Developing A Behavior Engine for the Fawkes Robot-Control Software and its Adaptation to the Humanoid Platform Nao

Diploma Thesis
by
Tim Niemüller

Submitted in April 2009
Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Aachen, April 2009  ...........................................
(Tim Niemüller)
Acknowledgments

First of all, I would like to thank Alexander Ferrein and Professor Lakemeyer for the opportunity to work on robot software. Especially the freedom that they provided during the development of ideas and code was a driving factor for Fawkes happening. I also want to thank the AllemaniACs RoboCup team for many great events and cooperative team work, especially Stefan Schiffer, Masrur Doostdar, Daniel Beck, Vaishak Belle, Stefan Jacobs, Andreas Strack, Lutz Hermanns and Carsten Gester. Working with Stefan and Masrur during many competitions for service robots was an enlightening experience. I want to express my gratitude to my cooking friends, especially to Christian Buck and Tobias Weyand, for reading and discussing the thesis. I am very grateful to Anne van Deest for her patience in all those years, when it was just another RoboCup event and only a few more hours in the lab. I would also like to thank my family for their support and interest in weird machines playing soccer.
## Contents

1 Introduction ........................................... 1

2 Background ........................................... 3
   2.1 Mobile Robotics and Robots .................. 3
      2.1.1 Robot Behavior .......................... 4
      2.1.2 Robot Platform .......................... 4
      2.1.3 The Humanoid Robot Nao ............... 4
      2.1.4 The AllemaniACs Middle Size League Robot 7
   2.2 RoboCup .......................................... 8
      2.2.1 Middle Size League (MSL) ............. 10
      2.2.2 Standard Platform League (SPL) .... 10
      2.2.3 RoboCup@Home League .................. 11
   2.3 Robot Software Framework Characteristics .. 11
      2.3.1 Terminology ............................... 12
      2.3.2 Characteristics of a Mobile Robot Framework 13
   2.4 Behavior Execution ............................. 14
      2.4.1 Execution Paradigms .................... 15
      2.4.2 Hybrid Deliberative-Reactive Paradigm 16
   2.5 AllemaniACs RoboCup Team ..................... 17
   2.6 ZaDeAt – South African-German-Austrian Research Group 18

3 Related Work ......................................... 19
   3.1 Robot Software Frameworks .................. 19
      3.1.1 Saphira .................................. 20
      3.1.2 Miro ...................................... 20
      3.1.3 Orocos ................................... 21
      3.1.4 Orca ...................................... 22
      3.1.5 RoboFrame ................................ 22
      3.1.6 Player .................................... 23
      3.1.7 NaoQi .................................... 24
   3.2 Behavior Execution Implementations ....... 25
      3.2.1 PRS-lite .................................. 25
      3.2.2 Colbert ................................... 26
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3 URBI</td>
<td>26</td>
</tr>
<tr>
<td>3.2.4 XABSL</td>
<td>27</td>
</tr>
<tr>
<td>3.2.5 RoboCup Team Description Papers</td>
<td>28</td>
</tr>
<tr>
<td>3.3 Agent Modeling with UML Statecharts and Hybrid Automata</td>
<td>29</td>
</tr>
<tr>
<td>4 The Fawkes Robot Software Framework</td>
<td>31</td>
</tr>
<tr>
<td>4.1 Decision for a new Framework</td>
<td>31</td>
</tr>
<tr>
<td>4.2 The Fawkes Framework</td>
<td>32</td>
</tr>
<tr>
<td>4.2.1 Component-based Design</td>
<td>34</td>
</tr>
<tr>
<td>4.2.2 Plugins and Threads</td>
<td>34</td>
</tr>
<tr>
<td>4.2.3 Main Loop</td>
<td>35</td>
</tr>
<tr>
<td>4.2.4 Aspects</td>
<td>36</td>
</tr>
<tr>
<td>4.2.5 (Soft) Guarantees</td>
<td>38</td>
</tr>
<tr>
<td>4.2.6 Blackboard</td>
<td>39</td>
</tr>
<tr>
<td>4.2.7 FireVision</td>
<td>40</td>
</tr>
<tr>
<td>4.2.8 Configuration</td>
<td>41</td>
</tr>
<tr>
<td>4.2.9 Networking</td>
<td>41</td>
</tr>
<tr>
<td>4.2.10 Central Clock</td>
<td>42</td>
</tr>
<tr>
<td>4.2.11 Logging</td>
<td>43</td>
</tr>
<tr>
<td>4.2.12 Main Loop Timing Constraints</td>
<td>43</td>
</tr>
<tr>
<td>4.2.13 Webview Plugin</td>
<td>44</td>
</tr>
<tr>
<td>4.3 Fawkes on the Nao</td>
<td>44</td>
</tr>
<tr>
<td>4.4 Fawkes on the AllemaniACs MSL Robot</td>
<td>48</td>
</tr>
<tr>
<td>5 Theoretical Foundations for Skills</td>
<td>51</td>
</tr>
<tr>
<td>5.1 Hybrid Behavior Architecture</td>
<td>51</td>
</tr>
<tr>
<td>5.2 Low-level Control, Behaviors and Agents</td>
<td>52</td>
</tr>
<tr>
<td>5.3 Modeling Skills with Hybrid State Machines</td>
<td>54</td>
</tr>
<tr>
<td>5.3.1 Hybrid State Machines</td>
<td>55</td>
</tr>
<tr>
<td>5.3.2 Set of Skills for a Particular Skill Space</td>
<td>56</td>
</tr>
<tr>
<td>5.3.3 Skill Hybrid State Machines</td>
<td>56</td>
</tr>
<tr>
<td>5.3.4 Example Specific for the Nao search_ball Skill</td>
<td>58</td>
</tr>
<tr>
<td>6 Implementation of a Lua-based Behavior Engine</td>
<td>59</td>
</tr>
<tr>
<td>6.1 Design Decisions</td>
<td>59</td>
</tr>
<tr>
<td>6.2 Lua</td>
<td>60</td>
</tr>
<tr>
<td>6.2.1 Values and Types</td>
<td>61</td>
</tr>
<tr>
<td>6.2.2 Tables and Metatables</td>
<td>62</td>
</tr>
<tr>
<td>6.2.3 First-Class Functions and Proper Tail Calls</td>
<td>62</td>
</tr>
<tr>
<td>6.2.4 Functables</td>
<td>63</td>
</tr>
<tr>
<td>6.2.5 Function Closures</td>
<td>63</td>
</tr>
<tr>
<td>6.2.6 Object-Oriented Programming in Lua</td>
<td>63</td>
</tr>
<tr>
<td>6.2.7 Threads and Coroutines</td>
<td>64</td>
</tr>
</tbody>
</table>
CONTENTS

