sensor or effecter commands, for example to reset sensor or to activate gripper. This command is also used to set up viewports and viewpoints.

- `MISSPKG`: We can use this command to set multiple joints at the same time to control the mission package. Also we can specify the pose of the last joint and poses of all connected joints will be computed for us.

- `TRACE`: By this command we can turn on/off the capability of tracing the path of a robot. It drops points into the world in specified interval and color as the robot moves.

- `STARTPOSES`: Sends a request to receive *Game info message* that holds a set of possible start positions. Each map has at least one start position. It allows the controller to automatically spawn the robot.

We need to comprehend all the info messages and commands, to be able to contemplate the way of implementing containers for both commands and info messages as effectively as possible. Henceforth, if not explicitly stated as a header, we will understand headers of info messages and commands used in a sentence as abbreviations for the corresponding message string with all respective segments.

2.2.4 Robot Types

There are currently four types of robots in USARSim. Each and every robot is either Aerial, Nautical or Ground vehicle, alternatively a Legged robot. Consequently, there are different segments of *State message* relevant for each type of robot meaning different structure of STA message occurs depending on the vehicle type. Each type of robot has a different kind of control, too. Clearly a Legged robot can not have the same DRIVE command parameters as a Nautical vehicle. Ground robot for example might require steer angle of front and rear wheels or speed of left and right side wheels, whereas Aerial vehicle needs linear, lateral, altitude and rotation speeds defined to start moving. Nautical vehicle has ruder and stern plane angles adjustable and a propeller speed causes it to move ahead. Legged robot has multiple joints, which need to be set in order to operate the robot in a walk-like manner.

[1] World controller allows to add, move or delete object during the simulation
[2] Communication station can be used as a rely point for multiple robots

2.2.5 More complicated Problem

In order to properly verify our implementation of a controller, in which AI agent can be designed, we shall create a robot solving more complex problem. Aerial vehicle called AirRobot portrayed in figure 2.1 is a standard USARSim robot and will be used for performing the solution.

The task this robot ought to master is following. The robot will be spawned to the world at any given point, it will be given the coordinates of an area that the robot should explore. It will provide a height map of this area as an output. For that it will use Range scanner sensor mounted on the bottom of the robot. This aircraft should maintain a reasonable altitude, so it will not get stranded in often changing terrain relief, and so that it will effectively utilize the field of view of the Range scanner. Another distance measuring sensor should be used during the flight for avoiding obstacles such as walls, trees and others. The robot itself should assess when to go back to the starting position to recharge the battery.

The reactive planner POSH that is already integrated in Pogamut will be used to solve this problem. And to clarify whether the POSH is suitable for solving this task or not, there shall be provided another solution in plain Java as a comparison.

This problem might be useful in future robotics. Height map can be visualized in 3D, thus enhanced terrain maps or help for the blind might benefit from this.

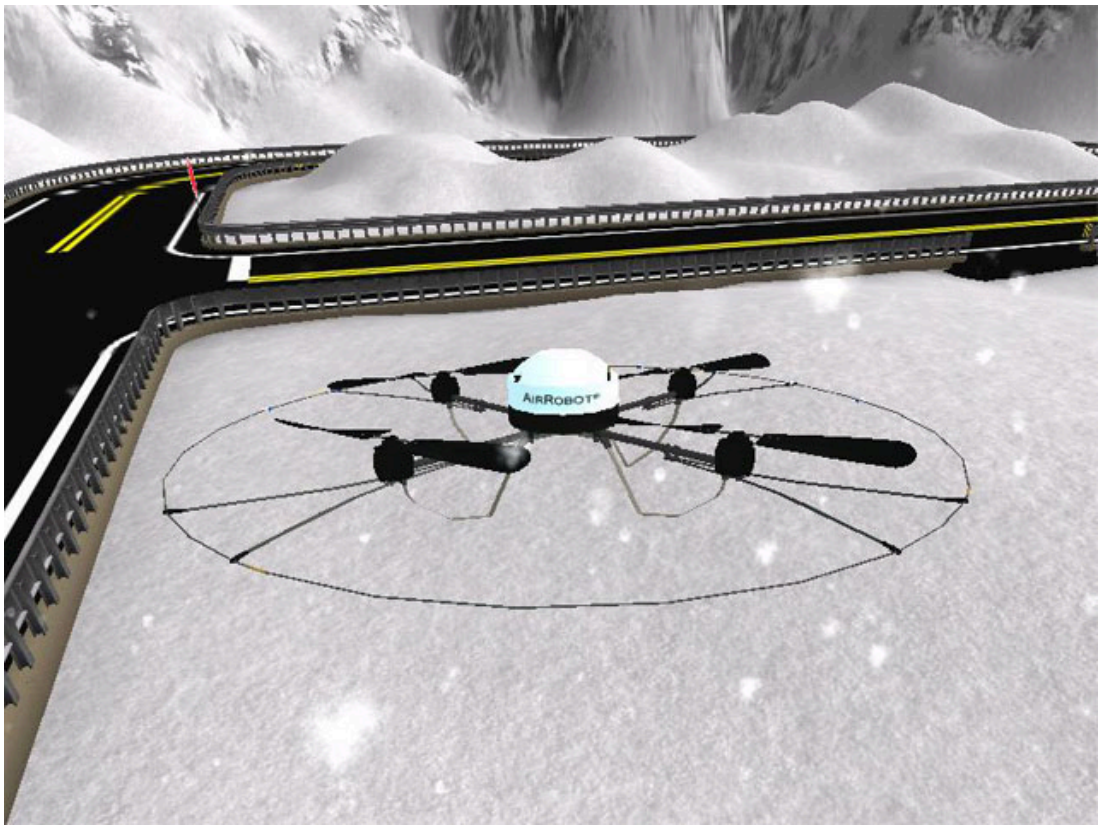


Figure 2.1: Demonstration flight of AirRobot used for solving more complicated problem utilizing POSH planner.

3. Implementation

This chapter examines the implementation of containers for info messages and for commands. Modules and sample robots are introduced.

The integration with Pogamut lies in utilizing the Pogamut infrastructure for agents, which provides us with the capability of connecting to the server, sending command messages via Command serializer and we can also register Listeners for any type of info message.

The LogicController is able to make further use of the agent infrastructure providing a *logic* method that is triggered periodically. In our case the logic method is triggered every time *State message* is received. This is where we expect the agent behaviour to be implemented. The period of STA messages depends on *USARBot.ini* file adjustment. Each robot may have different period but it is generally set to five times per second.

Our derived implementation of the agent infrastructure connects to the server and receives a vain NFO message that tells us useless information for us about the level. The reaction for that is a STARTPOSES command that results in obtaining another NFO message. This time, it is very important info message as it bears data about start poses. This is when our implementation of the agent lets the LogicController know it can spawn the robot by sending INIT command. As soon as it does, the STA messages start to appear, therefore the logic starts to execute.

In fact, every sample robot is derived from a LogicController.

3.1 Containers for info messages

For one type of info message, there might be different unrelated segments used in different situations. For example *Sensor messages* for Sonar and GPS sensors share only the "SEN" header. Thus it might seem misleading to use one container object with large number of parameters for each type of info message. Yet, we decided to do precisely so, since these containers will serve for input processing of info messages. We need to make this part primarily fast and reliable. The quantity of info messages appearing on the input can be great. Namely, range scanner sensor that measures a distance with laser ray, causes info message with 180 double-precision numbers to be sent five times every second, provided that it has implicitly adjusted point of view, resolution and a scan interval. Naturally, a robot can have multiple sensors of the same kind mounted on its body. This is why we do not want to waste time pigeonholing. We will refer to these containers as *elementary structures*.

3.2 TCP/IP message parser

JFlex is a lexical analyzer generator for Java focused on speed and full Unicode support. We use it to transfer text messages received from the USARSim server into Java objects. Using this tool requires creating a specification file that will be compiled by JFlex to a single java class. This class will have a constructor taking

a Stream as an input to be read from. The specification file has the following structure:

```
User Code
%%
Options and declarations
%%
Lexical rules
```

The first part contains code that will be directly copied at the beginning of the generated class. We need to specify target package for the generated class as well as imports that will the parser use. This section is terminated by "%%".

The second part contains options, macros and regular expressions. Here we can specify which interface should the generated class implement, if it should be public or not and other customization options [13]. At this point we declare helper methods that will be used by the parser. For example conversion methods are declared here (String to Integer etc.). Also in this part we declare lexical states used by the third part. For each message prefix from the server we claim one state. And the last important statements are macros and regular expressions. We specified these so that they correspond with data types used in messages received from the USARSim server. For example for matching Latitude data we would use "`LATITUDE = {INT} \, {FLOAT} \, {ALPHA}`". Of course INT matches one or more digits, FLOAT matches every form of a decimal number and ALPHA refers to a single character.

