Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Matúš Gažo

Behaviour-based Control of Autonomous Robots

Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc
Study program: Informatics, Programming

2009

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Matúš Gažo

Riadenie autonómnych robotov založené na chovaní

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. František Mráz, CSc.
Studijní program: Informatika, Programování

2009

Poďakovanie patrí v prvom rade vedúcemu tejto práce RNDr. Františkovi Mrázovi CSc., bez ktorého nekonečnej trpezlivosti a pomoci by tu tento text nebol, a takisto mojim rodičom, bez ktorých by som tu nebol ja. Zásluhy má i nekonečná studnica vedomostí – Google a open-source komunita, ktorá v niektorých miestach vôbec umožnila, aby sa táto práca pohla ďalej. V neposlednom rade kolega, spolubojovník a motivátor Jaroslav Fibichr vytvoril krásne ilustrácie za čo mu ďakujem.

Názov práce: Riadenie autonómnych robotov založené na chovaní
Autor: Matúš Gažo
Katedra (ústav): Kabinet software a výuky informatiky
Vedúci bakalárskej práce: RNDr. František Mráz, CSc.
e-mail vedúceho: mraz@ksvi.mff.cuni.cz
Abstrakt: Cieľom bakalárskej práce bola implementácia frameworku v jazyku C++, ktorý bude slúžiť na kompaktnejší vývoj umelej inteligencie (UI) autonómnych robotov založenej na chovaní (behaviour-based robotics). Táto UI je založená na pravidlách, parametroch a rôznych "chovaniach" robota, v našom prípade vopred definovaných heuristicky. Platforma umožňuje posunúť ťažisko vývoja takejto UI na definovanie týchto chovaní a pravidiel namiesto programovania. Súčasťou platformy sú taktiež genetické algoritmy, ktoré umožňujú optimalizáciu parametrov heuristického riešenia.
Funkčnosť konceptu bola overená na konkrétnom simulovanom pokuse s viacerými robotmi, ktorým bola zadaná jednoduchá konštrukčná úloha. Všetky simulácie boli uskutočnené pomocou simulačného softvéru Player/Stage.
Kľúčové slová: robotika založená na chovaniach, robot, genetické algoritmy, simulácia

Title: Behaviour-based Control of Autonomous Robots
Author: Matúš Gažo
Department: Department of Software and Computer Science Education
Supervisor: RNDr. František Mráz, CSc.
Supervisor's e-mail address: mraz@ksvi.mff.cuni.cz
Abstract: The aim of this bachelor's thesis was an implementation of a library, which should enable to develop controllers for autonomous mobile robots. The controllers are based on ideas of Behaviour-based robotics. This means that such a robot has a set of elementary behaviours and its control system consists of a set of parametrized rules, which decide according to the state of its sensors which one of the pre-defined elementary behaviours the robot has to use. This program shifts the substantial part of a robot control development from programming to defining such elementary behaviours and sets of rules. It includes a genetic algorithm that can help to optimize parameters of the rules used and/or the behaviours used in a controller.
As a proof-of-concept a simulated experiment with multiple agents was conducted, in which the robots were assigned a simple construction task. All simulations were done in the Player/Stage simulator.
Keywords: behaviour-based robotics, robot, genetic algorithms, simulation

Contents

Chapter 1

Introduction

This work, the Bebal-library, focuses on a simplified design of behaviour based controllers for autonomous robots. For non-involved readers, an autonomous robot is an intelligent machine capable of carrying out tasks without the help of humans. The most significant part of an autonomous robot, without regard to its form, size or capabilities, is his controller, that gives it the ability to respond to changes in its environment perceived by its sensors. It is possible to refer to the controller as to artificial intelligence (AI). However, only a specific subset of all possible robotic AIs is implemented in this library, the so-called Behaviour-Based AI (BBAI), which is based on a modular heuristic decomposition of intelligence and results in displaying and clear identification (in the controller program code) of human-like behaviour, e.g. wander around, find object, return to base etc.
A behaviour-based controller consists of a set of pre-defined behaviours and a mechanism that decides which behaviour should be currently the active one. The decision is done solely on the sensor input of the robot and the entered rules for when to trigger a specific behaviour. The Bebal-library takes care of the essential part of a controller and is capable of understanding behaviours and rules defined by the user by extracting them from text files. Therefore it shifts the focus from classical programming to defining such behaviours and rules. Furthermore it is capable of optimizing the controller with the use of genetic algorithms, if possible or desired. These were also the reasons for writing this bachelor thesis.

In order to show the functionality of the Bebal-library an experiment has been conducted. Multiple homogenous (uniform) robots in a virtual environment (a simulator) were given a construction task. The controller was built only by using the library, at first only with human heuristic input, in the second step genetic algorithms are used to increase performance and to show the usability of this library.

This thesis is divided into seven chapters. In the first, a short introduction to behaviour-based robotics is made, the second gives an overview of available robot simulators and gives insight to genetic algorithms and other theoretical background. The third chapter describes the final solution and implementation of the Bebal-library and explains some algorithms used in it. The fourth and fifth chapter contain a user manual and a programmer's reference, that together with attached examples give a manual on how to use the library. Finally in chapter six the experiment is evaluated. Ending with chapter seven, conclusions are made and possible follow-ups are listed.

Chapter 2

Research

Discussed in this chapter is software that is similar to the Bebal-library or could be used as a part of it. Furthermore, necessary terms and concepts used are explained.

2.1 Overview of available Software

The Bebal-library in its current form is tightly coupled with the Player/Stage robot simulator [1] [2]. This was however a design choice as other software is also available.

2.1.1 Enki

Enki is a robot simulator able to simulate large groups of robots. It provides collision and physics support for robots on a flat surface and faster than real-time simulations. Advantages of Enki are also that it is open source and written in C++, the preferred programming language for the Bebal-library. Various hardware robots are supported, currently implemented is the Khepera, ePuck, Alice and S-Bot robot.

2.1.2 Pyro

Pyro is short for Python Robotics and as the name suggests it is a library written in Python, an interpreted programming language. It focuses on providing a simple environment for experimenting with artificial intelligence and robotics. The low-level hardware and software beneath it is shielded from

the user, but Pyro offers an interface through which it can be programmed in Python. Pyro includes a robot simulator but uses also third-party simulators such as Robocup Soccer, the Khepera simulator and even Player/Stage. Nevertheless it is able to control a variety of hardware robots.
The Bebal-library stands partially in competition with Pyro as it offers methods for evolutionary robotics (see 2.4) too, however the main goals of the projects differ.

2.1.3 Breve

Breve is a free software package that makes it easy to build simulations of artificial life. Users can define behaviours of multi-agent systems in a 3D world with Python or a simple language called steve. It also features plug-ins for evolutionary computation.

2.1.4 Player/Stage

Player/Stage is a project for researching robots and sensor systems. It consists actually of two standalone parts, Player and Stage, but they are developed simultaneously and are complementary to each other.
Player is a server for robot control, providing an interface to robot sensors and actuators over the network. It does not distinguish between simulated and real hardware robots. Stage is the backend for Player (amongst various others such as Gazebo) and simulates a population of mobile robots in a two-dimensional environment along with their sensors and other objects. Therefore Stage is responsible for the simulation of the robots interactions in an environment while Player provides an interface for controlling the robots actions.
Player/Stage was selected as the cornerstone for the Bebal-library because of the projects maturity, software stability and simplicity as Player/Stage offers only plain functions for robot control and simulation.

2.2 Background

A variety of robot simulators does exist (see previous section) but only a few – if any – are designed for a high-level abstraction programming of their controller. The original motivation for creating the Bebal-library was to choose a slightly different approach for creating robot controllers which had to fulfil

several requirements:
Create an easy, efficient and clean way of defining a robots controller with the ability to parametrize all vital variables of it. The definition has to be hardware independent and abstract while preserving the possibility of low-level access for finer control. Also should be already existing software used as much as possible (re-inventing the wheel is almost always a bad idea. . .). And finally follow the principles of Evolutionary robotics by providing functionality of Genetic algorithms to further optimize and evaluate the parameters or the created controller.
All this comes not without a prize, which in our case is the limitation to be able to create only behaviour-based controllers and the obligatory use of Player/Stage.

The following sections will expand the theoretical background on topics needed to take advantage of the functionality the Bebal-library provides.

2.3 Behaviour-based Robotics

Behaviour-based robotics is a branch of robotics that utilizes BBAI (Ch. 1) and is heavily based on reactive systems. "A reactive robotic system tightly couples perception to action without the use of intervening abstract representations or time history" is the definition from R.C. Arkin [12]. In addition he also assigns four main characteristics to reactive systems and hence to Behaviour-based robotics:

1. "Behaviours serve as the basic building blocks for robotic actions."
2. "Use of explicit abstract representational knowledge is avoided in the generation of a response."
3. "Animal models of behaviour often serve as a basis for these systems."
4. "These systems are inherently modular from a software design perspective."

This definition requires no enhancement as it captures all important traits of Behaviour-based robotics and is still valid.

A typical behaviour-based robot controller is composed of behaviours and

the corresponding thresholds for the sensors. By comparing the thresholds and the actual sensor readings it determines which behaviour is executed. The determination process can differ based on what type of a behaviour-controller is used. We can basically distinguish controllers that are organized vertically or horizontally.
In controllers with a horizontal alignment of behaviours the behaviours are equal to each other. Then in each controller loop all behaviours are evaluated and assigned a "truth" value based on the extent the sensor readings meet the defined thresholds. The truth-value ranges so from 0 (sensor readings do not match at all) to 1 (perfect match). The resulting actions are blended together from all behaviours but weighted with the truth-value of each behaviour.
On the other hand, vertically aligned controllers prioritize behaviours. The first behaviour defined in order has the highest priority, the last one has lowest priority. The controller loops then through the behaviours ordered by their priority starting with the highest. If the sensor readings match the defined thresholds the behaviour is executed, otherwise it evaluates the behaviour with the nearest lower priority. The Bebal-library uses this organization of behaviours for its controller.

A good example of a behaviour-based controller is a controller that contains only two behaviours:

- Avoid obstacle: If the robots sensors detect an obstacle it is avoided by changing course.
- Wander: When the path in front of the robot is clear it just moves forward.

This very simple controller ensures that the robot avoids any detected object without any knowledge of the environment he is situated in. Behaviour-based robotics is suited for situations and tasks where fast responses from the robot are required. It however has limitations in planning or coordination with other robots and some tasks may be unsolvable with only a behaviour-based AI.

2.4 Evolutionary Robotics

Evolutionary robotics (ER) is a methodology used to develop robot controllers based on ideas of natural evolution and Darwin's theory – the fittest survive [11]. For this purpose, genetic algorithms are used as they provide functions to simulate a population of individuals, in which each individual can be understood as a setting of the robot controller.
Alternatives for robot learning are artificial neural networks or reinforcement learning.

2.5 Genetic Algorithms

Genetic algorithms (GA) are basically a heuristic method of finding solutions to complex problems, that have no exactly defined solving algorithms. GA, or in general all evolutionary algorithms are based on evolutionary principles as seen in nature and use methods observed in genetics – heredity, selection, crossover and mutation – in order to find or improve the initial solution.

The genetic algorithm starts with a population of randomly generated individuals and happens in generations. Each individual is defined by a chromosome and represents a solution of the given problem. The chromosome contains the individuals genes, typically encoded in binary numbers, but other notations are possible (a matrix, strings, etc.). In each generation, the fitness of every individual in the population is evaluated. The fitness is computed by the so-called fitness function which estimates the quality of a solution/individual. Based on their fitness individuals are stochastically selected to form a new generation in a process called Selection. This is motivated by a hope, that the new population will be better than the old one. The selected individuals are also altered by applying Crossover and Mutation methods that cause random changes to the genetic pool. This new population is then used in the next iteration of the algorithm. A final condition determines when the GA ends. Commonly this condition is the number of generations the GA is allowed to work. Occasionally a fitness value can be set and the algorithm proceeds until this value is reached, but it may happen that such a solution is never found.

The following pseudocode illustrates how the GA proceeds:

```
create initial population
while final condition not met do
  evaluate all individuals and assign each a fitness value
  while not enough individuals in new population do
    Selection
    Crossover on selected chromosomes
    Mutation
  end while
end while
```

Individual genetic operators are explained in detail in following subsections.

2.5.1 Representation of an Individual

In our case the individual is actually a list of values used as parameters for the behaviour-based controller. The parameters to be evolved by the genetic algorithm are chosen by the experimenter and can be looked at as "genes". A set of N genes is in terms of GA a "chromosome". An individual is represented by a vector of B-bit wide binary numbers (genes), where each number is a parameter of the controller. We use numbers of the C++ data type `double` capable of storing real numbers. The size of B is fixed and depends only on the computer architecture of the used computer.