6.2.8 Integration of Lua with C ............................................. 65
6.2.9 Modules and Packages ................................................. 65
6.2.10 Register-based Virtual Machine .................................... 66
6.2.11 Incremental Mark-And-Sweep Garbage Collector ................. 67
6.3 Providing C++ Features to Lua ......................................... 67
6.4 Behavior Engine .......................................................... 69
  6.4.1 Skill Execution Run-Time Plugin .................................. 69
  6.4.2 Lua Environment ...................................................... 71
  6.4.3 Skills ........................................................................ 72
  6.4.4 Skill Execution, Sandboxing and Agent Communication ....... 74
  6.4.5 Predicates .............................................................. 77
6.5 Skills Employing Skill Hybrid State Machines ....................... 79
  6.5.1 Finite State Machine Driver and Basic States ................. 79
  6.5.2 Jump States and Jump Conditions .................................. 82
  6.5.3 Skill Hybrid State Machine ......................................... 82
  6.5.4 Implementation of a Skill Using a SHSM ....................... 83
6.6 Skills for the Target Platforms and Domain ......................... 84
  6.6.1 Nao Skill to Search for the Ball .................................. 84
  6.6.2 MSL Goalie Skill ...................................................... 86
6.7 Skill Development Tools .................................................. 87
  6.7.1 Graph Generation via Graphviz ................................... 88
  6.7.2 SkillGUI ............................................................. 88
  6.7.3 Simulator Integration ................................................ 90
6.8 Reactive Lua Soccer Agent for the Nao .............................. 92
7 Evaluation ........................................................................... 95
  7.1 Evaluation Challenges .................................................... 95
  7.2 Fawkes Robot Software Framework .................................. 96
    7.2.1 Fawkes Software Characteristics ............................... 96
    7.2.2 CPU and Memory Assessment ................................... 98
    7.2.3 Run-Time Measurement on MSL and Nao Platforms ....... 100
  7.3 Lua-Based Behavior Engine .............................................. 105
    7.3.1 Skills for the Nao and MSL Robots ............................. 105
    7.3.2 Code Sharing Possibilities and Limits ......................... 107
    7.3.3 Tests with the Nao Soccer Agent ............................... 108
8 Summary and Future Work .................................................. 111
  8.1 Summary ................................................................. 111
  8.2 Future Work .............................................................. 113

References ............................................................................ 115
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Humanoid Robot Nao – Important Hardware</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>The AllemaniACs MSL Robot – Important Hardware</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Robots used in RoboCup</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Behavior Execution Paradigms</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>AllemaniACs Logo</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>ZaDeAt Logo</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Fawkes Application Structure</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Modern CPU Dies</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Main Loop</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Fawkes BlackBoard – Conceptual Overview</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Fawkes Component Configuration for the Nao</td>
<td>45</td>
</tr>
<tr>
<td>4.6</td>
<td>Fawkes Component Configuration for the MSL Robot</td>
<td>49</td>
</tr>
<tr>
<td>5.1</td>
<td>Behavior Levels in the Fawkes Software Stack</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Thermostat Hybrid Automaton</td>
<td>55</td>
</tr>
<tr>
<td>6.1</td>
<td>Lua Logo</td>
<td>60</td>
</tr>
<tr>
<td>6.2</td>
<td>Wrapping C++ instances in Lua</td>
<td>68</td>
</tr>
<tr>
<td>6.3</td>
<td>Graph of the search_ball skill for the Nao robot</td>
<td>86</td>
</tr>
<tr>
<td>6.4</td>
<td>SkillGUI: a GUI to execute, visualize and debug skills</td>
<td>89</td>
</tr>
<tr>
<td>6.5</td>
<td>The Webots Simulator with two Nao teams</td>
<td>91</td>
</tr>
<tr>
<td>6.6</td>
<td>Nao Soccer Lua Agent Graph</td>
<td>93</td>
</tr>
<tr>
<td>7.1</td>
<td>Timing of the Main Loop Hooks of the Nao SPL Robot</td>
<td>101</td>
</tr>
<tr>
<td>7.2</td>
<td>Timing of the Main Loop Hooks of the MSL Robot</td>
<td>101</td>
</tr>
<tr>
<td>7.3</td>
<td>CPU and Memory Benchmarks for MSL and Nao Robot Platforms</td>
<td>101</td>
</tr>
<tr>
<td>7.4</td>
<td>Nao Agent Lab Test Run</td>
<td>109</td>
</tr>
</tbody>
</table>
List of Listings & Tables

Listings

6.1 Lua Tail Call and Named Arguments .......................... 62
6.2 Lua Funtable .................................................. 63
6.3 Lua Closure .................................................. 63
6.4 Lua Object-Oriented Programming Example .................. 64
6.5 Lua C Module .................................................. 65
6.6 Lua Code for Virtual Machine Example ....................... 66
6.7 Code translated for register-based Virtual Machine ......... 66
6.8 Code translated for stack-based Virtual Machine .......... 66
6.9 Skill Set Definition for the Nao Platform .................... 70
6.10 Skill Set Definition for the MSL Robot Platform ........... 71
6.11 Skill Wrapper Generator Function ........................... 74
6.12 Predicate Library Module ................................... 78
6.13 PLM Application ............................................ 78
6.14 Finite State Machine Driver (excerpt) ....................... 81
6.15 Search Ball Skill using a Skill Hybrid State Machine ..... 85
6.16 Simple Goalie Skill for the MSL Robot ...................... 87
6.17 Code for the Nao soccer agent ............................... 94
7.1 Row Matrix-Vector Multip. ................................... 99
7.2 Column Matrix-Vector Multip. ................................ 99

Tables

7.1 Selected skills implemented for the Nao and MSL robots .... 105
Chapter 1

Introduction

Robots are machines that exist to perform specific functions or tasks in the physical world. Often, this is associated with immobile robots doing labor-intensive and high-precision work for example in the automobile industry. For mobile robots, which can move within the environment of operation, typical associations are search and rescue robots for earthquake scenarios or aerial or underwater drones for surveillance and research operations. Most of the time these robots are tele-operated by a human operator. Especially for embodied artificial intelligence (AI) research, robots are developed which act autonomously. They perceive the environment, use their cognitive abilities and programs to decide what to do on their own and use their actuators to manipulate the physical world. Moreover an autonomous mobile robot can move within the physical world, deciding by itself where to go, and how to get there. The mapping of sensory inputs to a pattern of actions and executing them constitutes the behavior.

To carry out any such task the robot needs a programming system that allows to develop programs from which the desired behavior emerges when executed. The behavior is usually composed in multiple layers of varying complexity, timing constraints, and commemoration features.