The third part contains lexical rules in a form of regular expressions and actions to trigger when the scanner matches associated regular expression. The scanner reads its input and keeps track of every possible regular expression. It executes action with the longest match rule. If more than one regular expressions have the longest match (they match the same input), the scanner uses action of the first rule specified.

For further restriction of regular expressions, lexical states specified in part two can be used. They are represented as integer constants in the generated class. Regular expressions can then be matched only if its associated lexical state is active. The currently active state can be changed during action executing. We use these lexical states to determine the type of incoming message. Distinguishing these types allows us to parse the rest of the message and use prepared data structures to hold data with common identifier. So for instance to parse the Latitude data from: "{*Latitude 47,40.3323,N*}" we would specify this:

```
<MSG_SEN>{
 "{Latitude " {LATITUDE} "}" {
 "Operation sequence in Java."
 }
}
```

The "<MSG_SEN>" signifies a lexical state. The "{Latitude " part matches exactly this string of characters and and the "{LATITUDE}" is a macro we defined in the second section of the specification file.

It is crucial to know the structure of the specification file to be able to extend the project to support a new sensor or a new info message type. It proved to be very reliable and robust tool which can quickly provide desired results.

3.3 Modules

The elementary structures used by the parser to fill them with incoming info message data are very large and many properties are empty depending on the type of the info message. Therefore we created data structures for covering all info messages of the same type and for enabling the user to access received data in a distinctive form (e.g. `SensorModule` provides information about particular sensors). These data structures are called *modules*. For each module there is a set of smaller containers each holding one elementary structure. These containers offer only relevant properties on the outside, and they cover every possible kind of a given info message (e.g. `SensorINS` provides Orientation and Location properties only). We will refer to these containers as *small structures*. Module utilizes a Pogamut infrastructure for agent modules. It is filled with its own listener, that listens for specific info messages from the parser.

Since most info messages has a first segment with "Type" as a name, we can easily distinguish the kind of a message. Therefore we created enumeration for each module with entries containing a string representation of the kind (value of the "Type" segment) and a reference to a small structure class. This allows us to simply ask the enumeration which small structure is dedicated to a given kind of info message in case we want to create a new instance of this container. Every time a module detects a new info message, it uses this procedure for proper classification.

Each type of info message requires slightly different approach. *State messages* appear regularly, therefore it suffices to know which type of vehicle is spawned to the world in order to file the messages within the State module. It is because *Mission package state messages* report state about a component of a robot, that we integrated Mission package module with the State module.

The module for *Response messages* follows the nature of these info messages. Acknowledgements initiated by sensor or effecter setup are gathered in a queue so that user can go through setting successes or failures. If we set some component very often and we do not care about the result of our SET action (we never obtain results from the module), it is good not to deploy this module at all as the *Response messages* cumulate unnecessarily. The info message queue is cleared every time we access its messages.

Modules for *Geometry messages* and *Configuration messages* comply the principle of these info messages that are triggered by GETGEO and GETCONF commands. The modules store data from info messages, but they also have a capability of sending queries to the server. Their utilization resides in following process. User sends a query about some type of configuration or geometry to the server. As soon as corresponding info message is caught by a module's listener, it is available for collection. This is unable to happen within one logic call and it is possible that the server will not be able to respond in two consecutive logic calls. That is why the user has to make sure whether the configuration or geometry information he or she requested is ready or not before trying to read it.

There is an issue with querying configuration and geometry data. The server is not capable of accepting more than one request at a time. This means that if we wanted to know configuration of a robot and a camera for example, we would have to query those requests in different logic calls.

Sensor messages are the most vital info messages as they represent perception of the world. We created a Sensor module which stores messages in its hashmap by type and by name. If a new message with yet unknown type and name is caught, the module creates a new category and within this category it creates a record with a new name. If a new message with known type and name is caught, the old records are overwritten with new ones.

When it comes to sensor data, there are situations when we need to offer users great precision. This is why we created Queued sensor module, which records data just as the non-queued module in the map by type and name with the difference that the new values do not overwrite the old ones, but are stored in a queue. This means that there is a queue of messages for each type and name of sensor that the current robot carries. Whenever a user accesses a queue of a specific sensor the queue flushes.

There may be many sensors mounted on a robot and it may happen that we need information just about one particular sensor. For this kind of situation we prepared Sensor specific module, which is a generic data structure that allows to create an instance of any supported sensor and to provide information just about this sensor.

The queued version of Sensor module should be used in a situation, when we need high quality readings. Logic may be triggered in larger intervals than in which *Sensor messages* arrive. The non-queued Sensor module is advisable to use when we want to know Sensor readings occasionally. For example we want to know whether the robot already traveled ten meters or not. It is not convenient to combine multiple Sensor modules because duplicate data occurs . Also instead of using ten Sensor specific modules, we should use either Queued or non-queued sensor module, because with ten Sensor specific modules we have ten *Sensor message* listeners. At least nine of them are triggered unnecessarily each time SEN message arrives.

3.4 Commands

Each type of command can have various unrelated segments depending on the subject the command belongs to. In this case it is easier to implement more subject-specific classes for targeting directly specific kind of command. For example SET command is capable of controlling effecters, sensors, cameras, viewports or joints, so we created SetSensorEffecter, SetCamera, SetViewports and SetJoint classes each supporting only relevant segments. Sensor and effecter command share the same segments, therefore they share the same class.

These classes are very simple, they contain appropriate properties and one or more constructors which fill inner variables. There is a method that converts properties and its values to a command string using a StringBuilder and following the USARSim protocol.

There are some command classes that have multiple constructors. It is because they support more ways of doing the same thing. For instance an INIT command can spawn specified robot with a specified name either on a start pose or on a place using a point in space and an orientation. So we have to specify either a String with a name or six numbers represented by Location and Rotation. In this case it might happen, that some segments will be empty. This does not matter,

since we do not include undefined variables to the Command string within the converting method. Also the server ignores unrelated or incorrectly formated segments.

3.5 Set of sample robots

Based on the class derived from LogicController and using modules and prepared tools, we demonstrate the usage of those tools on a few simple sample robots. There are currently four types of robots: Legged robot, Aerial, Nautical and Ground vehicle. Hence, we picked one robot of each kind and implemented a simple behaviour. We omitted the Aerial vehicle, though. We used it for solving the more complicated problem, so it is shown amply there how to operate it.

All sample robots are written in Java, although one of them (P2DX - Ground vehicle) is provided also in a POSH version. It is a very simple illustration of using POSH plan, but it is very helpful in getting acquainted with this technology.

3.5.1 Submarine - Nautical vehicle

This sample robot is designed for the ONS-PointLookout_250 map which has the largest water surface. The robot spawns in the air and it immediately falls into the water for it is a submarine. It abounds with a rudder, stern plane and a propeller. The robot is shown in figure 3.1. This it the apparatus we can control by DRIVE command. Implicitly the sub has a camera on a mission package, which is for panning and tilting the camera and there is a single ray laser sensor on a bottom of the submarine hull. That is very little in terms of world perception, yet we used this model to demonstrate several features.

The first and maybe the most important feature covered by this sample robot is obtaining Configuration data about the robot, camera and mission package. As we have established before, we can not send all three GETCONF commands at the same time. We are using Configuration module to query these requests one by one, each in a different logic call. Every time we call the method that handles queries of configuration data, only one of these branches executes:

- If the module is empty we ask about the robot's configuration.
- If the module has one message we ask about the camera's configuration.
- If the module has two messages we ask about the mission package's configuration.
- If the module has all three messages we proceed to the actual collection of information from the module.

The robot configuration data will give us an overview about robot's maximal ruder, stern plane angles and maximal propeller spin speed. The Camera configuration tells us how much we can zoom in or out and the Mission package configuration provides us with maximal rotation speeds, pan and tilt ranges. Now we can fully operate the actual robot.