Figure 2.1: A chromosome representing an individual with five genes (N=5). B marks a B-bit wide binary block.

2.5.2 Initialization

Initially in the genetic algorithm many individuals are randomly generated to form a starting population. This should ensure the covering of most of the range of possible solutions in the search space. Occasionally, the solutions may be seeded in areas where optimal solutions are likely to be found.
In the Bebal-library the algorithm replicates the initial solution provided by the user until the designated population size is reached. Additionally each replicate is randomly altered as described in the mutation process in Section 2.5.7.

2.5.3 Fitness Function

The fitness function assigns each individual (or chromosome/solution) a positive value based on how well the individual performed in its environment. It should reward positive results and penalize unsuccessful individuals. This requires a function that is fast to compute as the GA iterates many time, especially in evolutionary robotics, leading to a sometimes not straightforward definition of the function.
It is possible to look at the genetic algorithm as an attempt to find maxima in a fitness landscape, where a fitness landscape is the evaluation of the fitness function for all candidate solutions.

2.5.4 Selection

Selection is a process of choosing parent chromosomes from the population for creating an offspring, on which then further genetic operations are applied. The offspring is transferred into the new population and so it belongs to the next generation.
Selection can be achieved in various ways, however **tournament selection** has been proven to be effective in terms of quality and computing requirements. Tournament selection consists of a series of duels. In each duel of n genomes randomly selected from the current population. With a probability p the individual with higher fitness is chosen to continue to the next round of the tournament. After evaluating all rounds the winner is chosen as the offspring.
This type of selection has certain advantages compared to others, e.g. roulette-wheel selection in which the probability of selecting an individual is proportional to its fitness and the sum of fitness of all individuals in the generation. The advantages of tournament selection are:

- Efficiency of code, ability to work on parallel architectures
- No need to evaluate the whole population, it only need to know the fitness of individuals selected into the the tournament
- It is not influenced by the absolute values of the fitness function. This is important, as typically the optimal fitness value is unknown.
- The selection pressure can be easily adjusted (by setting n and p).

It has also been proved that the tournament size of 2 is sufficient and it increases the overall speed of the genetic algorithm.

2.5.5 Elitism

To ensure that the so far best solution found is not lost in progress of generations, a certain amount of the best (fittest) individuals is always copied without change or the application of genetic operators into the new generation. However this can be disabled by setting the number of elite individuals to zero.

2.5.6 Crossover

Changes to the population are brought in by genetic recombination operators – crossover and mutation. Crossover (or recombination) in its simplest form takes two chromosomes that represent two individuals and are therefore called “parent” chromosomes, and produces as a result two “child” chromosomes, or offsprings. With a pre-defined probability the parent and offspring chromosomes are not the same, but the parents genes are exchanged at a crossover point. As previously defined, a chromosome consists of N genes, where a gene is a B-bit wide binary represented number. The crossover point is the point beyond which all genes are swapped between the two parent chromosomes. The resulting chromosomes are the children. The crossover point is selected randomly on both parents’ organisms but always between two B-bit wide number blocks so the binary representation of data stays intact (Fig. 2.2).

2.5.7 Mutation

The purpose of the mutation process is to bring in small random changes into the genetic code of the population in order to extend the search space of the genetic algorithm. In a standard binary representation of the genome each bit is altered with a probability P_m. This is however not suited for altering B-bit wide number types used as parameters for the robot controller. The reason is that a change in a single bit may change the represented value significantly. This drawback can be solved by either encoding the numbers in Gray code (where two successive values differ in only one bit) or simpler, by using **creep mutation**. Creep mutation is fundamentally equal to the

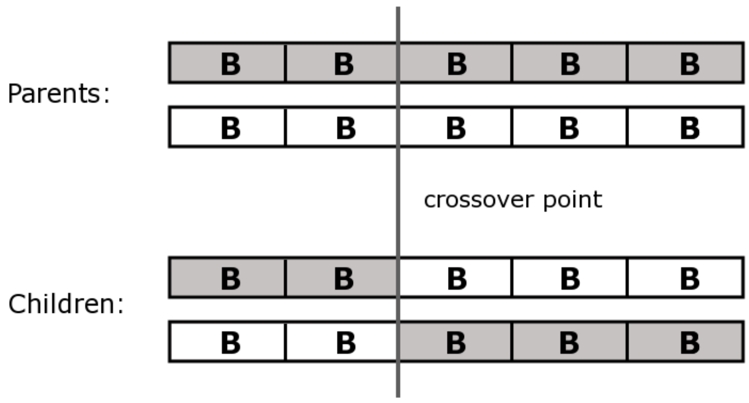

Figure 2.2: Example of a single point crossover between the second and third gene. B marks a B-bit wide binary block.

basic version, but instead of checking each bit, the B-bit wide blocks are converted to real numbers. Each number is then altered with a probability p by adding or substracting a (uniformly distributed) random number chosen from a interval. The interval is calculated from a user-defined range in which the appropriate gene value has to stay. After processing all numbers they are converted back to their binary representation and written to the genome.

Chapter 3

Solution and Implementation

This chapter focuses on the requirements put on the the library and the way they were solved while explaining the main architectural choices made.

3.1 Requirements

The main requirements for the Bebal-library were:

1. read user-defined behaviours and constraints for the BBR-controller and create one,
2. execute the simulation with one or multiple robots in the Player/Stage simulator,
3. log the results for later evaluation,
4. if enabled by option, perform an experiment based on ideas of evolutionary robotics, that is perform a series of test runs to optimize the controller constraints until the desired results are achieved (or any other condition is met, such as the maximum number of generations)

In addition the library was expected to fulfil so called "Software Quality Attributes" such as:

1. Usability – easy and fast to configure and use.
2. Modularity and extendability – the user should be allowed to extend the software with his own interfaces or modules.

3. Portability – it has to be possible to run it everywhere where Player/Stage can be run.

4. Performance – in general the hardware requirements of this software should be low and it should be executable on every computer that is capable of running the Player/Stage simulator without delaying the simulation process.

3.2 Solution

The following hierarchical structure of the solution provides an overview of the architecture of the library. Each part contains a description, the desired output and the functional requirements for it.
In short, the library acts as a layer between the low-level Player library (*libplayerc++*) and the user providing so additional functionality, see also Fig. 4.2.

1. **Construction of the BBR Controller**

   (a) Description:
   This part creates the BBR Controller that is used later on for managing the robots actions.

   (b) Output:
   It creates a controller on start-up from constraints and behaviours the user has to provide as input (XML and plain-text configuration files).

   (c) Functional Requirements:

   - Read user input: It reads and processes user input in order to create a BBR-controller. The input should be a XML-file with the behaviours and an optional plain-text file with variables that are used as constraints for the sensor input evaluation. If the user chooses to use genetic algorithms to tweak his solution, then these variables are optimized.
   - Behaviours: Individual behaviours have to be listed, represented and accessible from the controller. A behaviour consists of:

- Conditions for the sensor input (Rules) – thresholds (constraints) for one or more sensors and when it should be triggered. An evaluation function then determines if a behaviour should be activated or not based on the input.
- Calls (Actions) – In case a behaviour is marked as active, what it should do with actuators in sequence.

Behaviours in our perception can be divided into two types.

i. Behaviours with conditions.
ii. Behaviours without conditions. These behaviours will not be taken into account when evaluating the current active behaviour and are of two kinds
    A. *custom* behaviours meaning their actions are programmed in modules
    B. list of actions to be executed when called (similar to a procedure).

2. **BBR Controller**

(a) Description:
The controller is responsible for reading the robots sensor input, evaluating it and to act according to its behaviours.

(b) Output:
The BBR controller is a so called homogenous controller – it is replicated as many times as there are robots defined. Each copy manages the actions of a single robot, thus the output is a sequence of actions in each step of the simulation.

(c) Functional Requirements:

- Interface with robot sensors: Interfaces to all available sensors (or *ClientProxies* as they are called in the Player interface) are present along with capabilities to retrieve data from the environment.
- Main controller loop: The “heart” of the controller evaluates in each step conditions of the behaviours and executes corresponding actions on the robot’s actuators.

3. **Player/Stage Controller**

   (a) Description:
   Means of controlling the simulator have to be present with the possibility of retrieving and influencing the environment (position of robots, obstacles, . . .).

   (b) Output:
   Spawns a Player/Stage process for each simulation and connects to it.

   (c) Functional Requirements:

   - Run Player/Stage server: Player/Stage has to be spawned in a separate thread or process in order to work properly. Then it is also possible to control the process (to start/end it).
   - Player/Stage interface: This is solved with a little trick as the libplayerc++ library provides a *SimulationSensor* that interfaces with the Stage simulator allowing so to influence the simulator data.

4. **Genetic Algorithms and Evolutionary Experiments**

   (a) Description:
   This module ensures that the software is capable of conducting a series of simulation runs in order to optimize the controller (or more exactly, its rules and parameters) using genetic algorithms.

   (b) Output:
   When a configuration file describing the experiment, containing at least the rules for the robot controller, variables for the GA (mutation rate, . . .) and other needed properties (minimum fitness, maximum number of generations, etc.) is used as input, the experiment should start. The output is a set of optimized parameters for the controller and informations about the progress of the experiment (average/maximal fitness, time taken, etc.).

   (c) Functional Requirements:

   - BBR Controller: See previous section. The robot controller part is reused.
   - Representation of individuals: Each individual is represented through a chromosome holding the genes, or more specific their values, that are to be evolved.

- Genetic Algorithms: Methods of evolving individuals in time (selection, crossover, mutation to name the most important) are needed to successfully complete an evolutionary experiment. Also methods for extracting settings for the experiment are required.

5. **Logging**

   (a) Description:
   If enabled by the user a logger ensures that a simulation is retraceable even without running it again. It should provide enough data at the end of an experiment or simulation to use it for evaluation and comparison (e.g. progress of a population in time). Eventually it can be used to develop and debug robot controllers during their design.

   (b) Output:
   Various log-files, see 5.7.

   (c) Functional Requirements:

   - Logger configuration: It has to be possible to set what and where to is going to be logged. For the console output a logging library is used (*log4cxx*).
   - Output data has to be saved in a compatible wide-spread file format, therefore .csv and other “simple” text-only output formats were taken into consideration.

3.3 External Libraries

Parts of the implementation include 3rd-party libraries, namely the *boost-libraries* [9], *libxml2* [8] and *log4cxx* [10]. These were chosen for reasons of efficiency, quality and time saving, allowing to set the focus on the development of essential parts of the Bebal-library.
Of course the *libplayerc++* [3] library was used as a cornerstone as the whole purpose of the Bebal-library is to interact with Player/Stage.

Chapter 4

Users Manual

4.1 Introduction

Behaviour-based robot control is a method of controlling robots built upon several modular behaviours and a mechanism that decides which of the behaviours will be triggered according to the actual input from the sensors of the robot. Behaviour-based control is therefore reactive – the current action of the robot does not depend on its previous actions, but solely on its sensor inputs. Each behaviour is activated when certain conditions in its input are met and of course, it does matter in which order this behaviours are defined, as they are linearly evaluated and executed. This is also due to the so-called vertical organisation of behaviours the Bebal-library uses (in contrast to the horizontal organisation the Pyro simulator (see 2.1.2) can utilize).
The Bebal-library focuses on easy design of control programs for autonomous robots based on behaviours. This chapter is a guideline on how to use this library and to create control programs for your robot, either real or simulated in the Stage simulator. As the library can be used in two ways – as a classical behaviour based controller or to optimize an existent controller with genetic algorithms, the manual is divided in several sections. Sections 4.2 – 4.9 are relevant for both approaches and give a basic introduction on how to use this library. Sections 4.10, 4.11 and the Programmer's reference in Chapter 5 explain advanced control mechanisms and contain a guide for using genetic algorithms to optimize the controller. All important steps include simple examples that can also be found on the attached CD.

4.2 Prerequisites

The library itself uses a couple of external libraries and tools, which have to be installed on your system in order to successfully run and compile control programs. A Linux distribution or a comparable environment (such as Cygwin on Windows, Solaris or *BSD), is essential as the Player/Stage software is only available on this platform(s). Before installing the simulator software the following packages are needed:

- The OpenCV library[7] – dependency for Player, collection of algorithms for computer vision.
- The libXML2 library[8] – necessary for reading XML–configuration files.
- Boost C++ libraries[9] – required by the library for many operations.
- The log4cxx library[10] – for logging and debugging.