In this thesis we concentrate on autonomous mobile robots and a particular part of the behavior programming system. We give a definition to structure the behavior in three distinct levels, the low-level control layer interacting with the hardware, a high-level strategic decision layer, and an intermediate reactive layer that connects the low and high level subsystems. We focus on this intermediate layer, where we implement a behavior engine as an environment where reactive behaviors are executed and monitored. These behaviors are reactive execution entities – called skills – that fulfill a specific well-defined function, based on sensor inputs, making only local decisions, but that do not employ deliberation methods like planning, or make global decisions about the overall task solving and strategy. Skills are operated in the behavior engine and we propose a modeling approach using extended hybrid state machines for a
formal description of skills.

The behavior engine should be applicable to multiple platforms and domains. Two different platforms have been chosen for our experiments which act in the same domain, autonomous soccer playing in the RoboCup testbed. The robot platforms used in this thesis are the wheeled Middle Size League robot and the humanoid robot Nao. Two platforms have been chosen to evaluate the applicability of the behavior engine on different robots. The common robot soccer domain, although in separate leagues with different rule sets, allows us to explore the possibilities and limits of a unified behavior engine.

For a common behavior engine, we also need a common robot software framework on both robots. During the last two years, we have developed the Fawkes robot software framework for the Middle Size League robots. For this thesis, we adapt the software for the humanoid robot Nao and evaluate its applicability to the Nao’s more constrained computing system.

Therefore, this thesis has two major topics, the Fawkes robot software framework emphasizing extensions made during this thesis for its adaptation to the humanoid robot Nao, and the behavior engine which is used to execute and monitor skills, which are modeled using hybrid state machines. The behavior engine is implemented as a plugin for Fawkes and employs the Lua scripting language. We argue why a scripting language in general, and Lua in particular are useful for this task and how the formal model of skills has been implemented in the software system. The primary platform is the Nao, as it is made accessible for the Fawkes software for the first time. The secondary platform is the Middle Size League robot, for which large parts of the base software for sensor and actuator access already exists. It is mainly used for the evaluation of the behavior engine on different platforms, but still in the same domain. The thesis is embedded into the ZaDeAt project, an international research cooperation of the RWTH Aachen University, the University of Cape Town and the Graz University of Technology.

In Chapter 2 we describe the background of this thesis, which includes a short introduction to mobile robotics, robot platforms in general and the employed robots in particular, the RoboCup competitive testbed, general remarks about robot software frameworks and robot behavior execution, the AllemaniACs RoboCup Team and the ZaDeAt research group. We describe other works related to this thesis in Chapter 3. The Fawkes robot software framework and its application to the target robot platforms is detailed in Chapter 4. In Chapter 5 we present the theoretical foundations of the behavior engine by defining robot behavior levels and the chosen modeling approach for skills. The implementation of the behavior engine and example skills are depicted in Chapter 6. In Chapter 7 we evaluate the Fawkes robot software framework, especially considering the adaptation to the Nao platform, and the behavior engine and its applicability to different platforms and domains. We conclude with a summary and an outlook to future work in Chapter 8.
Chapter 2

Background

In this chapter we give background information about our work. First in Section 2.1 we will describe mobile robotics, robot behavior, and what kind of robots this thesis covers. Afterwards we will introduce the RoboCup domain in Section 2.2 and the particular leagues and robots this thesis is about. In Section 2.3 we will give general considerations about robot software frameworks and evaluation criteria thereof. We conclude the chapter in Section 2.4 with general remarks about robot behavior execution.

2.1 Mobile Robotics and Robots

Bekey describes a robot as a machine that senses, thinks, and acts. More specifically autonomous robots are capable of operating in the real-world environment without any form of external control for extended periods of time. Further mobile robots are capable of moving in their environment, e.g. by means of wheels or legs, aerial or underwater, and interact with it for example by grasping objects and relocating them [1].

In this thesis we will concentrate on autonomous mobile robots. These robots act on their own to fulfill a given task. For this they are equipped with sensors to perceive the environment, a computing unit to make decisions on actions based on the perception and one or more actuators to carry out this action. Specifically in the case of mobile robots we expect these robots to support some form of locomotion, which can be for example wheeled driving or legged walking. One of the original examples for such a robot is the robot Shakey developed in 1969 at the Stanford Research Institute (SRI). Shakey was equipped with a camera, a motor for wheeled movement and on-board logic to analyse the images and derive decisions based on the observation. It was the first robot to apply the sense-think-act cycle described in more detail later.
2.1. Mobile Robotics and Robots

2.1.1 Robot Behavior

We want to use machines to accomplish certain tasks. To accomplish such a task, they need to perceive the environment and map the sensory inputs to a pattern of motor actions. This mapping is called behavior and it serves as the fundamental component of intelligence in robot systems [2].

2.1.2 Robot Platform

We need a way to describe a robot, but maybe not each detail of the particular robot. When we speak about a robot platform we refer to several characteristics of a robot or a group of robots. The most dominant factors of a robot platform are the form of locomotion, the available sensors and actuators, the physical dimensions and the available computing system. The software system is not part of the platform. Sometimes the computing unit runs a specific operating system (OS) that cannot be easily replaced because particular hardware drivers are only available for that specific system. In this case the OS sometimes is taken into account when describing a platform. However, the used control software and programming environment, although often tightly coupled with a particular robot platform, is not part of the platform itself.

Robots which differ in details of their hardware can still be accounted to the same robot platform. An example might be two robots based on the Pioneer [3] robot platform with a differential drive and ultrasonic sensors. They might differ in their payload, for instance one might have a robotic arm mounted on the base while the other has not. They are still accounted to the same platform.

Often the terms robot platform, platform and robot are used interchangeably. In this thesis if we speak about a robot this denotes the robot platform, and not a specific instance unless stated otherwise.

In the remainder of this section we describe two specific robot platforms that are of particular interest in this thesis. This requires references to particular RoboCup leagues, which are explained a little later in Section 2.2.

2.1.3 The Humanoid Robot Nao

The humanoid robot Nao is the new platform for the Standard Platform League (see below for more information about the league). In early 2006 Sony announced the discontinuation of the Aibo robotic dog, which was the robot platform for the Four-Legged League. A public call for a new robot platform was issued by the RoboCup organization. Several companies applied and at the RoboCup 2007 in Atlanta, USA, the Nao robot, developed by the French company Aldebaran, was chosen and the new Standard Platform League league was formed. Until today RoboCup teams had access to three different versions of the robot. Version 1 was delivered to 16 teams, selected of numerous
applicants, by the end of 2007 and was replaced in April 2008 by an improved version 2. With this version the first world championship in this league was held during RoboCup 2008 in Suzhou, China. The robot was very fragile and in order to prevent damage robots were not allowed to touch each other. Several design issues became obvious during the competitions leading to high rates of repair returns, which Aldebaran processed in a “Nao Clinic” on site. In January 2009, during the Aldebaran RoboCup Workshop 2009 in Paris, France, the first version 3 robots were handed out to the teams. These robots are a lot more robust than the previous versions were. They are now able to fall without breaking and to stand up on their own.