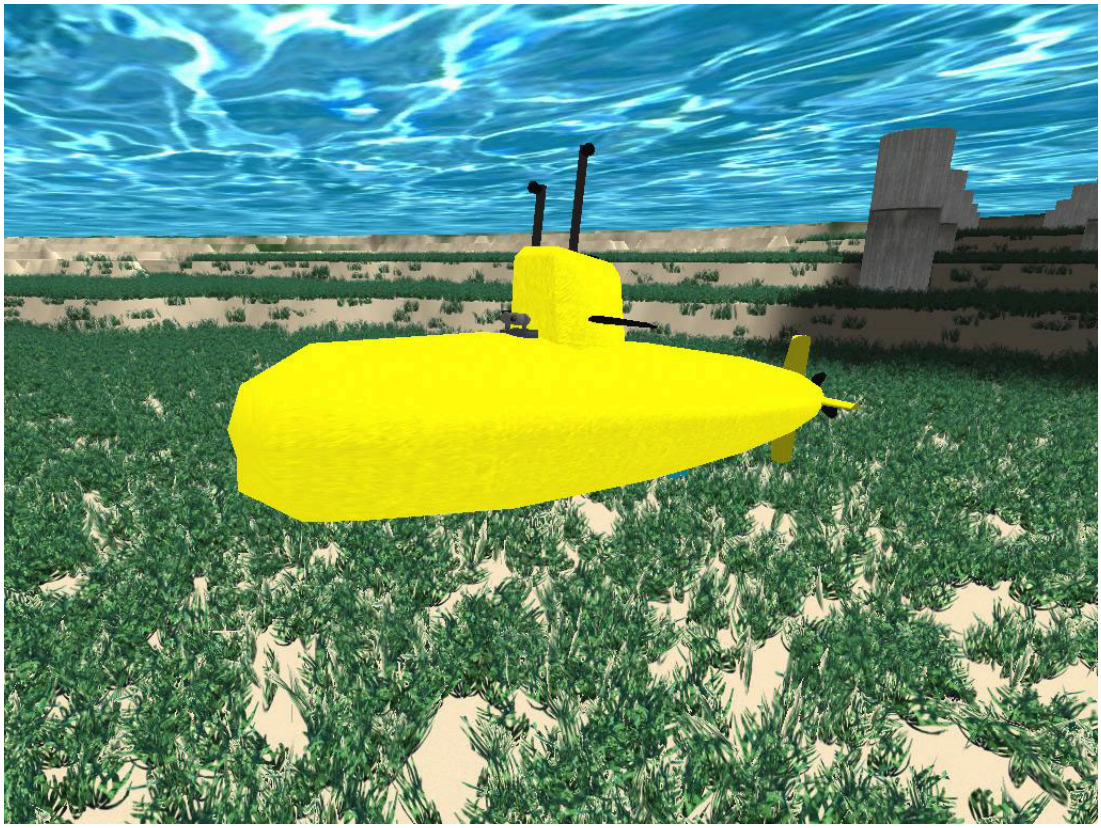


Figure 3.1: Submarine sample robot as Nautical vehicle representative.

We used this sample to show how to control the camera. When the configuration information is received, the camera starts to rotate through mission package pan and since we know maximal tilt bounds, the camera starts to move up and down as well. The camera itself changes the field of view to zoom in and out as it moves on the mission package. Data from camera were not used because it involves launching another component called Image server. This component and its protocol is not supported by our project.

We are unable to construct very complex and meaningful behaviour of this robot, because we have only one sensor to begin with. This sensor can be used to determine how far from the bottom of the sea our robot is. This physical quantity distorts with the intensity of sinking or rising of the submarine as the scanning angle changes and the sensor aims further. This is why we need to be careful with using the maximum values while controlling the robot.

We implemented a one-state regulation that tries to keep the submarine four meters above the bottom of the sea. If it is above this level it makes sure the sub is descending, if it is below this level it tries to ascend. It recognize the progress from the single ray laser sensor's actual and previous value. When the submarine is in situation when it does not need to correct the level it steers by the rudder to one side. Every 100 seconds it changes the turning direction not to spin in a circle. So it either steers by rudder or corrects its level by stern plane. If it finds itself less than half a meter from the bottom of the sea, it tries to reverse in opposite direction.

Obvious disadvantage is an absence of a sensor which would measure an orientation of the submarine. The thing is that if there appears a sudden drop, the

sub starts to submerge rapidly. As a consequence the single ray laser sensor aim is nearly horizontal and it never meets its limits.

3.5.2 ERS - Legged robot

Legged robots are the most difficult robots to control because we have to move different combination of joints each time. As a representative of Legged robots we have chosen AIBO ERS robot shown in figure 3.2. This robot has three joints per leg and its head is also capable of three kinds of motion. It takes a lot of time before one finds a key combination of front and back legs to move the robot in a desired direction. We prepared this robot for the DM-spqrSoccer2006_250 map.

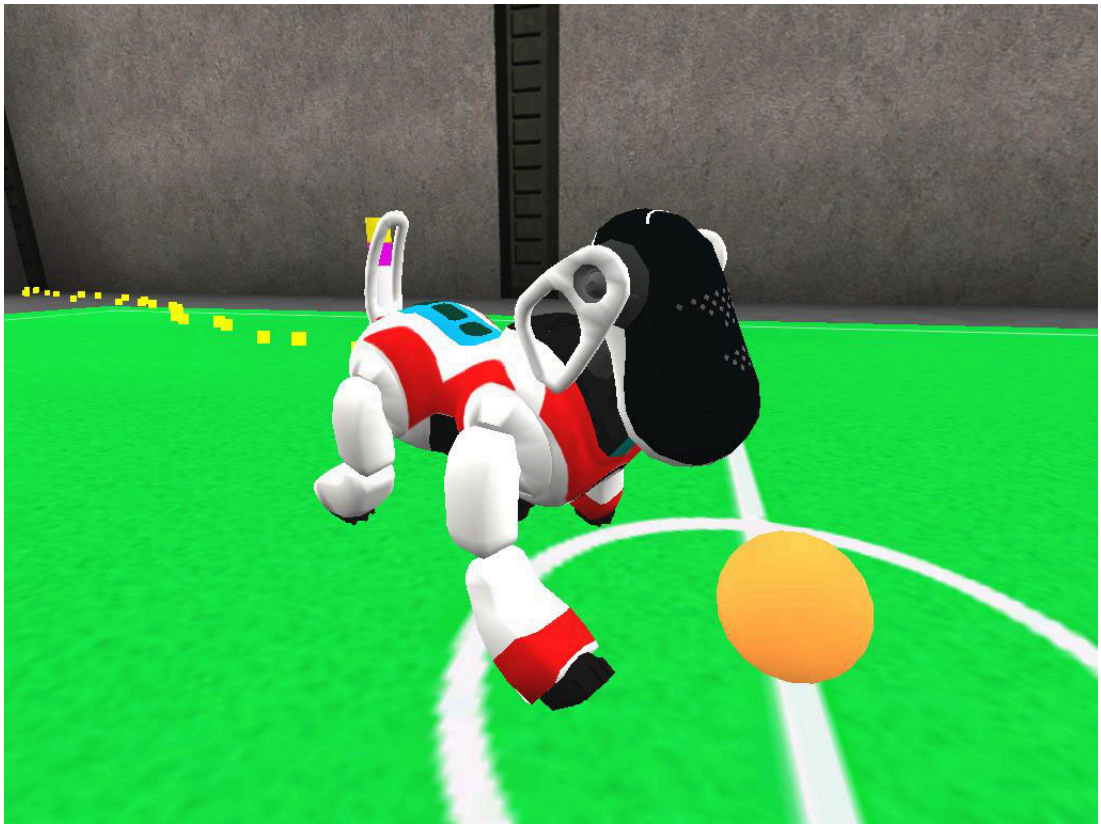


Figure 3.2: AIBO ERS sample robot as Legged robot representative.

We used this robot to demonstrate the MULTIDRIVE command which is capable of moving multiple joints at the same time. We do not use all the sensors this robot has to offer, which is a great opportunity to use Sensor specific module. For our purpose it is sufficient to make the robot walk, and to give it some meaning we let it follow the ball which is an asset of this map. The sensor we use for that is called Helper sensor and it supplies data feed with 2D position, 3D position, estimated radius of the ball from the camera view point and more. We are using 2D position to guide the robot towards this ball if it is visible. Is it is not, the robot walks in circles. To navigate the robot towards the ball in close distances requires to bow robot's head down for the camera to see the ball. When the ball is within reach the robot continues to walk forwards which results in robot kicking the ball accidentally.

We used this sample robot to show how the TRACE command can be used. It only needs the color and interval set to start dropping points into the world and we can see, where was the robot moving.

There is more advanced work dedicated to this type of robot in [14]. It provides a valuable information regarding the Helper sensor, which is not mentioned in the USARSim manual [12].

3.5.3 P2DX - Ground vehicle

We opted P2DX robot as a Ground vehicle exemplar. It is displayed in figure 3.3. On this robot we show that we do not need to use sensor modules, although they come in handy. This robot has a lot of sensors, yet again we use only SICKLMS which is a Range scanner sensor. This sensor uses laser rays to measure distance in front of it in 180 degrees. We hooked a listener to SEN messages and every time Laser ranges are ready, we take the ranges, which are basically 180 double precision numbers and we divide them into thirds. From each third we calculate average value and based on the greatest of these three numbers, we decide how to setup left and right motor speeds. This means that the robot tries to head for the most open direction. If the robot finds itself in a situation when an obstacle is too close in front of it, the robot reverses.