For help on how to install appropriate libraries please see the Programmer's reference.

4.3 Player/Stage

Player/Stage is the environment for all simulations and experiments conducted using this library. It consists of two independent, but cooperating programs:

Player is a network server for robot control, providing an interface to the robot sensors and actuators over the TCP/IP network. It also does not make a distinction between real and simulated robots (or types of robots for that matter) allowing so to rapidly develop and test control programs in a simulation and apply them on real robots, usually with none or minor adjustments.

Stage simulates a population of mobile robots along with a variety of sensors, actuators and other objects placed into a two-dimensional environment. It also provides a commonly used plugin (libstageplugin) which is used to map the simulated sensors to Players virtual devices. The control program, or in this case also the library, then utilizes this interface.

The detailed installation process of Player/Stage can be found in the Programmer's reference.

4.3.1 Using Player/Stage

The first step before even to try to develop a robot controller is to define the environment the robot will have to operate. Stage allows the (re)creation of low-fidelity, two-dimensional worlds by importing them from a bitmap image and/or defining models through just one configuration file, usually ending with a .world file suffix. This file also sets other vital properties of the simulation such as speed or the number of robots along with their sensor capabilities.
A .world file has a pretty straightforward syntax, as shown in the following avoid_example.`world` file:

```
# defines Pioneer-like robots
include "include/pioneer.inc"
# defines 'map' object used for floorplans
include "include/map.inc"
# defines sick laser scanner for the robot
include "include/sick.inc"
# size of the world in meters
size [16 16]
# set the resolution of the underlying raytrace model in meters
resolution 0.02
# update the screen every 10ms
gui_interval 10
# or uncomment the following line to disable the Stage GUI
#gui_disable 1
# the amount of real-world (wall-clock) time the simulator
# will attempt to spend on each simulation cycle.
interval_real   100
# the length of each simulation update cycle in milliseconds
interval_sim    100
# configure the GUI window
window
(
  size [ 591.000 638.000 ]
  center [-0.010 -0.040]
  scale 0.028
)
# load an environment bitmap
map
(
  bitmap "bitmaps/cave.png"
  size [16 16]
  name "cave"
)
# create a robot
pioneer2dx
(
  name "robot1"
  color "red"
  pose [-6.5 -6.5 -90]
)
```

Lines beginning with a hash key (#) are comments, otherwise the line sets a property. The most important sections are the "map" and the "pioneer2dx" properties. "map" contains information about the image-file the world has to be created from, the size of the world (in meters) and assings this world a name (id). "pioneer2dx" describes the robot model and its available sensors, in this case a red robot named "robot1" with only a ring of sonar sensors and a motor added by default. An actual picture of the Pioneer 2DX robot can be seen in Fig. 4.1.
For speeding up experiments or test runs of the control program it is possible to set the simulation speed by adjusting the "interval_real" and "interval_sim" properties. Setting the real interval smaller than the simulation interval boosts the simulation speed, although the "interval_real" property should never be zero, so the Player software has time to process all messages, and should be adjusted very carefuly.
Additionaly the GUI may be disabled by setting the "gui_disable" property to 1. This works in the current version of Player/Stage (2.x) only after applying an attached patch (see section Troubleshooting in the Programmer's reference).

In order to connect the simulated world with the Player server providing drivers (proxies) that the library accesses, another configuration file (usually ending with .cfg) for Player is required. To load and use the world

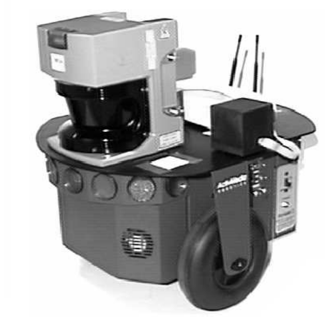
Figure 4.1: The original Pioneer 2DX robot.

created in Stage into Player, this file (avoid_example.cfg) should look like this:

```
# load the Stage plugin simulation driver
driver
(
  name "stage"
  provides ["simulation:0" ]
  plugin "libstageplugin"
  # load the Stage world file into the simulator
  worldfile "avoid_example.world"
)
# create the environment
driver
(
  name "stage"
  provides ["map:0"]
  model "cave"
)
# Create a Stage driver and attach position2d and
# laser interfaces to the model "robot1"
driver
(
  name "stage"
  provides ["6665:position2d:0" "6665:sonar:0" ]
  model "robot1"
)
```

It uses the same syntax as the Stage configuration file, but each section has to be named "driver", as everything in Player is represented as a driver/proxy. The proxies this configuration file would provide are "simulation", "map" that relate to the properties of the world and the simulation and "position2d", sonar" that apply to our previously defined "robot1", allowing the library and the controller to gain control over the robots motor and sonar sensor on port 6665.
Multiple robots can be defined similarly as "robot1" by only adjusting the ports Player will listen on (e.g. 6666, 6667, ...).

As the full possibilities and configuration options of Player/Stage would go beyond the scope of this manual, they can be found in Player/Stage documentation [3].

4.4 Overview

The main goal of the Bebal-library is to simplify development of behaviour based robot controls and minimize the amount of low-level programming code to be written so the experimenter can focus on the substantial part of the controller. The library takes care of:

- processing defined behaviours,
- merging them into a controller that interacts with Player and
- providing *visual* feedback in an environment simulated by Stage

as shown schematically in Figure 4.2.

All behaviours used are to be stored in an XML-file with a specified structure, from which the library then extracts all informations it needs to create a controller. The resulting function of the designed controller can be described as a loop with the following body:

1. Take the first behaviour defined.
2. Evaluate sensor conditions associated to this behaviour.

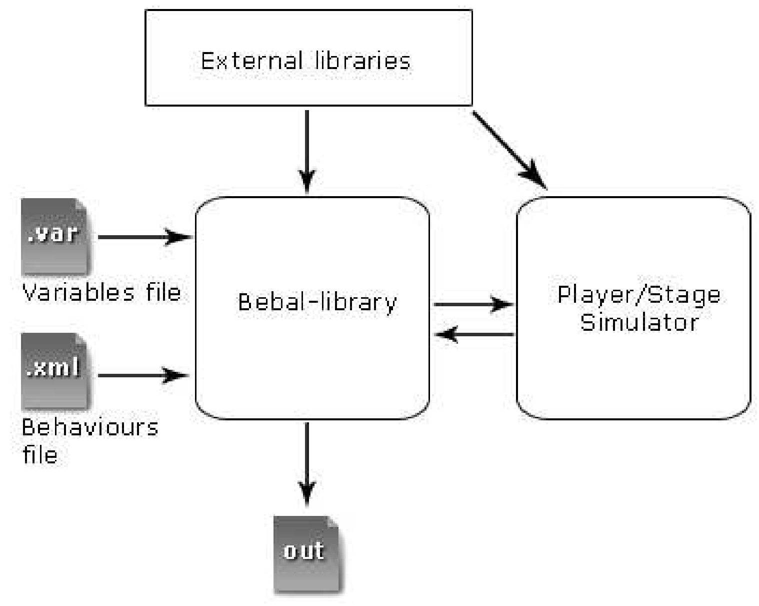

Figure 4.2: Schematics of the library

3. If all conditions comply with their constraints, execute associated actions (basically by calling other behaviours such as "move") and continue with the next iteration from step 1.
4. If not, take the next behaviour in order (if possible) and go to step 2.

4.5 Built-in Sensors

The Bebal-library by default provides interfaces to the most common subset of sensors/actuators offered by Player/Stage. A list of them follows along with a description what they do, if attached to a robot:

- Sensors:
  - laser – measures distance to objects that reflect a laser beam,
  - sonar – measures distance to objects that reflect IR or ultrasonic waves,
  - blobfinder – provides simple visual input, tracks colored objects,

- Actuators:
  - gripper – handles objects (e.g. for foraging pucks),
  - motor – moves the robot (in 2D-space).

Additionally there is a "simulation" sensor that serves as an interface for dynamical setting and reading properties of the simulated world.
All sensors can be evaluated (with the exception of the motor, but it is free to implement a custom evaluation function) with appropriate commands and bound with conditions to a behaviour.

More sensors and actuators can be easily implemented should they be needed, see the Programmer's reference chapter for more information on that subject.

4.6 Basic Behaviours

There are two basic behaviours already in the library that can be used right from the start and need not to be defined. The adjective "basic" roots in the fact that these behaviours are more or less just an interface to their actuator counterpart and that from this behaviours more complex behaviours can emerge.

The **move** behaviour causes the robot to move in two–dimensional space. It takes two arguments: speed (ranging from 0.0 to 1.0) and angular speed (in degrees per second).

The **gripper** behavior commands the gripper device on the robot (if present), usually used to grab and carry objecs. It requires one argument, whether to open or close the gripper. It can also be used as an input sensor to determine whether the gripper paddles are open or closed.

4.7 Defining Behaviours

The very substantial part of creating a behaviour based controller is defining behaviours.
All behaviours are to be stored in an XML-file following the listed DTD template:

```
<!ELEMENT bbl (Robots,Behaviours)>

<!ELEMENT Robots (Robot+)>

<!ELEMENT Robot (#PCDATA)>
  <!ATTLIST Robot name CDATA #REQUIRED>
  <!ATTLIST Robot host CDATA #REQUIRED>
  <!ATTLIST Robot port CDATA #REQUIRED>
  <!ATTLIST Robot log_movement CDATA #IMPLIED>
  <!ATTLIST Robot timeout CDATA #IMPLIED>
  <!ATTLIST Robot max_cycles CDATA #IMPLIED>

<!ELEMENT Behaviours (Behaviour*)>

<!ELEMENT Behaviour (Argument*,Conditions?,Calls?)>
  <!ATTLIST Behaviour name CDATA #REQUIRED>
  <!ATTLIST Behaviour type CDATA #IMPLIED>

<!ELEMENT Argument (#PCDATA)>
  <!ATTLIST Argument value CDATA #REQUIRED>

<!ELEMENT Conditions (Condition*)>
<!ELEMENT Condition (Constraint*)>
  <!ATTLIST Condition type CDATA #REQUIRED>

<!ELEMENT Constraint (#PCDATA)>
  <!ATTLIST Constraint value CDATA #REQUIRED>

<!ELEMENT Calls (Call*)>
<!ELEMENT Call (Argument*)>
  <!ATTLIST Call behaviour CDATA #REQUIRED>
```

In other words, there has to be exactly one *Robots* section and one *Behaviours* section, encapsulated in the *bbl* root-node.
The *Behaviours* section should contain all defined behaviours. Each behaviour has a name and contains a *Conditions* and a *Calls* section. In the first section, individual conditions for sensor inputs are listed. If all condtitions are met, calls from the Calls-section are executed, when not, the controller of the library evaluates the next behaviour in order.

Each condition is bound to an input sensor (by the property *type*, see Section 4.5 for available sensors) and will most likely require parameters (*Constraints*), e.g. if the laser sensor returns a distance value smaller than 5. Arguments (and their order) for each built-in behaviour and sensor are described in the attached Doxygen-documentation under the Namespace reference - section Variables. The number of conditions and calls (actions) is theoretically unlimited.
It is possible to call not just basic behaviours such as "move", but also other user-defined behaviours. The name of the called behaviour has to be passed as an argument. In this case the conditions of the execution of the according behaviour are not evaluated but it causes only the execution of all actions of the called behaviour. This is similar to calling a (stored) procedure in functional programming, thus supporting the reuse of call-orders and emerging more complex behaviours.

The *Robots* section declares the connection settings for a *Robot* by setting to which Player-server the controller has to connect (*host* and *port*) and some other useful properties:

- *name* must hold the name of the robot as defined in the Player/Stage configuration files,
- *log_movement* is a boolean value, which when set to true enables logging of the path the robot travelled to a text-file,
- *timeout* sets the timeout in seconds the robot is allowed to interact within the simulation,
- *max_cycles* is the amount of simulational cycles (steps) the robot is allowed to execute within the simulation.
  In case both *timeout* and *max_cycles* is set the controller stops on the condition that occurs first.

The XML-file serves then as input for the library and should be tested before actually used with the xmllint utility (usually part of the libxml2-utils package) with the following command:

```
xmllint --valid --noout --dtdvalid \
[path-to-installed-library]/behaviours.dtd \
[behaviours-file-to-be-checked].xml
```

For details on DTD templates and XML syntax see [4].