The Nao is a humanoid robot with 21 degrees of freedom (DoF). Figure 2.1 shows the current version of the robot. The 21 DoF are distributed as follows. The neck has two DoF for panning and tilting. Each arm has four DoF, two for the shoulder and two for the elbow. The hand and wrist are numb in the RoboCup version, for the full equipped robot (educational version) the hand can be opened and closed and turned around the wrist. Each leg has five DoF, two in the hip, one in the knee and two in the ankle. The remaining DoF is a joint in the hip – which is split into two parts. These joints rotate the legs along the hip, but both sides are physically bound and thus only account for one DoF. The sensors consists of two cameras, one between the eye brows and one in the mouth. Both produce VGA resolution images and are based on a standard webcam chip. Only one camera can be used at a time and switching between them is a slow operation. Four ultrasonic sensors in the chest allow for detecting obstacles in front of the robot. In the chest there is an inertial
measurement unit (IMU). It has an accelerometer which measures accelerations along three axes (for a standing robot the directions are downward, forward and sideward), and a gyroscope with two axes (along the forward and sideward axes) to measure angular velocities. There are two bumper sensors in each foot to detect contact with an object, and four force sensitive resistors (FSR) which support limited measurement of the weight distribution on the feet. The robot features several microphones. In the RoboCup version there are two microphones, one on each side of the head near the speakers. In the full version there are additional microphones in the front and back of the head. The infrared transceivers in the eyes are not used in the RoboCup version either. A button in the chest is used to turn the robot on and that can afterwards be used freely from within the software. The robot has two speakers, one on each side of the head. Several light emitting diodes (LEDs) on the robot can be used for status indication, these are eight full-color LEDs in each eye, ten 16-step blue LEDs in the ears, one full-color LED around the chest button and one full-color LED in each foot. A full-color LED is a triple of a red, blue and green LED which can be mixed to produce arbitrary colors. The robot has colored body parts which can be exchanged and are used to indicate the team during RoboCup. See Figure 2.3(b) for an example of a Nao with red body parts and a soccer ball.

The Nao has a general purpose computing board in its head, which is the main computing unit. It features an AMD Geode\textsuperscript{1} LX-800 running at 500 MHz, 256 MB of RAM and 1 GB of flash memory as internal USB\textsuperscript{2} stick. It employs Wi-Fi IEEE 802.11 b/g for wireless communication and supports Ethernet for cabled high-speed connections, for instance to copy data to a laptop. On the board runs a modified version of OpenEmbedded Linux [4], called OpenNao. In the chest there is a computing board based on an ARM CPU\textsuperscript{3} of unknown type. This CPU cannot be used directly and is exclusively used by internal firmware not accessible for developers. Many micro controllers are distributed in the robot and control servo joints and monitor the hardware. All micro controllers are connected to the ARM board. The Geode board communicates with the ARM board via USB. Because the protocol has not been disclosed access to the hardware is only possible through proprietary software.

The Geode CPU is quite limited in its computing abilities. Not only is the micro architecture outdated and the processing speed very low compared to more modern embedded CPUs, but additionally the memory bandwidth and cache sizes are very small which mandates careful programming and allows only for restricted algorithms and methods to solve robotics tasks like localization and object recognition. The basic motion control software provided

\textsuperscript{1}CPU architecture by AMD compatible to Intel’s x86, cf. \url{http://www.amd.com/geode}
\textsuperscript{2}Universal Serial Bus, accessory device bus
\textsuperscript{3}CPU architecture by ARM Ltd., chip design company, cf. \url{http://www.arm.com}
by Aldebaran already consumes about 20% of the CPU resources for real-time execution of joint commands, even when the robot is not moving, which is significant. Aldebaran has expressed the intention on a workshop to move part or all of this load to the ARM CPU.

In summary the Nao has evolved into an interesting, robust and stable platform. Although it is apparent that this is still a development project and not a final product it is a good base for scientific research in robotics. The earlier versions suffered from the fragile hardware and many points of hardware failure, but this has been vastly improved so in 2009 we will probably see robot soccer games in the Standard Platform League without as much human interference as 2008. The major problem of the platform remains the software that the vendor forces on every user of the robot platform.

### 2.1.4 The AllemaniACs Middle Size League Robot

The AllemaniACs Middle Size League (MSL) robot has been developed in a joint venture of the Knowledge-based Systems Group (KBSG) and the Institute for Engineering Design (IKT) of the RWTH Aachen University. It was created from scratch and has several features nowadays necessary to be able to compete in the MSL and some unique innovations.

Figure 2.2 shows the robot and its most important hardware features, while Figure 2.3(a) shows a schema drawing of an intermediate development stage. The most notable features of the platform are a holonomic omni-directional drive and camera system. The omni-directional drive consists of three so-called omni wheels (also known as Swedish wheels [5]) on a equilateral trian-
2.2. RoboCup

The wheels have small discs or sub-wheels around the circumference that can roll perpendicular to the rolling direction [6]. This allows driving into any direction on the 2D ground plane without holonomic constraints. To achieve omni-directional image acquisition a camera faces upwards towards a spherical mirror. So it is possible to take 360° round-view images in every frame. The camera used is a Pike by Allied Vision Technologies with a resolution of 1000×1000 pixels. The quadratic image sensor size is especially useful for this application as it offers a maximum fit of the round mirror in the image in both, forward and sideward directions. A light shade has been added to shield the camera from field lighting. Additionally a Bumblebee-2 stereo-vision camera is mounted facing forward to the ground. The stereo image processing should allow for better detection of obstacles and features in front of the robot, such as for a generic soccer ball and not only an orange one. However, this feature has not been extensively tested and causes a high computational burden on the system. The robot has an original and innovative new pneumatic-based kicking and ball guidance device [7]. Three kick devices are mounted on the robot’s front. A stronger pneumatic cylinder is facing forward and allows for a high kick, shooting the ball over an opponent player. Two pneumatic muscles are positioned at angles of about 40° on the left and right side of the cylinder. They can be used to shoot the ball to the left and the right to trick an opponent and pass to another team member; or shoot forward flat on the ground when both are used at the same time. A ball guidance mechanism has been added that consists of two small arms, only one of which is outstretched at any time. The intention is to keep the ball in front of the robot when driving on an arc with the ball. The kicker and ball guidance device is controlled via an IO Warrior board that is connected via USB. It features a micro controller and proprietary wiring for valve switching. Two air tanks are used to power the kicker. They are filled using a compressor before the game and last for about two dozen kicks. The robot wears black textile, because it has to be mostly black according to the MSL rules.