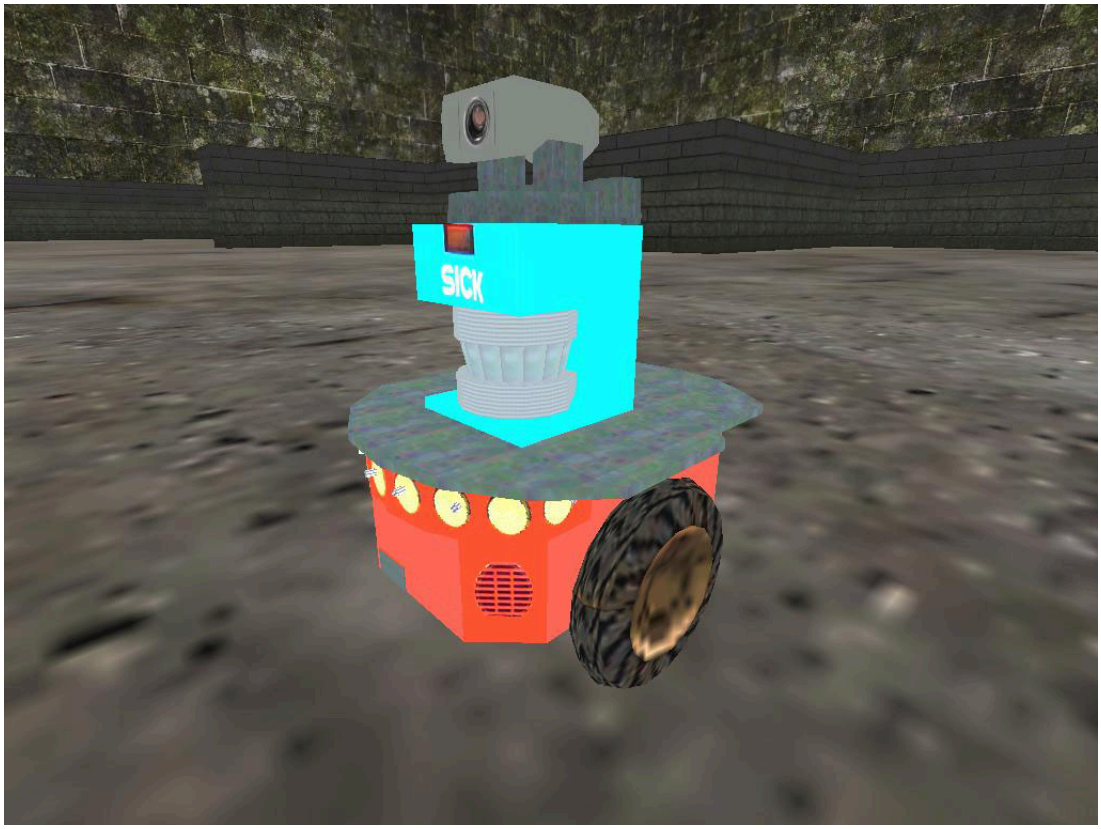


Figure 3.3: P2DX sample robot as Ground vehicle representative.

This robot behaviour is very simple, therefore we produced a POSH version of the same robot with the same behaviour. The POSH version of the robot was controlled by POSH plan displayed in figure 3.4.. This version of P2DX uses Sensor specific module for obtaining ranges from the Range scanner. This example

is handy in understanding how the POSH plans work. We can comprehend this plan as a simple hierarchical list of *if-then* rules processed from the top. The dark blue *Sense* rectangles from this figure can be understood as the *if* part and the *then* part would be expressed by the light blue *Action* rectangles. *Senses* and *Actions* used in POSH plan are implemented in Java. *Senses* return boolean values and *Actions* usually send a command to the robot.

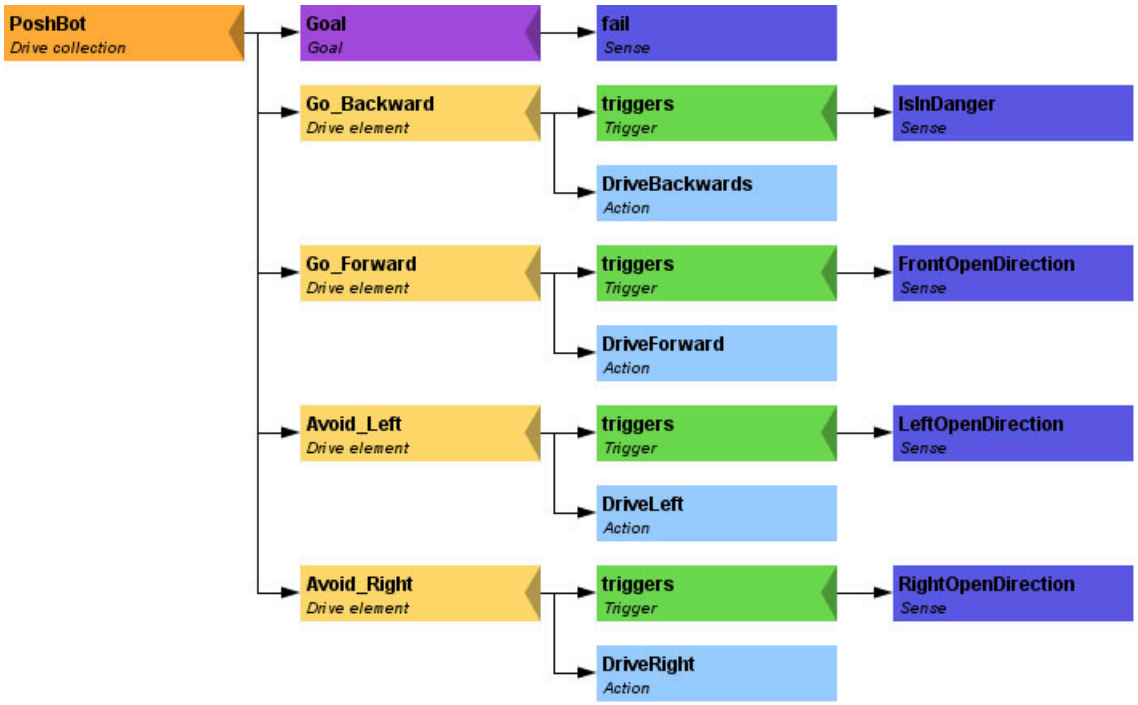


Figure 3.4: P2DX POSH plan causes the robot to drive into the most open direction and reverse if an obstacle is too close.

We can see, how the rules in this simple example are sorted. The first rule is the most important one. The `IsInDanger` *Sense* in fact embodies checking for an obstacle in front of the robot. As we go down we encounter inferior rules. Actually we can say we have only two levels of importance, because there is only one valid rule from the last three at any given point. Each of these last three *Senses* asks about the most open direction and since this information is obtained from the Range scanner by dividing the ranges into thirds and selecting the fittest, there will always be one most suitable direction. The POSH plan structure is analyzed further in the fourth chapter.

4. POSH vs. Java

In this chapter we will go through the solution of more complicated problem in POSH and Java and we compare both approaches. The task was to create an Aerial vehicle that would provide a height map of an area. To do that, it will be given the coordinates of an area to explore. It will use a Range scanner sensor to scan the heights of terrain and it should maintain a reasonable altitude for scanning. The robot will be able to avoid obstacles such as walls or trees not to hit them and fall. The robot will evaluate when to return to its base before it runs out of battery and it will be able to recharge and continue in scanning procedure.

4.1 Solution analysis

The USARSim simulator does not support the possibility of charging the battery. This is why we have devised a way to simulate it. To eliminate the dead battery consequence (the robot shuts down and falls uncontrollably), we changed a battery life to a very high value at the configuration file *USARBot.ini*. Battery capacity is simulated as a constant in our program. The robot supervises the battery status and range, which is continuously calculated from the distance the robot traveled since the last recharge and from the beeline distance of the robot from the base. If battery level drops below the calculated boundary, the robot immediately returns for a recharge. There is a reserve of 15% to allow the robot to avoid obstacles along the way home. It should be noted that the capacity of the battery in the simulator is a sheer number of seconds remaining before the robot shuts down. Also the robot's speed is increased when it returns to its base or when it flies to continue the scanning procedure which helps it slightly extend its battery life.

After careful consideration, we have decided to use following sensors for the AirRobot: INS sensor, Ground truth, Range scanner and a Sonar sensor. We have kept all the sensor's adjustments by default.

Inertial Navigation System (INS) sensor simulates a navigation tool that uses gyroscopes and motion sensors to estimate robots position and orientation in space without the need of external references. This sensor offers dense sampling, but its disadvantage lies in a drift gained over time. It is important to bear in mind that the noise causing the divergences is proportional to the rate of change. We considered using GPS sensor, but its resolution was not sufficient for our purpose of navigating the robot accurately.

The Ground truth sensor provides undistorted information about robot's current location and orientation. We use this sensor only to reset the INS sensor's cumulated error every time the robot lands at the base.

Range scanner uses laser rays to measure distance in 180 degree field of view and its reach is 20 meters by default. We attached this sensor on the robot's bottom part where originally mission package with a camera was mounted. The only modification of this sensor is that we have set the visibility to false in the configuration file. After complex testing, we established the robot scanning altitude at six meters. The Range scanner captures very wide strip and the robot is

resistant to altitude changes of a rough terrain. To be able to fly over ravines and water surfaces there needs to be a level to be a level, below which the robot can not sink unless charging. Water can not be distinguished by the Range scanner and falling into a ravine would cause the robot to be stuck there.