4.8 Using Variables

It is preferable to create a text-file with a `.var` extension to hold variables for sensor constraints, that might be used when creating rules to behaviours, instead of plain numbers for sake of code clarity, centralization, faster access or even optimalization by genetic algorithms. The file structure is kept simple:
Each line has to consist of an unique variable name and a value, separated by an arbitrary number of whitespaces (space, tabulator), e.g. a file avoid_example.`var` contains the following:

```
VAR_AVOID_SONAR_MIN 1.0
VAR_AVOID_LEFT_MOVE_SPEED 0.2
VAR_AVOID_LEFT_MOVE_ANGLE 15.0
VAR_AVOID_RIGHT_MOVE_SPEED 0.2
VAR_AVOID_RIGHT_MOVE_ANGLE -15.0
```

4.9 Basic Example

Assuming the use of the variables-file from the previous section, a very simple example behaviour file (avoid_example.`xml`) could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<bbl>
<Robots>
  <Robot name="robot1" host="localhost" port="6665" />
</Robots>
<Behaviours>
  <Behaviour name="avoid_left">
    <Conditions>
      <Condition type="sonar">
        <Constraint value="SONAR_DIRECTION_NW" />
        <Constraint value="SONAR_MIN_RANGE_SMALLER_THAN" />
        <Constraint value="VAR_AVOID_SONAR_MIN" />
      </Condition>
    </Conditions>
    <Calls>
      <Call behaviour="move">
        <Argument value="VAR_AVOID_LEFT_MOVE_SPEED" />
        <Argument value="VAR_AVOID_LEFT_MOVE_ANGLE" />
      </Call>
    </Calls>
  </Behaviour>
  <Behaviour name="avoid_right">
    <Conditions>
      <Condition type="sonar">
        <Constraint value="SONAR_DIRECTION_NE" />
        <Constraint value="SONAR_MIN_RANGE_SMALLER_THAN" />
        <Constraint value="VAR_AVOID_SONAR_MIN" />
      </Condition>
    </Conditions>
    <Calls>
      <Call behaviour="move">
        <Argument value="VAR_AVOID_RIGHT_MOVE_SPEED" />
        <Argument value="VAR_AVOID_RIGHT_MOVE_ANGLE" />
      </Call>
    </Calls>
  </Behaviour>
  <Behaviour name="wander">
    <Conditions>
      <Condition type="sonar">
        <Constraint value="SONAR_DIRECTION_FRONT" />
        <Constraint value="SONAR_MIN_RANGE_GREATER_THAN" />
        <Constraint value="VAR_AVOID_SONAR_MIN" />
      </Condition>
    </Conditions>
    <Calls>
      <Call behaviour="move">
        <Argument value="1.0" />
        <Argument value="0.0" />
      </Call>
    </Calls>
  </Behaviour>
```

```
</Behaviours>
</bbl>
```

This XML-file defines a simple controller that is not very robust and the robot might get stuck, but is sufficient for demonstrational purposes. Using this behaviour definitions the robot evaluates the sonar (infrared) sensors in front of it. If the sensors readings cross a threshold defined by the `VAR_AVOID_SONAR_MIN` variable (with the value of 1.0 specified in the var-file mentioned in section 4.8), meaning effectively the robot approaches a wall or an another obstacle that reflects IR to a distance of less than 1.0, the avoid_left or avoid_right behaviour becomes active. It causes then the robot to avoid it, either by turning left or right, depending on either which behaviour was defined first or is the only behaviour to be evaluated as active (this could happen for example if the obstacle is registered only on the left side).
Note that the "move" behaviour is called directly with numerical parameters, but it is preferable to set them in the variables-file in the same way as for the "avoid" behaviours.

To start the simulation run the compiled bebal-controller program by typing the following into a console or shell (in the directory where the examples are located in):

```
# ./avoid_example -c avoid_example.cfg \
 -b avoid_example.xml -v avoid_example.var
```

or in general

```
# ./my_bebal_program -c player-configuration-file.cfg \
-b behaviours-file.xml -v variables-file.var [-d 1]
```

where:
`my_bebal_program` is the compiled executable of the controller program,
`-c` is the path to the player configuration file,
`-b` is the path to the behaviours-file,
`-v` is the path to the variables-file,
`-d 1` enables optional debugging output when set.

A properly launched program should show up a window like in Fig. 4.3 and the robot should start moving and avoiding walls. The (not so clear) red circle is the moving robot and the line depicted in the image is the trail of the way the robot has traveled. The trail has to be enabled manually in the Stage simulator by selecting View – Show trails from the menu or by pressing the “T” hotkey.

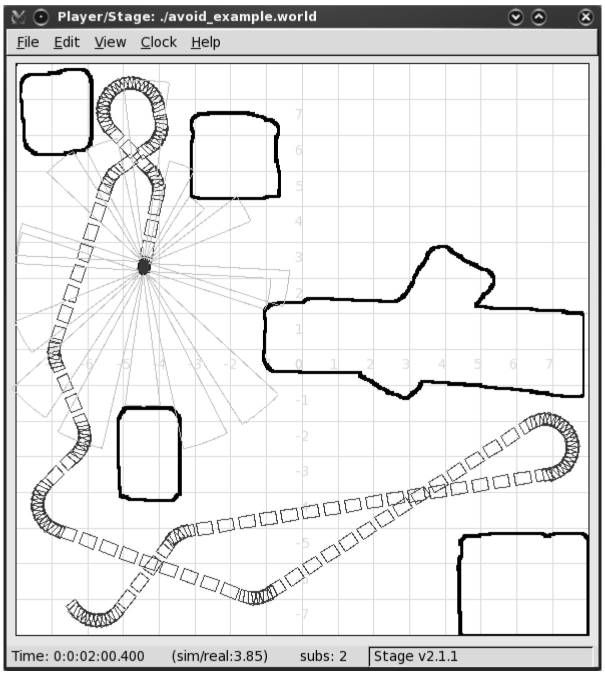
Player/Stage: ./avoid_example.world
File Edit View Clock Help
Time: 0:0:02:00.400 (sim/real:3.85) subs: 2
Stage v2.1.1

Figure 4.3: The Stage simulator window with the running example.

4.10 Advanced Behaviours and Sensor Evaluation

With the exception of some basic control programs or simple sensor input the need for more complex sensor evaluation and robot control raises. As it is not realistic to cover everything within textual configuration files, classical controller programming can not be omitted. However, it is limited to implementing interfaces the library provides, both for sensors and behaviours. Refer to the Programmer’s reference on this topic.

4.11 Using Genetic Algorithms

For optimizing the behaviur of a robot, it is possible to fine-tune the variables used by the controller by conducting experiments based on genetic algorithms. To do so, a text-file with a `.exp` suffix has to be created, where the values can be set by the user. The file contains a list of key-value pairs describing the experimental setup:

```
CROSSOVER_RATE        0.8
MUTATION_RATE         0.5
NUM_ELITE             2
POPULATION_SIZE       12
PLAYER_CONFIG         gripper.cfg
BEHAVIOURS-FILE       gripper.xml
VAR-FILE              gripper.var
GENES                 VAR_APPROACH_BLOCK_SPEED_COEF
GENE_RANGES           10.0-10000.0
TOURNAMENT_SIZE       2
GENERATIONS           50
LOG_FOLDER            log/gripper
SIM_INTERVAL_REAL     20
SIM_INTERVAL_SIM      100
PLAYER-EXECUTABLE     player
```

For explanation of what each term or property means review the Genetic Algorithms section [2.5]. A brief description of each parameter follows:

- *CROSSOVER_RATE* sets the probability of a crossover between two parental chromosomes.

- *MUTATION_RATE* sets the probability of a creep mutation of each gene in an offsprings chromosome.

- *NUM_ELITE* defines how many of the best individuals (the "elite" sorted by fitness) will be copied into a new generation.

- *POPULATION_SIZE* is the number of individuals in one generation to be tested.

- *PLAYER_CONFIG* relative path to the Player configuration file. A Stage configuration file must be also defined as referenced in the Player config.

- *RULES-FILE* is the relative path to the .xml file containing defined robots and behaviours.

- *VAR-FILE* is the relative path to the .var file containing the variables used in the behaviours–file.

- *GENES* lists all selected variables that should be used as "genes" and therefore be optimized by the GA (at least one or all from the file). The variables are to be separated by a colon, e.g.:
  VAR1:VAR2:VAR3:...
  Also they have to be defined in the *VAR-FILE*.

- *GENE_RANGES* sets the limits for the creep mutation for each gene/variable using the following syntax:
  VAR1_MIN-VAR1_MAX:VAR2_MIN-VAR2_MAX:...
  Each section separated by a colon represents the minimal (`VAR1_MIN`) and the maximal value (`VAR1_MAX`) the corresponding variable (e.g. `VAR1`) can reach in the mutation process.

- *GENERATIONS* is the maximum number of generations the GA can reach.

- *SIM_INTERVAL_REAL* is the value of the `interval_real` variable from the Player configuration file. It has to be the exact same value!

- *SIM_INTERVAL_SIM* is the value of the `interval_sim` variable from the Player configuration file. It has to be the exact same value!

- *LOG_FOLDER* is the path to a folder, where the results of the experiment are to be saved. The parent directory has to be present on the system (e.g. "log" in this case).

- *PLAYER-EXECUTABLE* is the name of the Player executable on the installed system (usually "player", but can be e.g. "robot-player" on Ubuntu when installed from the repository).

This experiment-file is used in a prepared experiment (as described in Section 5.6.2) and can be found on the CD. It can be started (after compiling) with

```
# ./gripper -e gripper.exp
```

or in general by

```
# ./my_bebal_experiment -e name-of-the-experiment-\
configuration-file.exp
```

where `my_bebal_experiment` is the compiled executable of the Bebal-controller with settings for this specific experiment.

Chapter 5

Programmer's Reference

5.1 Introduction

This reference contains all information needed to install and use the Bebal-library with its full potential. It will not go into deep details of every class and function, please refer to the included HTML documentation generated by Doxygen ([5]) for that.

5.2 Overview

The Bebal-library is composed of 4 essential classes:

- Behaviour: defines a standard Behaviour containing sensor bound conditions and a list of actions.
- Sensor: serves as an interface to the Playerc++ library ClientProxies, providing sensor evaluation or execution (for actuators).
- bLib: the actual brain and interface to the library, merges behaviours into a controller, reads configuration files, holds variables and so on.
- Genetics: used for experiments, implements genetic algorithms and uses them to optimize variables of the controller.

5.3 Installation

5.3.1 Prerequisites

The following software is needed in order to use and compile the library and your own control programs or modules.

- A Linux distribution (or comparable environment, such as Cygwin on Windows, *BSD or Solaris) [6]
- Player Robot Device Interface [1]
- Stage Robot Simulator [2]
- The OpenCV library [7]
- The libXML2 library [8]
- Boost C++ libraries [9]
- The log4cxx library [10]

Various additional libraries or software may be required to satisfy package dependencies, depending on what is already installed on the target system. Of course a C++ compiler, like GCC, is required to build the library, but it is installed on the most linux distributions by default.

5.3.2 Installing the Software

As by far not all distributions provide packages for the Player/Stage software and for the universality of this reference, a description of a manual installation follows. This assumes that the user is familiar with basic linux shell commands and software installation from source packages on *NIX–like environments.

The packages must be installed exactly in this order to compile and work correctly, as they are dependent on each other:

- OpenCV 1.0
- Player 2.x
- Stage 2.x

A rudimentary guide to installing software from source packages from the linux (root) shell follows:

```
# cd path-to-extracted-package
# ./configure
# make install
```

Troubleshooting

Installing this software from source packages can be a little tricky, as some parts of it are not maintained anymore.

At first, it is likely the OpenCV compilation will fail, as it is incompatible with newer versions of ffmpeg, which it is dependant upon. This can be resolved by patching the OpenCV library before compilation with the attached patch file:

```
# cd path-to-extracted-opencv-package/otherlibs/highgui/
# cp path-to-patch/opencv-cvcap_ffmpeg-img_convert.patch
# patch cvcap_ffmpeg.cpp opencv-cvcap_ffmpeg-img_convert.patch
# rm opencv-cvcap_ffmpeg-img_convert.patch
```

It is also vital to configure the software with the following parameters, otherwise the linking will fail.