The computing unit is a MiniITX board based on laptop hardware, featuring an Intel Core 2 Duo CPU at 2 GHz, 2 GB of RAM and an 8 GB flash drive. The cameras are connected via IEEE 1394 A (Bumblebee-2) and B (Pike). A Gigabit Ethernet port can be used to copy large amounts of data, while a wireless connection employing Wi-Fi 802.11a can be used for robot interconnection and sending instructions from a laptop. The system runs Fedora Linux 10.

2.2 RoboCup

A popular application domain for mobile robotics and our main testbed is RoboCup [8]. It provides a competitive environment where researchers and engineers from all over the world benchmark their robots and software sys-
By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.

This shows that the RoboCup is a way to foster research and development of robotics systems and software and to provide a common test environment. Teams from universities and companies all over the globe present and compare different approaches to tasks like locomotion, localization, and behavior execution and monitoring. There is one large annual event – the RoboCup world championship – accompanied by several smaller events around the globe like the German Open at the Hannover Fair.

The competition is structured in several different leagues concentrating on certain aspects of robotics. The dominant leagues are about autonomous soccer playing robots. Other leagues focus more on applied robotics like the Rescue League where robots operate in a disaster area and the RoboCup@Home League where robots assist in a household. We will concentrate on two soccer leagues and the RoboCup@Home League where we have applied our software system and behavior engine. See Figure 2.3 for pictures of example robots of the particular leagues.

In these leagues robots compete in terms of fastest and most robust perception, self-localization, decision making and speedy execution. A dominant factor driving the development during the last few years in the prevalent RoboCup robot soccer leagues like the Middle Size League is the increasing speed by which the games are played. Although at the moment the humanoid leagues and especially the new Standard Platform League are comparatively
quite slow we expect the games to become a lot faster within a few years time. Recently techniques developed for the robot soccer domain have been transferred to the RoboCup@Home competition [10].

For robots playing soccer or interacting with humans in a household we have real-time constraints for the robot’s reactions. For instance, if a human steps in the pathway of a driving robot, the robot has to stop or drive around the human, but it may not harm the human under any circumstances. Therefore it is important that the information from the sensors is evaluated quickly to detect the obstacle and the decision to stop is made as fast as possible.

We will now describe the particular RoboCup leagues which are most relevant for this thesis.

2.2.1 Middle Size League (MSL)

In this league robots of the size of about 40 cm × 40 cm play on a soccer field of 18 m × 12 m. The robots are custom-built by the teams and usually support holonomic motion and feature an omni-directional vision system which allows for a 360° round view in every frame. Over the last few years the speed has steadily increased and the precision for example for shooting the ball has vastly been improved. Additionally the most successful robots support team play that allows for passing between the robots or placing a robot in front of the goal before another robot kicks the ball towards it. The robot of the AllemaniACs RoboCup team was described earlier in Section 2.1.4. A particular challenge in this league is the high speed by which the games are played. Constraints to ease the development of the robots like colored goals or field border markers as localization features have been removed during the last years to increase the complexity stepwise. In 2009 for the first time the goals are plain white instead of blue and yellow, and colored-coded corner poles have been removed. These features were used before to resolve the symmetry of the soccer field. This is not possible anymore and other ways need to be found to cope with the symmetry. More and more teams show cooperative team play like passing or short ball touching during indirect free kicks.

2.2.2 Standard Platform League (SPL)

This is a new robot soccer league succeeding the former Four-Legged League (FLL) which was based on the Aibo robotic dog. In this league the robots are standardized and no hardware modifications are allowed. The new platform is the humanoid robot Nao by Aldebaran Robotics, we described earlier in Section 2.1.3. It is about 60 cm high and plays on a field of 6 m × 4 m. As the league is rather new teams are generally still struggling with basic problems like robust and stable locomotion, efficient computer vision for localization
and feature extraction. Teams that already participated in the FLL have an advantage as they can transfer their knowledge gained in this league to the SPL. Besides new code being written or FLL software being adapted many teams are consolidating their software system to achieve robustness to have a continuous game play over some time.

2.2.3 RoboCup@Home League

In this league robots assist in a household and fulfill typical tasks. These tasks include interacting with the human, e.g. following or guiding the human, communicating by speech or gesture and fetching items from the environment. Imminent problems to solve are safety and reliability, and human-robot interaction. In this league most prevalent are the requirements phrased in Asimov's laws, mostly important the first one: “A robot may never harm a human being” [11, 1]. While the robot is moving and working in the environment it must always react to humans, and injuring or even only touching a human must be avoided in any case. Additionally while moving in the environment the robot may not damage, for instance, the furniture or anything else in the environment. Although this league is not a defined target for this thesis work, experiments have been performed to test the applicability of the proposed framework and behavior engine in this league as well. The robot is shown in Figure 2.3(c).

2.3 Robot Software Framework Characteristics

Mobile robots have to fulfill a wide range of tasks from low-level sensing, data processing and actuator control to high-level decision making and strategic planning. All these tasks have to be integrated into a unified software system. Several different approaches to building software systems for robots have been pursued. Before we are going to describe some of these in more detail, we will give some general considerations about robot software frameworks, middleware and architectures.

Robot software systems (RSS) are in most cases bundled as a self-contained universe concerned with all aspects of building robotic systems [12]. They come with means of internal and external communication, provide a programming or configuration interface to plug together an application that can access sensors and effectors, share data among different modules, and sometimes provide a runtime system.
2.3. ROBOT SOFTWARE FRAMEWORK CHARACTERISTICS

2.3.1 Terminology

In the following we explain the terminology for the description of the different building blocks and abstraction layers for RSS based on literature references and our own understanding.

**Component.** A component is defined as a binary unit of deployment that implements one or more well-defined interfaces to provide access to an interrelated set of functionality configurable without access to the source code [13, 14]. It adheres to a specified “contract” and expects certain input data and produces and provides specified output data. The contract also states what the component expects from its context, e.g. certain timing constraints. Often, the interfaces required in a particular system architecture are not exactly the interfaces offered by component, and likewise the system sometimes does not provide exactly the interface a component provides. In these cases the components need to be put around a matching “facade”, a set of interfaces that most closely resembles the features of the component in terms of the available interfaces[15].

**Module – Library** A module or library is a coherent set of implemented functionality, using an object-oriented data encapsulation as useful (but not necessary) design paradigm. A module can be compiled separately, and is portable to different platforms given compatible compiler and operating system support. Modules inherently hardware-dependent, such as device drivers, are often not portable [15].