We designed a Sonar sensor layout located on a ring surrounding the robot and protecting the propellers. These sensors are small simple cylinders measuring a distance up to five meters by a single cone beam. Originally we tested the layout with four of these sensors on each side, but we have learned that the two most front ones reported mostly duplicate data, because the most dangerous obstacles occur right in front of the robot. Therefore we designed the final arrangement of Sonar sensors as shown in figure 4.1. The gray triangles illustrate the five meter range of individual sensors that emit a cone beam. The beam angle is fifteen degrees. Furthermore, the figure displays the robot's threat zones that we have arranged. In case there is an obstacle within any of the blue triangles, it means that the robot is in a low risk situation. If an obstacle reaches one of the red triangles, it is considered that the robot is in a high risk situation and that a collision may occur. Otherwise the robot finds itself in no risk situation. The Sonar sensors are located only on the front part because we expect it will not need to fly backwards. The fact that the low and high risk zones are not equally distributed around the front of the robot allows it to better respond to the obstacles that are in collision course. Shorter cone zones on the side make it easier to fly around obstacles, while the longer zones in front leave solid maneuvering space.

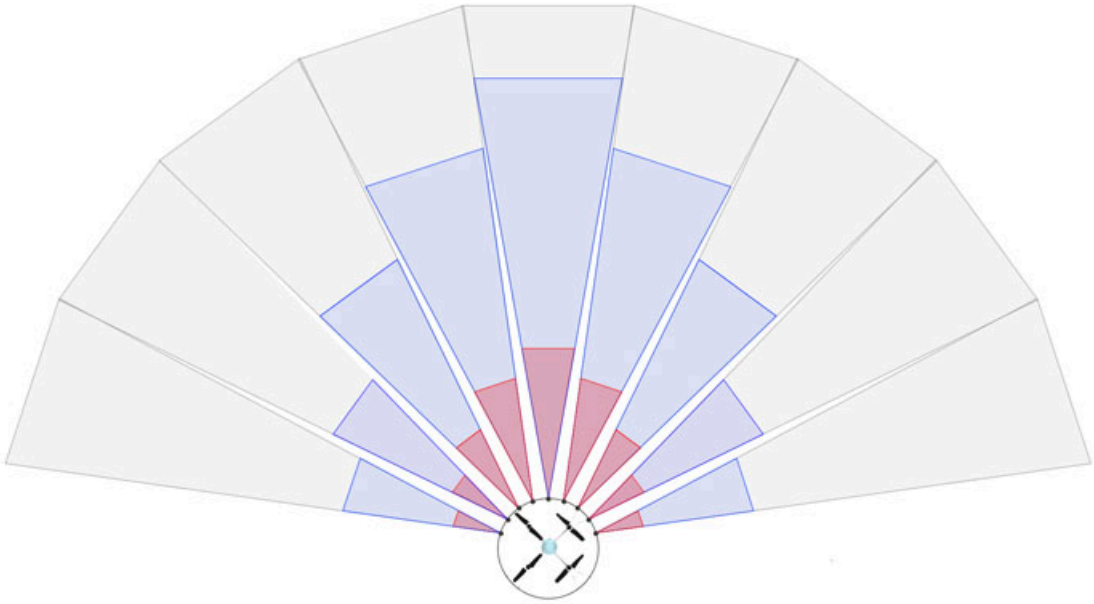


Figure 4.1: Layout of Sonar sensors mounted on the AirRobot with low and high risk level zones highlighted.

Because the INS sensor drifts overtime and more changes cause greater deviation, we have to reduce the amount of movement changes. Therefore we outlined the principal of robot movement to cover the whole scanning area with the least possible change in motion. According to the scan area attributes, the longer side is selected. Based on the minimal density that corresponds with the scan width, it

is calculated how many of these longer side sections will be needed to go through. The robot begins its path from the nearest corner and flies back and forth one step further every time so that it continuously covers the whole territory. In case the input scanning area is too large (either width or height is greater than a constant currently set to 150 meters) it will be divided into quadrants. Each quadrant is scanned separately as if it was smaller separate scanning area.

Every time the robot lands for a recharge or when it has finished scanning one quadrant, the INS sensor resets. This is done by setting Location and Rotation values from Ground truth sensor to INS sensor. Then a battery related variables are reset and all necessary variables are prepared for next take off. The INS sensor can be set only when the robot is not moving. This is ensured by keeping the robot landed on the ground for several seconds.

The robot's resulting behaviour is following. As soon as the robot is spawned into the world, it takes off straight up. After reaching scanning altitude, it heads to the nearest corner of a scanning area. Once it gets there, it slows down to capture terrain by the Range scanner in greater detail and it begins the scanning procedure. It scans the terrain in wide strips to cover the whole area. If the battery level falls below computed threshold, it returns to its base to recharge. After charging is complete, the robot resumes the scanning procedure where it was suspended. If the robot encounters an obstacle, it tries to go around. If it does not succeed within a certain time limit, it will try to chose the following goal. After the scanning procedure is done, the robot lands at the start location.

4.2 Scan preview

For both implementations there is a common utility for displaying the current state of the robot. It is a JForm portrayed in figure 4.2, divided into two parts. In the left white stripe, there appear actual data such as elapsed time, distance from next goal, battery used, range and more. It draws Range scanner rays to capture situation of this sensor at the top. Directly below this the terrain state is drawn and finally nine columns symbolizing the state of individual Sonar sensors are there also. In the second black part of the preview screen, there is the state of robot scanning progress captured. The AirRobot is utilizing this JForm via setters. It transfers sensor data, estimated location and orientation and also important points in space such as landing point, points where the robot got into low or high level risk situations and where it needed a recharge. These points are used in the output image. This JForm has a double-buffered drawing system not to flicker.

At the beginning, a preview bitmap and array of double-precision numbers are initialized. Data from the Range scanner are written to both of these. Shades of gray represent calculated terrain elevations from the sensor in our preview bitmap. Darker shades correspond with depths and lighter ones with heights. It is important to remember that it is a preview, hence the resulting spectrum does not match the color spectrum seen on the preview screen. We are unable to know what the height difference between the highest and the lowest value will be. Despite the slow motion of the robot, the data received from the sensor are fairly sparse, for we are given one dimensional list of figures. Thus it makes approximation between every two beams and between every two sensor readings.

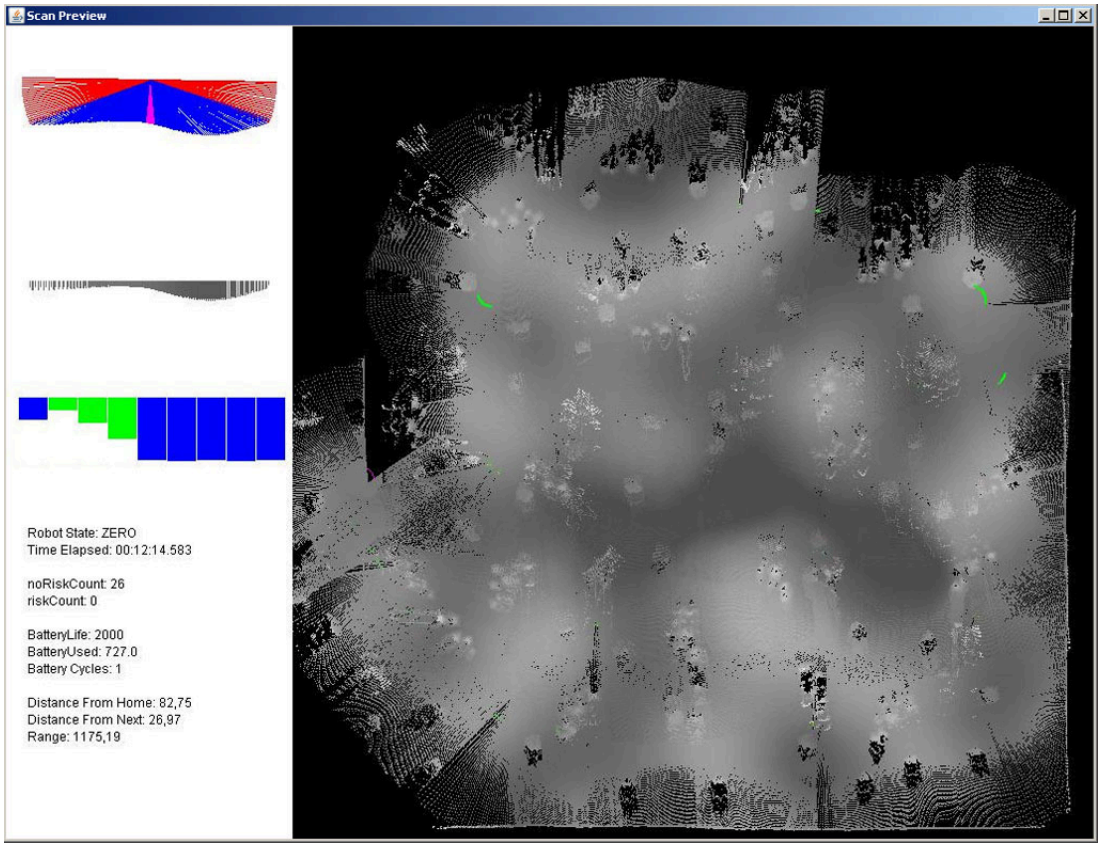


Figure 4.2: Preview of the JForm reporting status of the scanning procedure.