```
# ./configure --with-ffmpeg --enable-shared --without-quicktime
```

For the Player/Stage installation it may also be required to set various paths and environment variables, depending on the local settings. Assuming the default installation location (/usr/local/) this can be solved by exporting them:

```
# export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:\
/usr/local/lib/pkgconfig/
# export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib/
```

For allowing the user to disable the Stage simulator GUI by setting `gui_disable` in the configuration file, another patch is required (applying to Stage version 2.0.4/2.1.1 and the use of the blobfinder):

```
# cd path-to-extracted-stage-package/src/
# cp path-to-patch/stage_disable_gui.patch
# patch model_blobfinder.c stage_disable_gui.patch
# rm stage_disable_gui.patch
```

This patch is to be applied at own risk, as it is not modifies the software. Functionality is even then not guaranteed as Stage officialy does not support the option of disabling the GUI.
Unexpected behaviour (memory exceptions, failing to read the entire config file, etc.) was encountered when compiling Player and Stage along with already installed Boost libraries, so it may be recommended to install them only afterwards.

5.3.3 Compiling the Library

To install the Bebal-library into the users system, it has to be compiled. There is a `makefile` attached as usual for Linux-software, therefore it should be sufficient to execute the following commands to install the library:

```
# cd path-to-bebal-library-source-files
# ./configure
# make && make install
```

For sake of completeness a list of needed settings (flags) for the g++ compiler follows:

- Includes:
  `-I/usr/local/include/player-2.1/ -I/usr/include/libxml2 -I/usr/include/boost-1_37`
- Library search path:
  `-L/usr/local/lib -L/usr/lib/boost-1_37`
- Link with libraries:
  `-lplayerc++ -lxml2 -llog4cxx -lboost_filesystem`

The library installs by default into /usr/local/lib. A successful installation of the library creates static and dynamic libraries named libbebal-1.0* in that folder. C++ header files are located in /usr/local/include/bebal-1.0.

5.4 Creating custom Sensors

The Bebal-library provides interfaces only to the most commonly and widely used fraction of all drivers (proxies) available in Player (or libplayerc++ respectively). Some simulations may require additional sensor input or, more probably, custom evaluation of the sensor data. For example the blob–sensor is a good candidate, as almost every simulation created is likely to differ in the way the blob–data has to be handled, depending on what the controller will require, e.g. only a single line of visual input or the center of a colored object.
Programming a custom sensor interface consists in general of eight steps:

1. Creating a derived class from the Sensor-class.

2. Creating a default constructor *MyCustomSensor()*. Most of the time it should be sufficient to call the Sensor()–constructor here.

3. Creating a copy–constructor *MyCustomSensor(const MyCustomSensor& cs)*. Call the default Sensor() copy–constructor here with the reference to MyCustomSensor as a parameter. If the custom sensor has some extra data structures, copy them also here.

4. Implementing the function *void createSensor(PlayerCc::PlayerClient* Robot)*. This function should construct and initialize the appropriate ClientProxy from the libplayerc++ library.

5. Implementing the function *bool eval(arg_list_t* arguments)*. Can do nothing if not applicable (mostly if the proxy is an actuator), or put custom data evaluation function here. It should return *true* or *false* based on whether the evalution succeeded or not. The parameter *arguments* is a vector of boost::any values that can hold anything, designed to pass commands or constraints to the evaluation function. Unbox them to C++–data types by using the BOOST::unboxValue() function defined in the library.

6. Implementing the function *void execute(arg_list_t* arguments)*. Can do nothing if not applicable (mostly if the proxy is a passive sensor), or put custom commands for the ClientProxy here. The parameter *arguments* is to be used similar as in the *eval(arg_list_t* arguments)* function.

7. Implementing the function *PlayerCc::ClientProxy* getDevice()*. Should return a pointer to the created device/proxy.

8. Implementing the function *Sensor* clone()*. Should return a copy of the current MyCustomSensor–class by creating a new one using the copy–constructor. Usually it is sufficient to write something like:

   ```
   return new MyCustomSensor(*this);
   ```

   This is necessary for cloning robot controllers when multiple agents are to be simulated at once.

9. Optionally implement *void getVariables()* that should register custom variables in the library. Custom variables can be used as commands for execution or evalution in the behaviours-file.

After including the custom sensor header files to your controller program, it is necessary to register the custom sensor under a name it will be recognized when defining behavioural rules. To register the sensor use the function

```
registerSensor("custom_sensor_name", new bbl::MyCustomSensor());
```

A practical example can be found in `bebal/proxies/BlobSensor.cpp`. This sensor is customized to detect blobs of the color it gets passed as an argument and optionally test the distance to the blob. Included is an excerpt of the file with the most important functions and parts along with their description:

```
// Sets to which robot the BlobSensor should be attached to.
void BlobSensor::createSensor(PlayerCc::PlayerClient* Robot)
{
  this->pRobot = Robot;
  blobfinder = new PlayerCc::BlobfinderProxy(this->pRobot);
}

/**
 * Sets the variables to use as comparators for the distance to
 * the blob (smaller or greater than a distance).
 */
void BlobSensor::getVariables()
{
```

```
  if(blib == NULL) return;
  blib->registerVariable("BLOB_MIN_RANGE_GREATER_THAN",
  BLOB_MIN_RANGE_GREATER_THAN);
  blib->registerVariable("BLOB_MIN_RANGE_SMALLER_THAN",
  BLOB_MIN_RANGE_SMALLER_THAN);
  // ... truncated
}

/**
 * Evaluates the blobfinder with a given command.
 * Takes 0/1/3 arguments packed in a list.
 * @param[in] color Color: a long integer that defines the
 color (0x00RRGGBB)
 * @param[in] compare Comparator: see also getVariables();
 * @param[in] range A double value with the user-defined range to
 compare against
 * @return TRUE if the condition is met, FALSE otherwise
 */
bool BlobSensor::eval(arg_list_t* arguments)
{
  if((*arguments).size() > 0)
  {
    int color;
    BOOST::unboxValue((*arguments)[0], color);
    if(color<0) color = -color;
    bool seeing_color = false;
    int blob_index = -1;
    float range_min = 65000;

    // find our color
    for(uint i=0; i< blobfinder->GetCount(); i++)
    {
      blob = (*blobfinder)[i];
      if(blob.color == (unsigned int)color
        && blob.range < range_min)
      {
        range_min = blob.range;
        blob_index = i;
        seeing_color = true;
      }
    }

    // check for distance?
    if((*arguments).size() > 1)
    {
      int cmp;
      int range;
      BOOST::unboxValue((*arguments)[1], cmp);
      BOOST::unboxValue((*arguments)[2], range);

      // we have to see the color
      if(cmp == BLOB_MIN_RANGE_GREATER_THAN
      || cmp == BLOB_MIN_RANGE_SMALLER_THAN)
      {
        // ... truncated for demonstrational purposes
      }
    }
    else if(seeing_color) return true;

  }
  // ... truncated
  return false;
}

//! Get the BlobfinderProxy.
PlayerCc::ClientProxy*  BlobSensor::getDevice()
{
  return blobfinder;
}

//! Clones this Sensor
Sensor* BlobSensor::clone()
{
  return new BlobSensor(*this);
}
```

This custom sensor will be also reused in an example later in Section 5.6.2. The most important function in this sensor is the `eval()` function which determines whether a condition of a behaviour is evaluated as true or not. It takes several parameters that can be understood as commands accessible when defining behaviour conditions in the XML-behaviours-file. In general the user is free to determine what commands or parameters his custom sensor will accept, but it is a good practice to register the commands as variables in the library so the conditions or actions in the behaviours-file are more understandable. Sensors that are implemented in the Bebal-library have their commands documented in the programmer's reference generated by Doxygen.

5.5 Creating advanced Behaviours

Complex robot tasks may require precise coordination or be dependent on more sensor inputs than the standard functions in the Bebal-library provide. Because of this custom behaviours can be defined. They can be also seen as procedures that temporarily take over the control of a robot for executing specialized tasks, e.g. approaching a coloured block or to do some random avoiding, but in principle their limit is the designers imagination only. However custom behaviours should not be misused as a replacement for conventional robot controllers but used only when the task is too difficult or impossible to express as a set of rules and actions in the .XML behaviours-file. Therefore they also should take over control only for a short period so the Bebal-controller can function as intended.

Programming a custom behaviour takes in general of five steps:

1. Create a derived class from the Behaviour-class
2. Creating a default constructor *MyCustomBehaviour()*. Most of the time it should be sufficient to call the Behaviour()-constructor here.
3. Creating a copy-constructor *MyCustomBehaviour(const MyCustomBehaviour& cb)*. Call the default Behaviour() copy-constructor here with the reference to MyCustomBehaviour as a parameter. If the custom behaviour has some extra data structures, copy them also here.
4. Implementing the function *void execute(arg_list_t* arguments)*. Put custom commands or procedures here. The parameter *arguments* is

a vector of boost::any values that can hold anything, designed to pass on commands or parameters. Unbox them to C++-data types by using the BOOST::unboxValue() function defined in the library.

5. Implementing the function *Behaviour* clone()*. Should return a copy of the current MyCustomBehaviour-class by creating a new one using the copy-constructor. Usually it is sufficient to write something like:

   ```
   return new MyCustomBehaviour(*this);
   ```

After including the custom behaviour header files to your controller, it is possible to register the custom behaviour under a name it will be recognized later on when defining behaviours (with the *type="custom"* property of a <Behaviour> section, see 4.7). To register the behaviour in the library use the function

```
registerCustomBehaviour("custom_behaviour_name",
  new MyCustomBehaviour());
```

An example of such a behaviour is the *ApproachBlockBehaviour*. Basically it locates the nearest green block through the blob-sensor and approaches it. This will be necessary in order to complete a foraging task in the experiment described in Section 5.6.2. An extract from the `customBehaviours/ApproachBlock.cpp` file:

```
// Takes no arguments, approaches the block.
void ApproachBlockBehaviour::execute(arg_list_t* arguments)
{
  blobfinder =
  (PlayerCc::BlobfinderProxy*) blib->Sensors["blob"]->getDevice();
  int nearest = -1;
  float range_min = 65000;
  // find the nearest green block
  for(uint i=0; i< blobfinder->GetCount(); i++)
  {
    blob = (*blobfinder)[i];
    if(blob.color == 65280 && blob.range < range_min)
    {  // color == green
      range_min = blob.range;
      nearest = i;
```

```
    }
  }

  // found a block!
  if(nearest >= 0)
  {
    boost::any speed, angle;
    double angle_d, speed_d, coef;
    double center = blobfinder->GetWidth()/2;
    blob = (*blobfinder)[nearest];
    // calculate a new angle, so the block is in the center
    // of the camera
    BOOST::unboxValue(
    blib->gVariables["VAR_APPROACH_BLOCK_SPEED_COEF"], coef);
    speed_d = blob.range/coef;
    angle_d = (center - blob.x);
    if(angle_d > 90.0 || angle_d < -90.0) angle_d = 0;
    if(speed_d > 1.0) speed_d = 1.0;
    angle = angle_d;
    speed = speed_d;
    arg_list_t* args = new arg_list_t();
    args->clear();
    args->push_back(speed);
    args->push_back(angle);

    blib->Sensors["motor"]->execute(args);
  }
}

// Clones this behaviour.
Behaviour* ApproachBlockBehaviour::clone()
{
  return new ApproachBlockBehaviour(*this);
}
```

5.6 Building Control Programs and Experiments

The Bebal-library can be used in two ways, either as a standard behaviour-based controller or in addition, conduct series of experiments with the aim of optimizing the controller with genetic algorithms.
On the basis of simple examples principles of writing own controllers are presented. In the first section a controller for a robot is described who's task is to avoid obstacles (see 4.9). The second section shows the program used in a simple evolutionary experiment in which the robots had to forage a green block with its gripper in a small arena (4.11).

5.6.1 Creating a Standard Behaviour based Controller

The following code is the controller for the example in Section 4.9 and shows the necessary steps in order to create a simple controller:

```
// File on the CD: examples/avoid_example.cpp
// include needed header files, we need them always
#include <libplayerc++/playerc++.h>
#include <bebal/args.h>
#include <bebal/bebal.h>

using namespace PlayerCc;
using namespace bbl;

int main(int argc, char **argv)
{
bLib* lib;
// parse arguments given to the program
parse_args(argc,argv);

try
{
// create a new controller and we may want to debug it
lib = new bLib(gDebug);
    // start Player and Stage
    StartPlayerServer(gPlayerConfigFile);
    // add a robot that connects to the Player server
    lib->addRobot(new PlayerClient(gHostname, gPort));
    // attach the robots sensors and actuators
    lib->attachSensor("motor", new bbl::MotorSensor());
    lib->attachSensor("sonar", new bbl::SonarSensor());
    // load the behaviours and variables that
    // define the controller
    lib->loadVariables(gBebalVariablesFile);
    lib->loadConfig(gBebalBehavioursFile);
    // and finally start
    lib->start();
    // stop the Player server
    StopPlayerServer();
  }
  catch (PlayerCc::PlayerError e)
  {
delete lib;
    return -1;
  }

  return 0;
}
```

The directive `#include <bebal/args.h>` provides a function called `parse_args` that eases the parsing of parameters passed to the controller. After calling this method several variables are available:

- `gHostname` holds the adress of the Player server the controller should connect to (localhost by default). Set with the *-h* switch.
- `gPort` defines the port for the connection (6665 by default) (*-p*).
- `gDebug` is a boolean value that if passed to the controller enables debugging output (*-d*).
- `gBebalBehavioursFile` contains the path to the `.xml` behaviours-file (*-b*).
- `gBebalVariablesFile` contains the path to the `.var` variables-file (-*v*).

- `gPlayerConfigFile` contains the path to the `.cfg` Player configuration file (-*c*).
- `gExperiment` contains the path to the `.exp` experiments-file (-*e*).

In order to be able to build the program, flags for the linker have to be set. It is necessary to link the controller with the bebal-library (libbebal-1.0), usually by adding the `-lbebal-1.0` flag to the arguments of the compiler. Setting appropriate include paths to the header files of the Bebal-library may be necessary, depending on the configuration of the system. Usually this would be /usr/local/include/bebal-1.0 together with the include flags as from Section 5.3.3.

5.6.2 Creating an Experiment

In contrast to the simple controller presented in the previous section, the experimental part of the library needs in addition a fitness function to evaluate individuals. Also the class *Genetics* is used as an encapsulating control structure. The controller part (the *bLib* class) is still present, however as a part of the latter class. A controller for the gripper experiment as described in Section 4.11 looks following (file `examples/gripper.cpp`):

```
// We need a behaviour responsible for approaching the block
#include "customBehaviours/ApproachBlock.h"

// We need to have a fitness function. This function ranks
// individuals with the inverse amount of time it took to
// reach the green block. In gRuntime is always stored the
// runtime of the bebal-controller.
unsigned int fitness_function(Genetics* gen)
{
  int ms_runtime = gen->blib->gRuntime.tv_sec*1000
  + std::floor(gen->blib->gRuntime.tv_nsec/1000000);
  return (unsigned int) std::floor(1000000/ms_runtime);
}

// A custom abort condition for the simulation. The simulation
// ends when the gripper closes and holds the block.
bool gripper_check_abort(bLib* blib)
```

```
{
  arg_list_t* args = new arg_list_t();
  args->push_back(GRIPPER_HOLDING_OBJECT);
  return !(blib->Sensors["gripper"]->eval(args));
}

int main(int argc, char **argv)
{
  // our experiment
  Genetics* exp;
  parse_args(argc,argv);

  try
  {
    // same as before, but encapsulated into Genetics
  exp = new Genetics();
  exp->blib->registerCustomBehaviour("approach_block",
  new ApproachBlockBehaviour() );
  exp->blib->registerSensor("blob", new bbl::BlobSensor());
  exp->blib->registerSensor("motor", new bbl::MotorSensor());
  exp->blib->registerSensor("sonar", new bbl::SonarSensor());
  exp->blib->registerSensor("gripper", new bbl::GripperSensor());
  // set the abort condition
  exp->blib->setAbortCondition(&gripper_check_abort);
  // set the fitness function
  exp->setFitnessFunction(&fitness_function);
  // load the experiment configuration file (gripper.exp)
  exp->loadExperiment(gExperiment);
  // begin!
  exp->startExperiment();
  }
  catch (PlayerCc::PlayerError e)
  {
  delete exp;
    return -1;
  }
   return 0;
}
```

Note that it is not needed to start the Player server manually, the Bebal-library takes care of that now.
After running this program multiple simulations will be run according to the settings specified in Section 4.11.
The abort function *bool gripper_check_abort(bLib* blib)* is nonobligatory in general, but in this case required as the fitness function measures the time it took to forage the block, so all individuals would get the same fitness if the simulation would end only on a robots timeout. It is possible to define custom abort conditions by defining a function following this template:

```
bool my_custom_abort_function(bbl::bLib* b){
  // returns true if the simulation should be aborted
}
```

and setting it in the experiment by calling
`setAbortCondition(bool (*custom_abort_condition)(bLib* blib))`.

Directions on how to build the controller are unchanged.

5.7 Results & Logging

To draw conclusions from simulations various data can be logged. This could be used to optimize the controller or reviewing the progress of average fitness of the population. For this purpose for every experimental run a folder named *run_xxxx* is created in a parent folder that is set with the *LOG_FOLDER* variable in the experiment configuration file as described in Section 4.11. *xxxx* stands for a four-digit number that is automatically incremented with every run. The folder contains three types of files:

1. A file named *results.csv*, which is a common .csv (comma-separated values) file that can be imported into OpenOffice or MS Excel. After the experiment ends, it should contain five columns with altogether as many rows as many *GENERATIONS* were defined in the configuration file:

   (a) generation – a generation counter.

   (b) min fitness – the minimal fitness encountered in the current generation.

(c) average fitness – the average fitness calculated of all individuals in the current generation.

(d) max fitness – the maximal fitness encountered in the current generation.

(e) time – sum of the durations that it took to simulate all individuals in the current generation.

2. The *run.log* file contains a copy of the output the program has written to the console, formatted in log4cxx [5.7] style.

3. Moreover for each generation a file called *xxxx.gen* is created, that logs for each individual the time (in milliseconds) needed to evaluate it, the resulting fitness and the genetic code used in following manner:
   i:time, i:fitness, i:GENE_1,i:GENE_2,...i:GENE_N
   where $i \in 0, ..., POPULATION\_SIZE$.
   GENE_1 to GENE_N are the values of the variables/genes that were selected to evolve over time as defined in Section 4.11.

The resulting folder-tree structure with *LOG_FOLDER* set to "log/test" will have the following form:

```
log/test
  |- run_0001
      |- 0001.gen
      |- 0002.gen
      |- 0003.gen
      |- ...
      |- nnnn.gen
      |- results.csv
      |- run.log
  |- run_0002
      |- 0001.gen
      |- 0002.gen
      |- 0003.gen
      |- ...
      |- nnnn.gen
      |- results.csv
      |- run.log
  |- run_0003
  ...
```

log4cxx

Interaction with the user after launching a Bebal-control program is done through the *log4cxx* library, which is similar to the popular *log4j* library for Java. It serves for displaying the progress of the experiment and other various informational and warning/error messages as text in the console. A variety of options can be set in a configuration file *logging.properties*, all of them documented in detail in [10]. The *logging.properties* file has to be in the same directory as the executables home-folder or, if omitted, the Bebal-library defaults to standard settings.

Chapter 6

Experiments

The Bebal-library offers a variety of functions for designing vertical behaviour-based controllers and optimizing them with genetic algorithms. As a proof-of-concept an advanced experiment was conducted aside of the more or less trivial examples listed in the previous chapters. The experiment had two goals – proving that it is possible with the Bebal-library to create a complex behaviour-based controller with a minimal amount of programming and that it can be taken a step further by letting the genetic algorithm to optimize its parameters. Results are shown in the next sections together with a detailed description of the experiment.

6.1 Experimental Design

The experimental setup was based on the work of the authors in [13]. Figure 6.1 depicts an arena of 24×24 meters simulated by Player/Stage. In that arena the robots were given a task of constructing a (preferably straight) wall from solid green blocks they had to find first. A total amount of 10 blocks was seeded in the location of the arena (marked by the number 1 in Fig. 6.1) opposite to the building site (number 3). The building site was marked with a yellow rectangle on the wall. A single green seed block positioned just in front of the middle of the rectangle signalized the wall-building location.
Robots were equipped with a laser scanner, a ring of sonar sensors, a gripper for grabbing blocks and a color camera. The number of robots in the setup varied from 1, 2 up to 4. For each of these numbers the genetic algorithm tried to optimize certain controller parameters (see 6.1.2) so the interference

between the robots would be minimal and they would build the wall faster. The genetic algorithm was set to terminate after 50 generations.
To sort out slow individuals in the genetic pool of the population a simulation run was limited to 180 second of real time, corresponding to about 15 minutes of simulated time. The trials with one and two robots have had a limit of 260 seconds (about 17 minutes of simulated time) as the 180 second limit was insufficient for usable results. This was also due to the weaker hardware configuration of the computer the latter trials were performed on, where the ratio of simulated to real time could not be set as high as on the faster computer. This ratio is set by the `interval_real` and `interval_sim` variables for the Stage simulator as discusses in Section 4.3.1. It was found out, that a dual-core processor clocked at 2.33 GHz can handle four simultaneously simulated robots at a `interval_real` setting of 15 (a speedup of factor 6), whereas a single-core processor at 1.6GHz managed only two robots at the ratio of 20 (speedup factor of 5). The `interval_sim` value was left at 100. Setting lower values resulted in erroneous behaviour of Player/Stage and sensor readings.

6.1.1 The Controller

Figure 6.1 with a progressing experimental trial shows the basic idea of the solution of the construction task: We let the robot find a block in the area they are seeded (number 1) by wandering more or less randomly through the arena. After a block is foraged the robot tries to find a wall and follows it in a clockwise direction (2) until the construction site is reached and recognized. Finally the robot gets into position and approaches the tail of the wall to be constructed (3). The block the robot is carrying is then dropped and the construction site is left, beginning againg with the first step.
The controller defines behaviours that can be seen as tasks a robot can perform. The following list explains briefly what each behaviour is responsible for:

- The **wander** behaviour does nothing more than causing the robot to drive straight until an obstacle or an object of interest is found.
- **Avoid obstacles** is triggered whenever the robot is on a collision course with an obstacle.
- **Approach block** behaviour – if the robot spots a green block with his camera, he tries to collect it (as long as he is not carrying one already).

- **Find** and **Follow wall** behaviours are responsible for bringing the robot to the construction site by following the walls of the arena in a clockwise direction.
- The **On construction site** behaviour recognizes when the robot arrived at the construction site driving him in a position from where he can comfortably attach the block to the wall.
- With the robot in position, the **Build wall** drops the block and calls the **Leave site** behaviour that navigates the robot away from the construction site.

This list is simplified and does not represent the position of a behaviour in the vertical behaviour hierarchy of the controller. The actual detailed controller consists of the files `construction.xml` and `construction.var` and can be found on the CD in the corresponding folder.

Common problems experienced while constructing the controller were to make a distinction between a wall and a green block and the robots getting stuck as they tried to approach a wall in a clockwise direction. The first problem was solved by dividing the sonar sensors facing left (as seen from the robot) into a western and a north-western section. The follow wall behaviour was triggered only if the sonar detected an obstacle in both sections at the same time. The green block was too small to be detected by more than one sonar sensor. The wall-approach problem was resolved easily by setting the threshold for the sonar sensors on the right side slightly higher than on the left side.
Also robots currently not carrying a block were trying to "steal" another robots block, resulting mostly in a crash. This was solved by detecting the red grippers attached to the robots with the camera. If a green block was spotted in the vicinity of a red object, it was forbidden and rather avoided than approached.
In trials with multiple robots the interference between them grew and they were hindering each other when following a wall or on the construction site. Therefore when a robot detected with the laser scanner another robot in front of him a random waiting interval helped to decrease the interference.

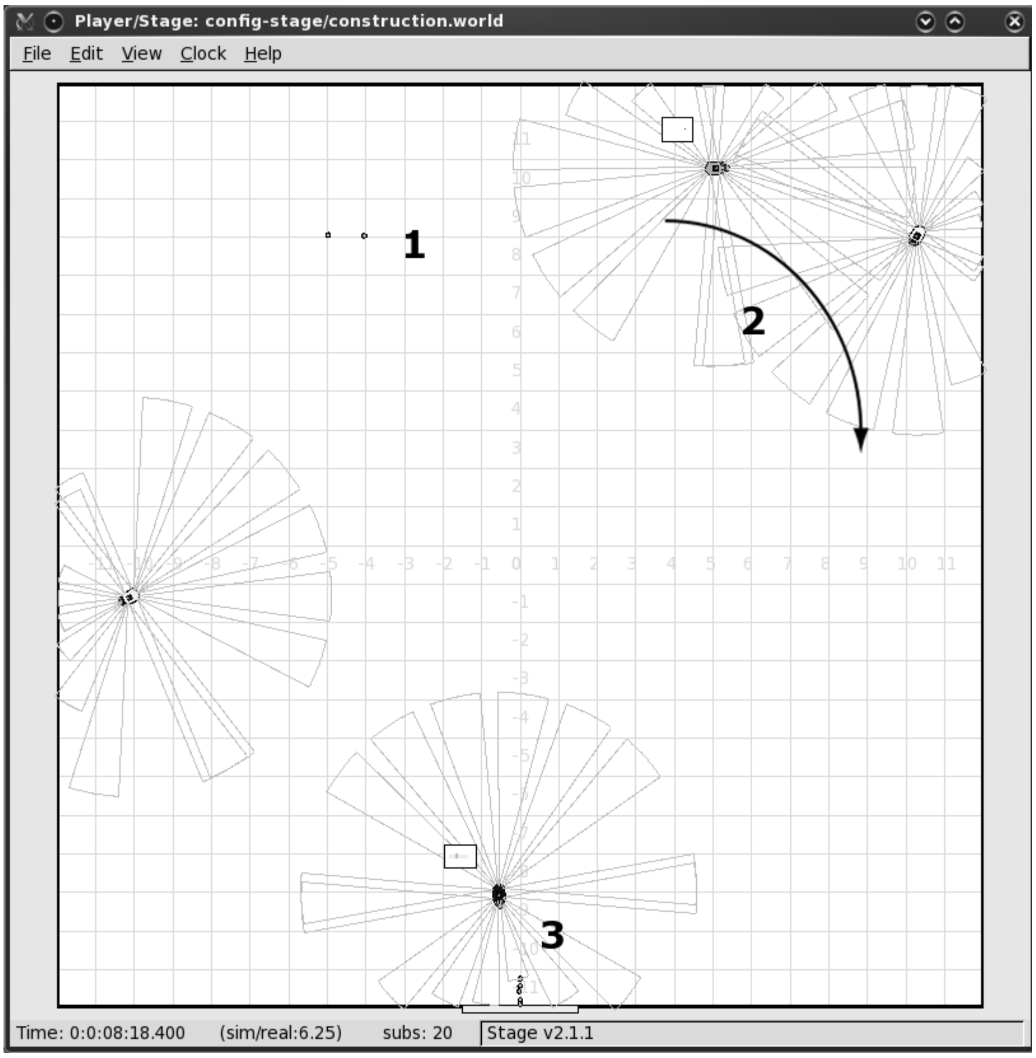
Player/Stage: config-stage/construction.world
File Edit View Clock Help
1
2
3
Time: 0:0:08:18.400 (sim/real:6.25) subs: 20 Stage v2.1.1

Figure 6.1: A running experiment in Player/Stage with four robots. The number **1** marks the origin of green blocks used for construction, **2** shows robots following the wall clockwise, **3** is the construction site with a wall being built in progress. The rays originating at the robots body are a visualization of its sonar sensors readings.

6.1.2 Optimizing the Controller

Since the original controller was handmade with a heuristic approach, a good performance under all circumstances is not guaranteed. Settings that function well for one or two robots may deliver poor performance with an increased amount of robots. This is due to the increased interaction between them as the probability of meeting a co-working robot rises. Part of the experiment was to enhance the controllers performance for a growing number of robots by letting a genetic algorithm optimize its parameters for each setting.
The parameters chosen to represent the genetic code of an individual were selected by determining on how much a parameter influences overall speed and conflict resolution, e.g. two robots following the same wall with a short distance in-between. Parameters of the controller selected from the variables-file of the experiment (`construction.var`) were:

- `VAR_BUILD_SITE_SPEED`: Responsible for how fast the robot approaches the construction site.
- `VAR_APPROACH_BLOCK_SPEED_COEF`: Determines the speed when approaching a green block by dividing the distance to the block with the coefficient (therefore a smaller coefficient causes the robot to approach the block faster and the other way around).
- `VAR_WANDER_SPEED`: Defines the speed of the robot when in wandering behaviour.
- `VAR_FOLLOW_WALL_LEFT_SPEED`: Sets the speed when following a wall and the robot gets too near to it.
- `VAR_FOLLOW_WALL_LEFT_ANGLE`: Sets the angular speed when following a wall and the robot gets too near to it.
- `VAR_FOLLOW_WALL_RIGHT_SPEED`: Sets the speed when following a wall and the robot gets too far from it.
- `VAR_LASER_AVOID_ROBOT_DIST`[1]: The distance at which the robot detects other robots in front of him when carrying a block. When a fellow robot is detected the robot waits a random time and tries to avoid the other one.

- VAR_RANDOM_WAIT_MIN[1]: Sets the minimal duration of the waiting interval (see previous variable).

- VAR_RANDOM_WAIT_MAX[1]: Sets the maximal duration of the waiting interval.

6.1.3 The Fitness Function

The fitness function (2.5.3) was defined as follows:

$$\text{fitness} = \frac{\texttt{MAXIMAL\_RUNTIME}}{\texttt{ACTUAL\_RUNTIME}} * 100 * \texttt{NR\_OF\_BLOCKS\_DROPPED}$$