**System Architecture.** The system architecture is a specific choice of functional building blocks (“components”), in order to build a software system that performs according to a specification (“architecture in the large”) [15]. In [1] the architecture represents the structure of the software, the way in which the robot processes sensor inputs, performs cognitive functions, and provides signals to output actuators and thus the architecture concerns the practical structure of a robot’s software.

**Framework.** A framework is a design and an implementation providing a possible solution in a specific problem domain [15]. It is used to model a particular domain or an important aspect thereof [16]. Frameworks are similar to software libraries, that means they are reusable abstractions of code wrapped in a well-defined API. Unlike libraries, however, the overall program’s flow of control is not dictated by the caller, but by the framework. This inversion of control is a distinguishing feature of software frameworks [16, 17].

**Component Architecture.** The component architecture determines the internal design of one single component in order to guarantee that the component performs according to its external interface/contract (“architecture in the small”) [15].
The application domain is mobile robotics. Mobile robots, as explained in Section 2.1, are machines that move and work in the real world and accomplish specified tasks. A robot software framework (RSF) should provide functionality and tools to support the implementation of the desired architecture for a particular robot control system. The vast amount of available frameworks shows that several different ways of providing the required infrastructure have emerged in the course of time.

2.3.2 Characteristics of a Mobile Robot Framework

There exists a plethora of different robot frameworks today. On the one hand different approaches how a robot should be controlled emerged, and on the other hand different methods how modules are combined and connected were developed during the last decades. In [18] certain characteristics are defined for the evaluation of software frameworks that we summarize below.

Robot hardware abstraction. The framework should not be tailor-made for a specific robot platform, but it should rather be portable to a variety of platforms. To reach this goal the actuator control and sensory input at the lowest level need to be interchangeable. Although the hardware may differ (for example one robot could have differential and another a synchro-drive locomotion), the fundamental principle of a component is often similar. Another aspect is the independence of the underlying operating system, which is closely coupled to the hardware and available drivers.

Extensibility and scalability. The robot framework must be able to easily incorporate new software modules and to use hardware newly added to the robot. This is a very important feature since in research environments new software modules are added frequently and hardware is constantly modified. Especially in robotics a wide variety of features has to be integrated and it is unlikely that a single team can do everything that is needed. Therefore the framework must be easily extensible by external software.

Scalability requires efficient communication and a well planned data flow. Additionally it may be desirable to allow for distribution of the computation over several machines to scale the software system. Nowadays the ability to exploit multi-core systems is an important scalability feature.

Limited run-time overhead. The run-time overhead can be measured in terms of memory and CPU requirements, i.e. the frequency by which the control loops are executed, and end-to-end latency, meaning the time that is required for a sensor reading to have an effect on the actuator command.

Actuator control model. The actuator control model is twofold. For one the overall structure of a robot software system imposes a certain preferred model of control (cf. Section 2.4.1). For the other this means how a single actuator module composes the actuator commands. For the latter different
actuator models should be possible and implemented per module, especially keeping in mind that the hardware is exchangeable and may require different approaches based on the capabilities. On the framework level, often the control model influences the structure of the software. However, in the prevalent hybrid reactive-deliberative control model (cf. Section 2.4.2) some flexibility should be provided that allows for multiple different approaches to be implemented.

**Software characteristics.** Software for mobile robots shares the common requirements for good software, like completeness, simplicity, correctness, and consistency. Especially in robotics the “worse is better” paradigm [19] is often required to deal with the complexity of robot systems. This means that often integration of third-party software has to start early and then has to be refined over several iterations of the development cycle. Different approaches in terms of structure, programming style and interfaces have to be incorporated. Additionally in mobile robotics there are real-time constraints, which means that the robot software needs to be able to react fast to exogenous events.

**Tools and methods.** Robotics is a rapidly changing field with high complexity. New technologies have to be integrated at a high pace and new procedures are taught to the robot to solve new tasks. Particularly because of this complexity modern tools are required that support software development. The framework itself must provide tools for data inspection, monitoring and debugging requiring reliable and efficient communication of the different tools.

**Documentation.** To achieve wide acceptance for a software platform rigorous documentation is crucial. The documentation has to be written on different levels. The philosophy behind the architecture has to be made clear, the programmers must have a reference manual and application programming interface (API) documentation, and operators of a robot need a user guide on how to start and control the software. Documentation can happen with in-code documentation tools like Doxygen\(^4\) and comments, as wiki-based collaborative documentation, or classical as printed manual.

## 2.4 Behavior Execution

Robots are only useful if they can accomplish tasks, usually when ordered by humans or when a specified event happens. To enable a robot doing so it must have a software component that, given a specified goal, produces actuator commands based on sensor input to fulfill the goal, usually with several intermediate steps. Sometimes this might even involve doing something not obviously leading to the goal, like taking a longer way to the target because

---

\(^4\)Doxygen: [http://www.doxygen.org](http://www.doxygen.org)
the shortest way is blocked, or first grasping an object that has to be displaced to reach the actually desired object.

### 2.4.1 Execution Paradigms

In the past several paradigms have emerged by which robot behavior is generated, executed and monitored. Three different paradigms have gained the most influence over the years. The paradigms describe the relationship between the primitives perception, cognition and acting. We will now describe the paradigms based on [2]. They are depicted in Figure 2.4.

Now classical is the hierarchical paradigm shown in Figure 2.4(a), often referred to as sense-plan-act (SPA) approach. There are two significant architectural features [20]. First the flow of control among these is unidirectional and linear. Second the execution of the plan is analogous to the execution of a computer program. Both, plan and program execution, are built using primitives composed of partial orderings, conditionals, and loops. During a run of the loop first the environment is perceived by some sensors. The data is then processed, for example by integrating the information into a world model and coming up with a plan what to do to reach the overall goal. Then the plan is executed. The intelligence of the system depends on the planner or the programmer, not the execution mechanism [20].

In 1986 Brooks came up with a radical departure from SPA, the subsumption architecture [21]. Brooks argued that reasoning based on the symbol system hypothesis were fundamentally flawed because it could never give an adequate model of the world, stating that the world is its own best model [22]. In contrast he proposed direct coupling from sensor readings to actuator commands. This is often referred to as the reactive paradigm, shown in Figure 2.4(b). In the subsumption architecture many simple behaviors run at the same time and based on an activation value a behavior can suppress or inhibit others. Although the paradigm produced exciting results and clever robots, it quickly became clear that throwing away planning was too extreme for a general purpose robot. Yet it has several good properties, especially the fast execution times.