Each approximation is calculated as the average of two consecutive figures. This gives us more coherent overview of scanned region. If there occurs a situation that involves overwriting a point previously recorded, it only alters it to an arithmetic mean of both old and the new values. This applies only when writing data to the array of heights, otherwise the preview bitmap's shades of gray are always redrawn by new values.

The size of resulting map can not be predicted (the robot might be getting broadly around large obstacle near the edge), so we designed a mechanism that resizes both preview bitmap and the array of heights at the same time as soon as the robot reaches one of the borders. To view larger landscapes, user can shift the preview bitmap by arrows on a keyboard.

After the scanning is done or if the robot is unable to finish the scanning procedure, it lands on its base and we have to manually terminate the program. If terminated correctly a new image appears in *USAR_Scan* folder on users Desktop. This picture captures the progress of the scanning procedure. Along with this picture one additional file is provided with raw height data and it can be further processed by another software.

4.3 POSH implementation

As mentioned before, POSH reactive planer is for controlling behaviour of virtual agents and POSH plan is a hierarchical structure for action selection. In POSH plans *Acts* and *Senses* are the leaves and in Pogamut they are implemented as

separate classes that both share a Context class in which we define modules and methods we would like to use in either one of them. There are aggregations for combining these primitives. *Drive collection* is the root of POSH hierarchy that determines action selection each cycle, *Competences* are basic reactive plans and *Action patterns* are simple sequences of actions. *Competences* work on a similar principle as the *Drive collection*. It is actually a nested hierarchical tree of *if-then* rules. There is a *Goal* as a first element in a *Drive collection* that triggers its *Sense* before any *Drive element* does. If the sense is successful the program terminates. We do not use this feature in our project. The sense returns always false, thus our program is infinite.

The POSH plans are evaluated from top to bottom. Every iteration the first *Drive element* in POSH hierarchy is tested and if its trigger does not succeed (Sense does not comply) it moves to the next *Drive element*. If it does succeed it executes its body which can be either *Competence* or an *Action pattern* or a simple *Action*. We will describe our *Drive collection* from top to bottom to help explain the implementation of the behaviour of our autonomous robot. The POSH plan for AirRobot is displayed in figure 4.3.

The most important thing is to launch the robot and for that we need to obtain configuration and geometry data. As mentioned in section 3.3 Modules, we can not simply send all queries at the same logic call. By the POSH plan we can actually solve this problem. We used `RobotLaunch` (the first *Drive element*) in combination with a *Competence* to list all the requirements before the launch is possible. These requirements involve the scan preview to be initialized to write data to, than we need the robot configuration and sonar geometry data and finally we need initial sensor data received. Each iteration, the planner resolves one of these requirements and as soon as the last one is processed, we consider this *Drive element* to be finished and its triggering sense will always return false from now on, effectively skipping this *Drive element* in the future executions.

The `RobotTakeOff` element is the next *Drive element* in our hierarchy and it contemplates the take off of the robot, so it reaches six meter altitude convenient for scanning. This is immediately followed by the `batteryWatch` element responsible for checking the battery level. When this element is triggered, there rises a necessity to treat the situation of instant interruption of scanning procedure differently at different scenarios. In case the robot failed to establish previously interrupted scanning procedure, it is clear that it will not be able to do so the next time. Therefore it terminates the scanning procedure prematurely. The correct handling of other situations depends on whether the robot is avoiding an obstacle or freely flying. All of the above states are summarized in a *Competence*.

Another mildly less important *Drive element* is the *stuckWatch* element. It monitors the time the robot is trying to avoid some obstacle. It may happen that the computed goal, which is being followed by the robot, ends up in an inaccessible area such a house or an impenetrable group of trees might represent. To prevent the robot from spending eternity encircling such an obstacle, we declare this goal unavailable after a long time trying and we proceed to the next in line.

The `sonarWatch` element handles obstacle avoidance. The robot behaves according to the risk zone it finds itself in. We introduced the risk zone layout in figure 4.1. If the robot is in the high risk zone, it evades by sudden movement in a direction that is exactly opposite to the location of Sonar sensor with the

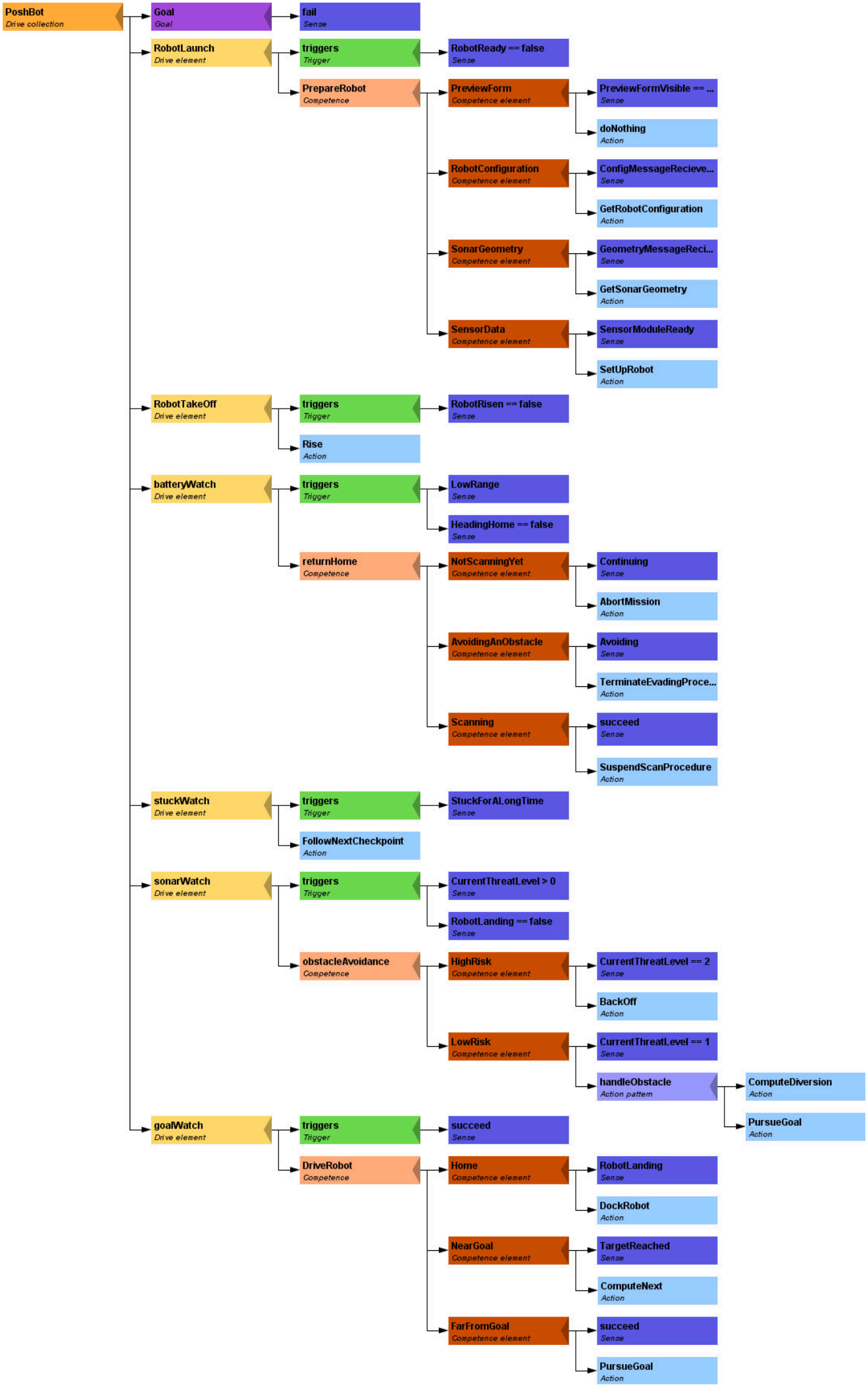


Figure 4.3: POSH plan for reactive behaviour control of AirRobot.

smallest distance measured. In case the robot is in low risk situation, it stops the forward motion, it takes the location of sonar that constitutes the greatest threat and it calculates a point in the most free direction to divert its path. In case the robot is in no direct danger the planner does not trigger this element.