This ensured an evaluation of individuals based on how many blocks they collected (represented by NR_OF_BLOCKS_DROPPED) and built the wall from. Each successfully transferred block was rated with 100 “points” . The ratio of the maximal runtime of a simulation trial (MAXIMAL_RUNTIME) and the actual time taken (ACTUAL_RUNTIME) could give the fitness a boost in case the individual managed to forage all blocks within the time limit (fitness values of 1000 and better). Bringing in the time in the equation creates a pressure on finding a better solution even if all blocks were collected.

6.2 Experimental Results

The simulated experiment showed that the Bebal-library is capable of enhancing a behaviour-based controller by using a genetic algorithm to find optimal values for the parameters of the controller. This is based on the results of the wall construction experiment, which will be analysed in detail in the following subsections.
Figures 6.2, 6.3 and 6.4 contain charts with the evolving fitness values of 1, 2 and 4 simultaneously working robots in progress of generations. All graphs have three plotted curves representing the minimal (dashed light grey curve), average (black curve) and maximal fitness (dashed dark grey curve) values of individuals in each generation. In addition trend lines (in corresponding colors) were inserted for a better display of the momentum of the fitness evolution.

[1]Does not apply to trials with only one robot.

Table 6.1 summarize the fitness of the first and last ten generations for all experimental setups. Evaluated was the minimal, average and maximal fitness reached by average for each amount of robots.

Trials with one Robot

This trial served as a reference point to how many blocks a single robot can forage in a given time (about 15 simulated minutes) when no interference from other robots occurs. From log files created during the experiment and data in Figure 6.2/Table 6.1 we can see that the maximal amount an individual in the late phase of the experiment has managed to build a wall from is on average nearly 4 blocks (3.78). This is an improvement of almost 50% compared to the 2 blocks (2.38) in the first ten generations. The average performance however improved only slightly over 50 generations, from 1.8 blocks to 2.5. This is due to the limitation on how fast the robot can move at all and the time limit set.

Trials with two Robots

In this experiment interestingly the average number of collected blocks did not change at all (3.5 in early generation to 3.4 in the end). This can be explained with an almost non-existent interference between the robots as there were only two, so basically just the performance of two robots was totaled. Individuals evolved that were able to collect a correctly deliver 6 blocks. Measured values for the trials with two robots are depicted in Fig. 6.3.

Trials with four Robots

The four robot trial was interesting to observe as for the first time some individuals were able to collect and correctly deliver all ten blocks to the construction site, represented by fitness values of ± 1000 and higher) in Fig. 6.4. Furthermore the genetic algorithm worked as supposed. It evolved over time settings for the controller which minimize the interference between robots and increase the average amount of block from 6.6 to over 8 (see Table 6.1). More importantly there were individuals in almost each generation that collected all 10 blocks and over the generations the best of them even finished about 3 (simulated) minutes before the time limit of 15 minutes

Average fitness values			
# of robots		**First 10 gen.**	**Last 10 gen.**
	min	79.2	135.0
1	*avrg*	*180.17*	*253.8*
	max	237.6	368.64
	min	198.0	108.0
2	*avrg*	*350.34*	*340.71*
	max	475.2	531.0
	min	261.1	378
4	*avrg*	*665.56*	*802.04*
	max	1098.3	1254.64

Table 6.1: Measured results of each trial. Compared is the average fitness of the first and last ten generations. A succesfully transferred block was rated with 100 points (see 6.1.3).

expired. They got gradually better (the maximal-curve in the graph) over the generations and reached in the end fitness values of over 1400.
It is worth noting that signs of a new behaviour emerged in most of the best individuals evolved by the genetic algorithm. These individuals exhibited a "bouncing" behaviour when following a wall as depicted in Figure 6.5 caused by setting the escape angular speed (defined by the `VAR_FOLLOW_WALL_LEFT_ANGLE` variable) to almost the allowed maximum of 20 degrees. Obviously it allowed them to proceed faster and avoid other robots. This behaviour was also partly observed in trials with one and two robots.

Conclusion of the Experiment

The three experiment variatons delivered useful results and it was shown that an increasing number of robots improves the performance of the system. Better controllers were evolved, best observed in the experiment with four robots. It may be possible to further increase the performance of the used controller by modifying the controller and/or using other settings for the genetic algorithm. This is left open to further reserach as it was not the aim of this experiment to bring the controller to perfection, but to prove a concept and show that the Bebal-library can be used in such experiments while producting useful results. Beyond that expectations, indication of evolving behaviours was discovered.

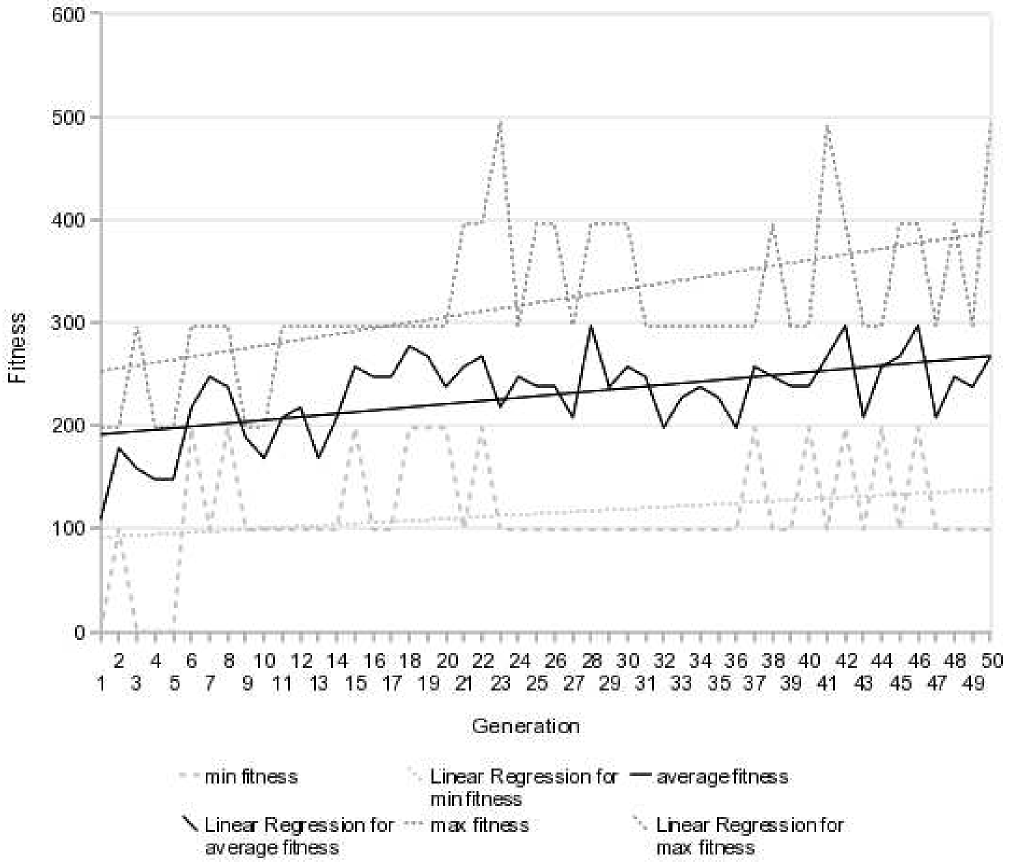

Figure 6.2: Results of trials with one robot.

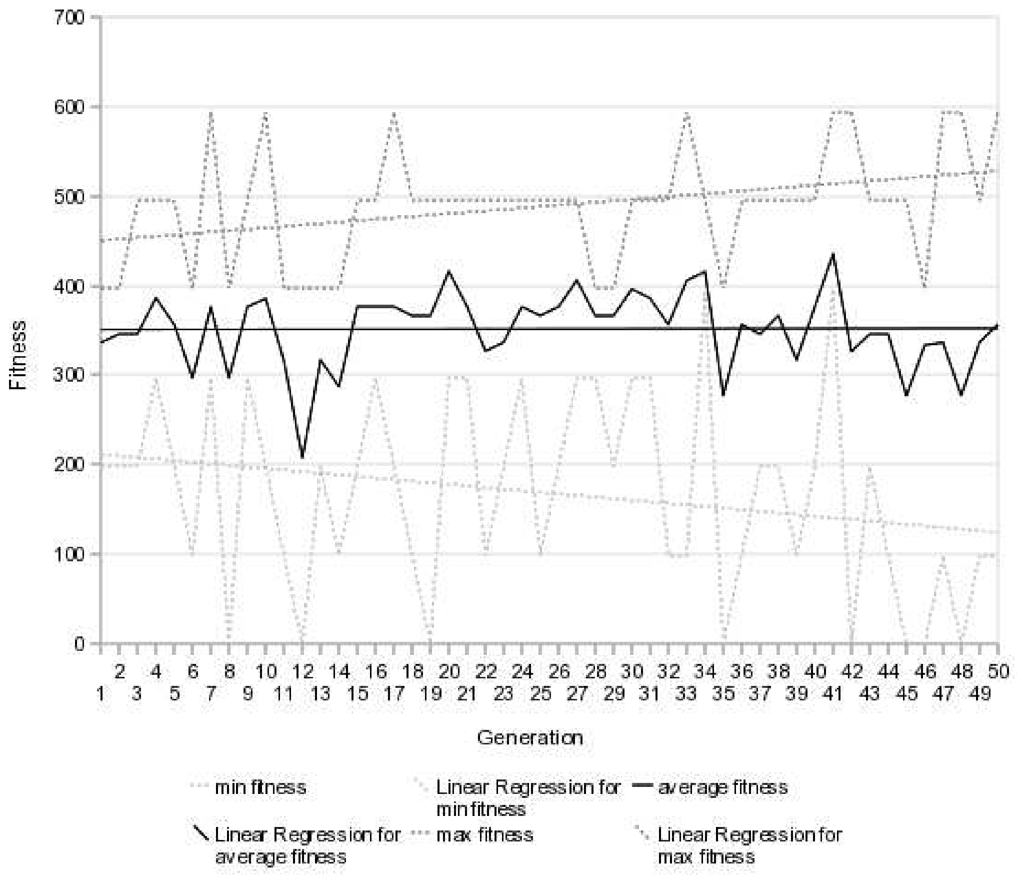

Figure 6.3: Results of trials with two robots.

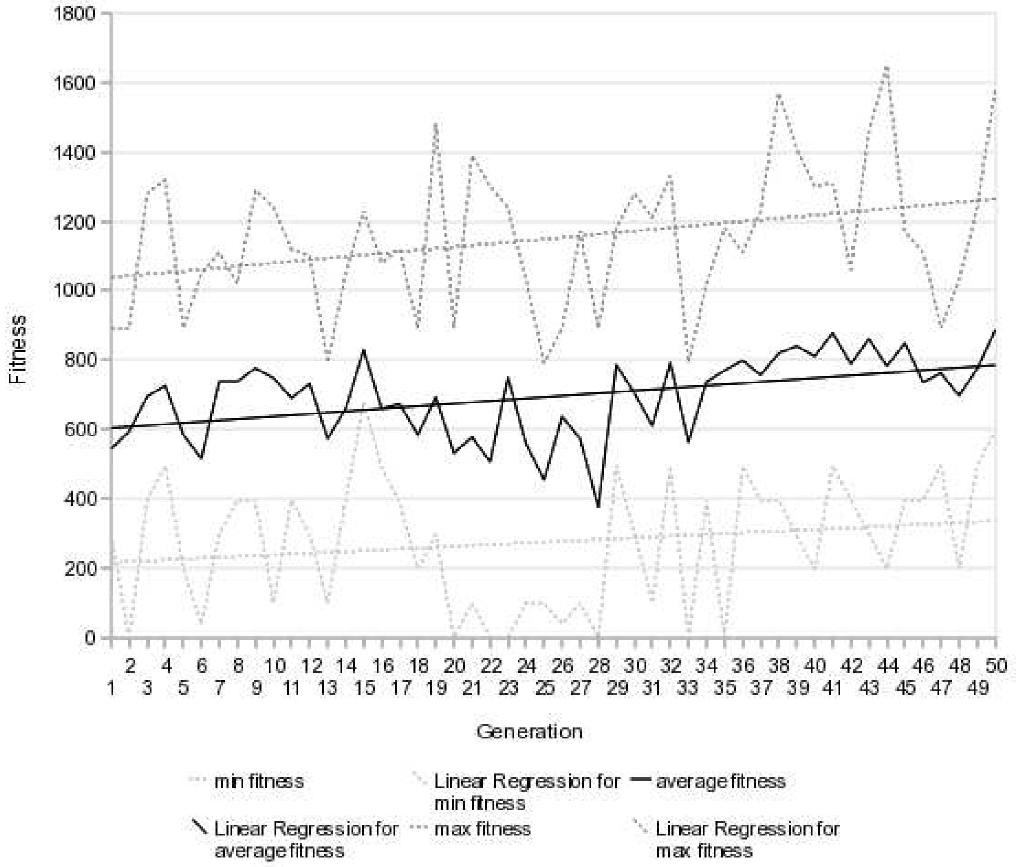

Figure 6.4: Results of trials with four robots.

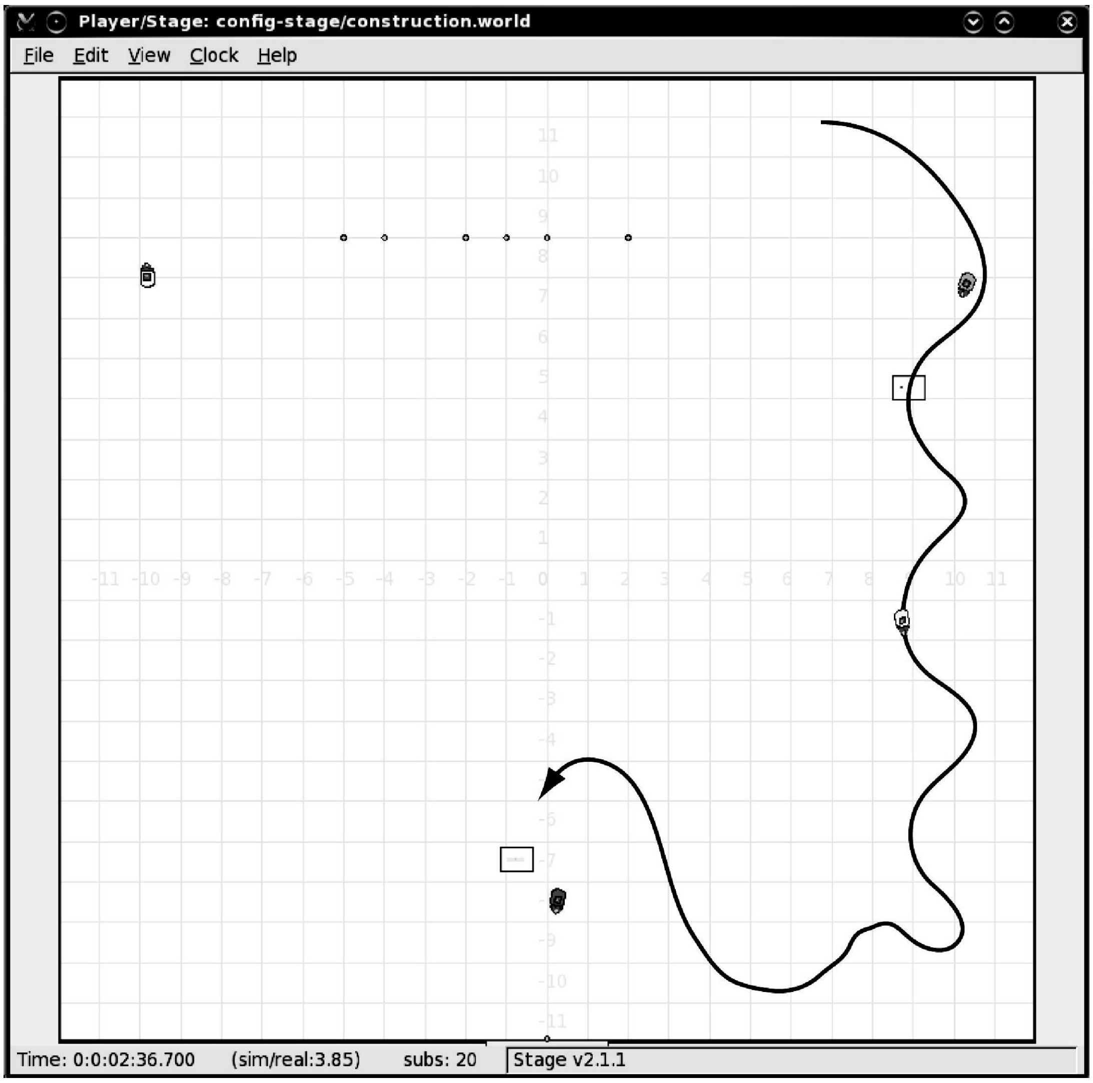
Player/Stage: config-stage/construction.world
File Edit View Clock Help
Time: 0:0:02:36.700 (sim/real:3.85) subs: 20
Stage v2.1.1

Figure 6.5: Player/Stage with four robots. The black line illustrates the movement along a wall of some succesfull individuals evolved by the genetic algorithm.

Chapter 7

Conclusion

The goal of the thesis was an implementation of a software library that allows to develop behaviour-based controllers for autonomous mobile robots simulated by Player/Stage. The controller design process had to be simple, efficient and preferably avoid programming whenever possible in order to shift the focus into defining behaviours and rules for them. In addition the library should implement genetic algorithms. Functionality had to be exhibited by performing an evolutionary experiment with multiple robots.

All the above objectives were completed, including the experiment that validates this approach to user-friendly creation of robot controllers. The Bebal-library allows the user to design controllers in its simplest form by creating only one XML-file in which behaviours of a robot are described. Parametrization of such rules and behaviours enables further to use another functionality of the library – genetic algorithms – for optimizing these parameters in order to improve the controllers. If needed, interfaces to write behaviours and sensor evaluation routines as modules in C++ are present.

The ability to function was illustrated by a series of experiments as described in Chapter 6. At first a nontrivial robots controller was created to solve a construction assignment in which the robots had to collect blocks and build a straight wall at a specified location. In the second phase we let the genetic algorithm optimize the controllers parameters for one, two and four robots working at the same time. The results of the experiment show that the library is capable of evolving parameters of a controller and achieve higher performance of the controller than its performance with original parameter values set manually. The average amount of wall-blocks collected increased from 6 to 8. Solutions that managed to forage all 10 blocks before the time limit of 15 (simulated) minutes have been found. It was even possible to observe an emerging behaviour as the genetic algorithm progressed.

This work could be further enhanced in various ways, such as by enabling to evolve not only parameters of certain behaviours but the behaviours themselves or by adding a selection of behaviour-based controller types (vertical versus horizontal, cooperative versus competitive). Experiments on hardware robots could be performed to see how the library can cope with noise and other real world problems. From a more technical point of view it is possible to support not only the Player/Stage simulator but other simulators, too. Various additional sensor interfaces that Player provides can be implemented. Also a graphical interface (GUI) could be proven useful for even easier management of settings and definition of behaviours.

Appendix A

Contents of the CD

The attached CD contains the following files and folders:

- **/construction_experiment_results** – Contains the results and log files of the experiment conducted in Chapter 6.
- **/doc** – Contains this thesis in digital form as well a programmer's reference generated by Doxygen.
- **/examples** – In this folder there are source and configurations files of the examples used through the text.
- **/software** – Contains the necessary software to install and run the Player/Stage simulator.
- **libbebal-0.1.0.tar.gz** – This is the software package of the Bebal-library that can be installed. When extracted it contains makefile routines needed to build the library and the source code.

Bibliography

[1] http://playerstage.sourceforge.net/index.php?src=player

[2] http://playerstage.sourceforge.net/index.php?src=stage

[3] http://playerstage.sourceforge.net/index.php?src=doc

[4] http://www.w3.org/XML/

[5] http://www.stack.nl/∼dimitri/doxygen/

[6] http://www.linux.com

[7] http://sourceforge.net/projects/opencvlibrary/

[8] http://xmlsoft.org

[9] http://www.boost.org

[10] http://logging.apache.org/log4cxx/

[11] Floreano D., Nolfi S.: *Evolutionary Robotics: Exploiting the full power of selforganization*, MIT Press, 2000.

[12] Arkin R.C.: *Behavior-Based Robotics*, MIT Press, 1998.

[13] Jens Wawerla, Gaurav S. Sukhatme, Maja J. Matarić: *Collective Construction with Multiple Robots*. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2002), 2002.