![Figure 2.4: Behavior Execution Paradigms](image-url)
This lead to the \textit{hybrid deliberative-reactive paradigm} (or \textit{hybrid paradigm} for short), outlined in Figure 2.4(c), by augmenting the reactive approach with a planner that deliberates and creates an execution policy. This policy is then executed and monitored by a reactive execution system. Thus both is combined, the fast and efficient execution of the reactive paradigm and the flexible and powerful planning of the hierarchical paradigm. The paradigm is also referred to as \textit{three-layer architecture} or \textit{three-tier architecture}. The hybrid paradigm is currently the most used one for robot behavior execution and has been applied in this thesis. Therefore we discuss it in more detail in the following section.

2.4.2 Hybrid Deliberative-Reactive Paradigm

Because of the specific importance we describe it in more detail. The reactive paradigm enabled robots to navigate and avoid obstacles in real-time because of the small overhead. However, the robot could not plan optimal trajectories because all remembering and reasoning had been eliminated, and it would only react locally striving somehow into the desired direction. This example shows the shortcomings of the purely reactive approach. Robots could not make maps, monitor their own performance, or even select the best behaviors to accomplish a task (general planning). To differentiate the simple functions, that just need a kind of state information from the more cognitively oriented functions, the latter are referred to as \textit{deliberative components} [2]. Nowadays the hybrid paradigm is regarded as the compromise combining the best of the hierarchical and reactive paradigms. With asynchronous processing deliberative components can be executed concurrently to the rest of the system for example for longer term strategy determination. At the same time quickly executed reactive behaviors with short update cycles control the robot according to the last strategic goal [2].

In hybrid systems the sensing operation is generally more complex. The data acquired from sensors is required in multiple places. The reactive behaviors and the deliberative component both require the sensor data, the latter possibly integrates all the data into a global world model. This is important to note when designing the information routing system of a RSS.

The hybrid paradigm is an extension of the reactive paradigm. But although it might seem that the behavioral component that stems from the reactive paradigm is left unchanged, it slightly differs in the hybrid paradigm. In the former the behaviors were purely reflexive. In the hybrid paradigm however the behaviors can be reflexive as well as innate or learned. We will later define behaviors as skills, to avoid confusion with purely reflexive behaviors, as suggested in [2].

The terms \textit{local} and \textit{global} have a specific meaning in hybrid systems. Often \textit{global} is used synonymously with \textit{deliberative}, and \textit{local} with \textit{reactive}. The de-
liberative component of a hybrid system often contains modules that require a global world model and that cannot easily be represented in the reactive behaviors. Reactive behaviors in most cases operate on sensor data, regarded as local information, often without specific typing information, e.g. a robot equipped with a laser range finder can avoid obstacles based purely on the distance reading without knowing if it is a human, a wall, or another robot which is in the way.

Three major hybrid architecture designs are described in [2]. The differences are usually how the architecture distinguishes between reaction and deliberation, how responsibilities are organized in the deliberative portion and how the overall behavior emerges. In the managerial architecture agents are located at the top for high-level planning. They pass a plan to subordinates, who refine the plan and gather resources and then pass those down to low-level reactive behaviors for execution. State-hierarchy architectures organize activities by the scope of time knowledge. This architecture usually has three layers representing the different times present, past and future. It has a different amount of state information that exists in each layer, hence the name. The layers are organized according to whether they contain no state (lowest level), contain state reflecting memories about the past, or contain state reflecting predictions about the future (highest level) [20]. The model-oriented architectures evolved from more traditional AI concepts of symbolic manipulation around a global world model. This is conceptually a step back to the hierarchical paradigm, but the world models had become less ambitious and more cleverly organized. The ability to compensate different speeds in perceptual routines by asynchronous execution alleviates the problem that the generation of the world model blocks the whole system. Fast reactive behaviors can even access the sensor data directly without waiting for the world model. Additionally with more computational power a central filtering of the data provides much better results [2].

2.5 AllemaniACs RoboCup Team

The AllemaniACs RoboCup team was founded in 2001 when the effort to build and program soccer robots was started at the Knowledge-based Systems Group (KBSG) of the RWTH Aachen University and was initially funded as part of the Priority Program 1125 (“Cooperative Multi-robot Teams in highly dynamic domains”) of the German National Science Foundation. The team participated in the 2D and 3D simulation soccer leagues, the Mid-Size and Standard Platform robot soc-
2.6 ZaDeAt – South African-German-Austrian Research Group

The thesis is embedded into an international research project called ZaDeAt. The name is based on the abbreviations of the names of the participating countries, South Africa, Germany, and Austria. The basis forms a bilateral research agreement between the RWTH Aachen University, Germany and the University of Cape Town (UCT), South Africa [23]. The Technical University of Graz joined the effort during the approval phase. The goal is to develop a competitive team for the Standard Platform League based on the Nao platform. The project was initiated in 2007 after a visit of Alexander Ferrein and the author to the INSITE 2006\(^5\) trade fair in Johannesburg for demonstration of the AllemaniACs robot for the RoboCup@Home League. The work is distributed among the different parts of the team based on the components envisioned for the software system (cf. Chapter 4). The locomotion is to be developed at UCT, the localization and odometry calculation will be written at TU Graz, while the base framework and behavior control will be provided by the RWTH Aachen University, of which a large part has been conducted during this thesis.

In the following, after referencing related work for robot software frameworks, architectures and behavior execution in Chapter 3, we describe the implemented robot software framework Fawkes and its conceptual background based on the hybrid-reactive paradigm in Chapter 4 and continue with the theoretical foundations of the behavior engine in Chapter 5, and explain the implementation in Chapter 6.

Chapter 3

Related Work

In this section we give an overview of the related work for this thesis. For one these are works about robot software frameworks which we will describe in Section 3.1. And for the other these are concepts and implementations of behavior execution, described in Section 3.2. We will also talk about a particular approach for modeling an agent’s behavior in more detail in Section 3.3, that is of particular interest for this thesis (cf. Chapter 5).

3.1 Robot Software Frameworks

Many software frameworks for mobile robots have been developed in the past. Often when a robotics project is started, a software that attributes to the specifics of the robot platform is required. Especially in the beginning of the mobile robotics era that meant writing a software specific to this one platform. But it was not until the 1990s that software frameworks emerged which were targeted to multiple robot platforms and that achieved widespread acceptance.

Today there are several such general frameworks, each with a specific design philosophy and technological background. These at least slightly different approaches made developers start over as the existing software did not fit their needs or a different mindset was followed.

One of the major problems with re-usability is that robot software is developed for a specific concept of module interconnection. A certain communication component is expected. Integrating a part of the functionality of one framework means that the concept of communication of this framework has to be used, integrated or connected to the framework the functionality should be made available for, which is often cumbersome.