The `goalWatch` element is the last *Drive element*. It is triggered when all of the above fail. This means that the robot has to have gathered all important data, it has to be in the right altitude with enough battery remaining and with no obstacle ahead. This element makes sure that the robot has always some goal to pursue. It also lands the robot when it is returning to its base.

4.4 Java implementation

As a different architecture we chose Java, which is a common programming language. Our primary goal was to authenticate the suitability of utilizing POSH in prototyping autonomous robot behaviour within USARSim. We wanted to know if the usage of POSH planner will provide any advantages. By using other different architecture, we would not be able to be sure whether any implementation error was caused by POSH or the other architecture. This makes Java safe and obvious choice.

The solution is implemented just by using the modules and logic calls triggered by STA message. The logic consists of several control blocks for simple organization of filling the scan preview with robot related data, processing *Sensor* and *Response messages* and for gathering information about robot at the beginning. In the main control block there are several important procedures taking place one by one. First it checks the altitude and Sonar sensor inputs than a direction of the robot respective to its goal. All of these factors are considered in setting up new robot flight direction. After the direction is set it checks if the robot has reached its goal or if charging procedure is complete, eventually if the robot should proceed to next quadrant, all depending on the state of the robot. The last thing to verify is the robot being jammed.

We designed a compact set of states, according to which the robot behaves. There are states for the back and forth style flying, where long straights follow the shorter ones until a scanning area is covered. Besides these, there is a state for a take off and for dodging obstacles and charging. Each state is associated with a goal used as a checkpoint for which the robot is always heading. Every time the robot gets within an acceptable distance from desired goal, it computes the next state in line according to which next goal is determined. Every state has its successor, and the process of getting the next state in line could be arbitrarily suspended by obstacle avoidance or by a need to charge the battery. In both cases, the current state and goal are stored in temporary variables while processing these interruptions.

When low risk situation occurs by the robot being close to some obstacle, the diversion point is calculated. The sonar with the greatest threat is picked and according to sum of both left and right side sonars from this one, it computes the diversion point in a direction that is presumably free of obstacles. This direction is computed from a position of the first risk free sonar from the free sonar segment.In case the goal direction is further than the computed one, the threat is ignored. This could happen after cornering or by near missing an obstacle that got into low

risk zone only at the edge sonar. Computed direction would only deflect the robot pointlessly for a moment to circle around an obstacle. A threat is also ignored if the goal is closer than an obstacle. When the robot is evading it does not change its altitude not to run into an obstacle directly above or below the robot. It mostly does not reach the computed diversion point. This point is used only to pull the robot away from danger. After a few iterations of not being in danger it switches for its former goal not to waste time heading away. We achieved similar techniques by using POSH *Competences* in previous implementation.

4.5 Comparison of both

Both implementations solve the same problem, they share substantial amount of the same code to make sure the comparison can be detached. It could happen that both implementations may be very different. After all, two executions of the same solution do not necessarily give the same result due to sensor noise and especially the drift of INS sensor. For example scanning of *DM-TallTestWorld_250* map which is about two hundred square meters may take five to fifteen minutes longer than another run and the robot might use different number of battery charges, too. Figure 4.4 shows final output image of this map. It is important to realize that the logic call in Java implementation triggers every time a *State message* is received. Although the POSH plan evaluation is triggered on the same impulse, it has notable overhead and does not directly correspond with the logic calls. This means that some routines, especially ones that involved iteration counting did not match and had to be altered. This fact actually occurred when tweaking sample robots. It may be because the robot moves so slowly, that the difference between logic calls and POSH plan evaluation is not that notable.

POSH implementation handles obstacle avoidance slightly differently which does not affect the avoiding concept. Overall, it turned out to produce nearly identical behaviour, which proves that utilizing POSH planner does not restrict usage of USARSim.

The advantage of using POSH planner was found in clarity and simplicity. It is crucial to understand the principal of evaluating plans which ultimately provides very profitable tool in which we can prototype behaviour of a robot rapidly. This was observed during development of POSH version of P2DX sample robot mentioned in 3.5.3 as well. POSH plans are easily modifiable through the GUI. We can take *Actions* and swap them with each other. Same goes for *Senses*. It is very fast, yet critical intervention that can be beneficial in some cases.

4.6 Results

The AirRobot scanning speed is very slow due to Range scanner low sample rate and resolution. It turned out that the robot has more time to avoid obstacles, on the other hand, testing is therefore very time consuming.

The *DM-TallTestWorld_250* map has many tree obstacles and it is a large area. The scanning of this map takes about one and a half hours. Observations showed that the robot does not complete the scanning procedure one time out of ten due to a collision with a tree. We have successfully completed more than

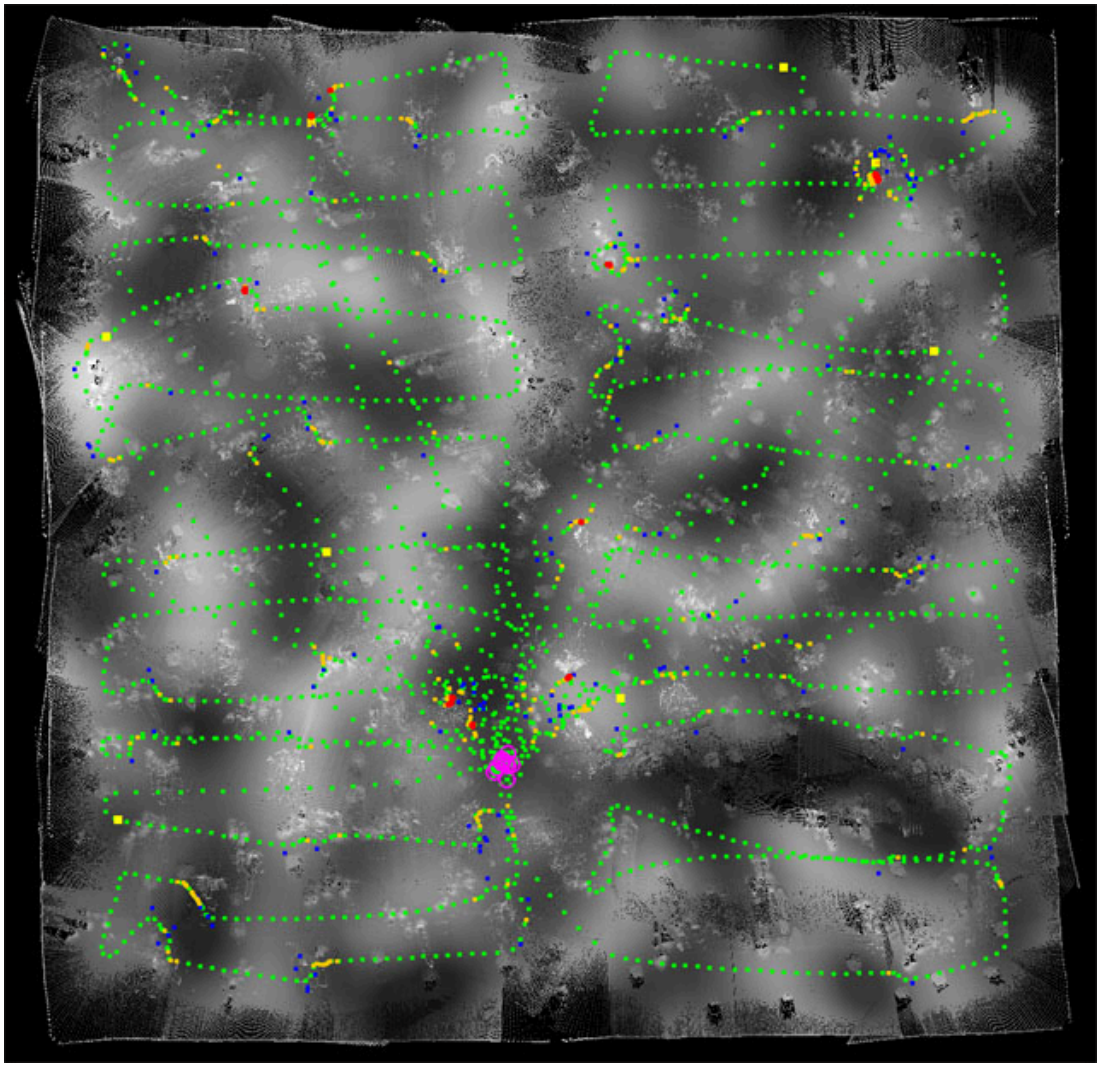


Figure 4.4: Scan output image example of DM-TallTestWorld_250 map. Green dots represent robot's path, blue are diversion points, orange represent Low risk situations, red are High risk situations, yellow squares are points where a recharge was needed, and violet circles stand for landing points.

fifty launches. Situations that caused the AirRobot to fall were observed with the same frequency regardless of the type of implementation. It was mainly caused by incomplete knowledge of surroundings. The robot gets itself into a narrow alley where it turns back, but if it encounters high risk situation, it may reverse into an obstacle and fall. Very rare was a situation where the robot was descending due to lowering of terrain and it touched the tip of a tree with its bottom part and fell.

Another testing was performed in a closed room of *DM-VMAC1* map. We let the robot to try to fly beyond the walls. The robot was flying along the walls but never fell down due to a collision. We let it fly around the room for several hours.

We created an ultimate test for the AirRobot that covers nearly one square kilometer at *ONS-PointLookout_250* map, which is beyond the current robots capabilities. Thus we used a Ground truth sensor instead of INS sensor to ensure

that a proper path is taken by the robot and we increased a battery life constant. As a result we experienced nearly nineteen hours long run that proved the memory space assigned by the scanning preview not to be too excessive even for such an extreme scanning. Important outcome of this test was that it does not influence the robot's behaviour performance.

5. Discussion

This chapter sums up problems and bugs experienced within USARSim, Pogamut and the ones in POSH. We look at lessons learned from integrating USARSim with Pogamut and from robot implementations. Also future work is mentioned.

5.1 Errors encountered

There are known issues with the POSH GUI editor. When creating new *Drive element*, default *Action* and *Sense* are generated with wrong names that prevent the plan from running. We have to replace both primitives with *Actions* and *Senses* from the toolbox.

Another issue observed numerous times was that after a build it spawned a robot into the environment, but it immediately vanished. The robot was mentioned in a scoreboard but was no longer present. New run solved this problem every time. We were not able to disclose the error rising this behaviour. It is important to be aware of the amount of memory the AirRobot program causes to allocate. Especially the scanning preview JForm has a lot of data to process and to store to be able to create the final output. It is understandable that the more time a scanning takes the more space it requires, for it has to store more data such as the robot path points and the dynamic bitmap and array that hold recorded heights.

An inconvenience caused by USARSim was the inability to recharge the battery. We had to implement our own system, that does not anticipate a scenario in which the robot is unable to return to its base for a recharge in time. We do not have an instrument which would make the simulator to induce the fall of an aircraft. As a matter of fact, in most cases, the robot had enough reserve to travel to the base, even in rough conditions where it had to dodge numerous obstacles on its way there.

5.2 Lessons learned

We have experienced a need to add a support of a new sensor when everything has been already created. We encountered Helper sensor that estimates a position of a ball from AIBO's camera. It was not mentioned in USARSim manual [12]. We observed what segments are appearing in a new *Sensor message* and we added corresponding properties to the mutual *elementary structure* used by info message parser. Next thing needed to fully support this sensor was to create a *small structure* for this particular sensor and add its reference to an enumeration type representing the list of supported sensors. It was the simplicity of mutual structure design that made it possible to enhance modules that easily.

The Sonar sensor layout proved to be fairly reliable even though it is composed from only nine conical beams. The robot does not have a complete awareness about what is situated around it. Thus it is possible that the robot gets into a high risk situation surrounded by bushes or trees in which it might reverse into an obstacle and crash. We came across these situations rarely so that we have

never been able to recreate the situation twice in a row.

It is important to realize that POSH plans are not able to control everything. It ought to handle reactive behaviour of the robot. Despite this, we used it to handle obtaining configuration and geometry data and to make sure that our preview JForm is initialized. It did not hold any benefit back, in fact it facilitated launching of the robot.

Implementation of POSH in Pogamut offers a method triggered before the first *Action* call and a method triggered after this rule is no longer valid. These methods are convenient for preparations and cleanups. Although it provides substantial profit, we have benefited rarely from these methods.

5.3 Future work

We provided an instrument for creating autonomous robots. It is possible to extend this tool to support a Communication station which can be spawned into USARSim world. This would serve as a basis for multiple robot support to create cooperative robot systems.

World controller is also a spawnable item that allows to create, move and delete objects dynamically during the simulation. The support of this item would allow us to make distinctive scenarios in which robots could be tested in more thoroughly.

Most robots have a camera mounted on mission packages. To support image processing a support of Image server is possible to implement. This would include a new TCP/IP connection and mastering very simple protocol.

Conclusion

We have created a plugin for Pogamut called PogamutUSAR2004 that enables fast prototyping of autonomous robots in USARSim platform. This involved a) creating a simple expandable system for parsing info messages into elementary structures, b) wrapping USARSim commands into Java objects, c) creating a set of modules offering simplified access to important properties, and d) creating classes integrating USARSim agent life cycle into Pogamut. Subsequently, we provided a set of simple robots as a proof of concept, each representing different type of robot and each demonstrating different control mechanisms.

Furthermore we specified more complicated problem and used POSH reactive planner for solving it. We used raw Java implementation of the same problem as a baseline for comparison of both approaches. AirRobot which is an Aerial vehicle was used for scanning terrain using a Range scanner. We designed a layout of Sonar sensors to give the robot capability of avoiding obstacles. As a consequence the robot is able to reliably evade a group of trees and minor obstacles by flying around them. The robot maintains optimal altitude for scanning. Its output is a height map of specified area.

We have found both pure Java and POSH equally capable in managing the reactive behaviour. POSH is suitable for rapid prototyping of complete complex agents and proved to be convenient for smaller beginner projects as well.

Bibliography

[1] Carpin, S., Lewis, M., Wang, J., Balakirsky, S., Scrapper, C. USARSim: a robot simulator for research and education. In: *Proceedings of the 2007 IEEE Conference on Robotics and Automation*, 2007, pp. 1400–1405. URL: http://usarsim.sourceforge.net/

[2] Zaratti, M., Fratarcangeli, M., Iocchi, L. RoboCup 2006: Robot Soccer World Cup X. In: *A 3D Simulator of Multiple Legged Robots Based on USARSim*, LNAI 4434, Springer-Verlag, 2007, pp. 13–24. URL: http://www.robocup2012.org/

[3] Miklic, D., Bogdan, S., Kalinovcic, L. A control architecture for warehouse automation - Performance evaluation in USARSim. In: *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 2011, pp. 109–114. URL: http://vma-competition.com/

[4] Balakirsky Stephen. *Robot development in UT3* [online]. 2011 [cit July 10, 2012]. Availible from: http://usarsim.sourceforge.net/wiki/index.php/Robot_development_in_UT3

[5] Bryson, Joanna J. *Intelligence by Design.* PhD Dissertation: Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science. 2001.

[6] Bryson, Joanna J. The behavior-oriented design of modular agent intelligence. In: *Proceedings of the NODe 2002 agent-related conference on Agent technologies, infrastructures, tools, and applications for E-services*, Springer-Verlag, 2003, pp. 61–76. URL: http://www.cs.bath.ac.uk/~jjb/web/bod.html

[7] Brom, C., Gemrot, J., Bída, M., Burkert, O., Partington S. J., Bryson, J. J. POSH Tools for Game Agent Development by Students and Non-Programmers. In: *Proceedings of CGAMES 06*, Dublin, Ireland, 2006, pp. 126 – 135. URL: http://amis.mff.cuni.cz/pogamut/tiki-index.php?page=POSH+GUI

[8] Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., Brom C. Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In: *Agents for Games and Simulations*, LNCS 5920, Springer, pp. 1–15. URL: http://pogamut.cuni.cz

[9] Scrapper, Ch., Balakirsky, S., Messina, E. MOAST and USARSim: a combined framework for the development and testing of autonomous systems. In: *Unmanned Systems Technology VIII*. Proceedings of SPIE, 2006. URL: http://moast.sourceforge.net/

[10] The Player Project *Player manual* [online]. 2011 [cit July 10, 2012]. Availible from: http://playerstage.sourceforge.net/

[11] Blank, D.S., Kumar, D., Meeden, L., Yanco, H. The Pyro toolkit for AI and robotics. In: *AI Magzine*, Volume 27, Number 1, 2006. URL: http://pyrorobotics.org/?page=Pyro

[12] Wang, J., Balakirsky, S. *USARSim manual* [online]. 2009 [cit July 10, 2012]. Availible from: http://usarsim.sourceforge.net/wiki/index.php/UT_2004_Manual

[13] Klein, Gerwin. *JFlex manual* [online]. 2009 [cit July 10, 2012]. Availible from: http://jflex.de/manual.html

[14] Zaratti, Marco. *Legged Simulator Usage* [online]. 2006 [cit July 10, 2012]. Availible from: http://digilander.libero.it/windflow/index.htm

CD contents

The compact disk included with the thesis has following structure:

- `Documentation` - Folder with generated Javadocs for PogamutUSAR2004 project and every sample robot.
- `Robot_Executables` - Folder with **.jar* files, each representing one built sample robot.
- `Sources` - Source files of PogamutUSAR2004 project and every sample robot.
- `USAR_Scans` - Folder with some of previously scanned areas (test results).
- `USARBot.ini` - File needed after installing USARSim. Intended for overwriting original version.
- `Content.pdf` - Content of the compact disk in PDF format.
- `Technical_documentation.pdf` -
- `Thesis.pdf` - The electronic version of this thesis.
- `User_documentation.pdf`
- `AirRobot_POSH_Plan.png`