We will now describe several frameworks that have been proposed in the past.
3.1.1 Saphira

*Saphira* [24] is a classical example for a robot software framework. It was developed by Konolige et al. in the mid-1990s at the Stanford Research Institute. Given the speed of development in computer science and robotics the framework might seem outdated, but even in today’s frameworks you see ideas which date back to the Saphira framework. This makes it worthwhile to have a closer look at Saphira.

The first target robot was Flakey, successor of one of the first robots, Shakey. Saphira was developed with three major goals. First *coordination* between different layers of the robot system had to be achieved. Secondly a strong internal representation of the world was chosen to accomplish *coherence* in the internal conception of the environment. And last the robot should provide means of natural interaction for *communication* with the user. This means the ability to understand task commands, as well as integrate advice about the environment or the desired behavior.

To achieve these goals Saphira is equipped with kind of a central world model, the Local Perceptual Space (LPS). The LPS is a geometric representation of the space around the robot. It integrated data from the robot’s sensors and allowed for adding *artifacts* – internal representations of objects produced from interpretations of the sensor data. The LPS is the core component around which the other components are grouped and was meant to provide the coherence of the system. The coordination is split into three layers, two of which are described in [24]. At the control level Saphira employs a behavior-based approach (cf. Section 2.4). The overall control problem is decomposed into smaller units, called basic actions. Examples are obstacle avoidance and corridor following. Basic actions are written and composed using fuzzy logic. Behaviors can be turned on and off by the higher layer. On the second layer there is a sequencer, which takes a number of goals and tries to compose sequences of basic actions to fulfill these goals. The original implementation of the sequencer used was PRS-lite, which was later replaced by Colbert (see Sections 3.2.1 and 3.2.2 for detailed information). On top of this a planner is located which does long-term deliberative reasoning. Here the planner can be thought of as an automatic generator of robot programs which are then executed by the sequencer. The planner is not discussed in [24].

Saphira is a closed source system, sold by MobileRobots, Inc. bundled with their robots, e.g. the Pioneer [3]. Therefore, the system cannot be adapted and used on other robots.

3.1.2 Miro

Several recent projects make use of CORBA [25] as a middleware for mobile robots. An example is *Miro* [26] which is a distributed object-oriented
framework and middleware. It concentrates on defining an object-oriented communication infrastructure based on CORBA. With CORBA functionality is provided by a set of objects that form a service. To use a service methods of its objects are invoked, often transparently over the network. Miro uses ACE TAO [27] as basis. On top of that a set of classes is provided that define services typically required on a robot system like locomotion and sensor access [28]. There is also a real-time software framework based on CORBA [29]. It uses prioritized communication queues to achieve short reaction times.

The main benefits of using a CORBA-based middleware is the possibility to use several different programming languages and the ability to distribute computations over many hosts transparently. Especially in the main domain of interest, in robotic soccer, all of the computation has to be done on the robot and off-loading is not allowed. Therefore it does not seem desirable to have a network based communication framework. Additionally the ability to use different programming languages is not such an advantage as it might seem. In the past we have seen that most of the development especially of low-level robotics software is done in C/C++, and for computationally intensive tasks it is likely that it will stay this way in the foreseeable future. It is only on the higher levels were timing and sometimes computational effort is less critical where simpler languages are a valuable addition.

There is some general criticism on the use of CORBA [30]. First and foremost the API is complex, inconsistent, and downright arcane and writing any non-trivial CORBA application is surprisingly difficult [30]. Other technologies such as SOAP [31], Enterprise JavaBeans or XML-RPC are used as replacements, especially because of their simplicity. The NaoQi software that is shipped with the Nao robot and that we are going to describe later also has chosen that path and uses SOAP for external communication.

3.1.3 Orocos

The Open Robot Control Software [32, 15], Orocos for short, is driven by the desire to standardize robotics software components and especially their interconnection. The aim is to provide comprehensive framework for robotics applications. Like Miro, Orocos relies on CORBA by means of ACE TAO for the communication and interconnection of components. Orocos however has reached a more wide-spread use compared to Miro. One reason for this is that Orocos comes with several base libraries that can be used aside from its interconnection philosophy. These are the Real-Time Toolkit (RTT), the

---

Bayesian Filtering Library (BFL), and the Kinematics and Dynamics Library (KDL). RTT provides the basis for the Orocos system. It provides abstraction classes for hardware and operating system feature access, defines the basic component model, optionally implements the CORBA communication and a runtime framework for (real-time) tasks. The BFL provides an application independent framework for inference in dynamic Bayesian networks with utilities like Kalman and particle filters. The KDL supports modelling and computation of kinematic chains like robotic arms or complete robots.

3.1.4 Orca

Orca [13] is a spin-off of Orocos. It was split off in 2004 when funding for the original project had ended. It has the same strong or even stronger focus on component-based software engineering. Instead of CORBA the middleware Ice [33] is used for communication and component interconnection. Ice is based on the experience with CORBA by simplifying the API and omitting features considered to be redundant or less useful. Although Orca was once understood as the Open Robot Controller Architecture it does not impose a certain architecture, like a central blackboard or a layered separation, but it rather aims to provide a framework with communication infrastructure and some common building blocks. Orca provides a base library called Hydra that contains drivers for robot hardware and implementations of algorithms commonly used in robotics such as path planning. While investigating Orca it became obvious that Orca provides many features for a wide range of usage scenarios. But the overall architecture and design is very loosely defined which make it cumbersome to get a definitive overview for the decision if Orca is applicable to the problem domain at hand. Orca components all run as separate processes and solely communicate over a network connection via Ice. Although Orca is more active in the research community and publishes papers especially about robot software engineering the framework is lacking coherence. It is fragmented into several pieces and thus seems not like an ideal candidate to build on.

3.1.5 RoboFrame

RoboFrame [34] is a software framework developed by the RoboCup team of TU Darmstadt. It was developed with very constrained hardware systems in mind, like a custom-built biped robot. Therefore no middleware was used for communication but proprietary internal message passing.

An application based on RoboFrame extends the software at predefined extension points. For this each application creates modules that are distributed across a number of threads. All modules of the application communicate by
sending messages. The messages are defined in the module source code, and not externally in a file. For larger chunks of data a blackboard is provided to distribute data. Messages are transmitted by a router component which handles the transfer of the messages between the sender and a number of receivers, both can reside in the same or different threads, or in different application instances possibly on different computers. Data stored in the blackboard can only be exchanged locally in the same application.

The problem with the definition of the communication interface in the module is that if multiple modules should provide the same interface, for instance, two different implementations of a localization module, both modules need to be updated if the interface is changed. This prevents well-defined interfaces which can be easily re-used. The framework is semi-closed, meaning that it is sent on request, but it is explicitly limited to non-commercial and educational purposes.

### 3.1.6 Player

One of the best known robot software projects is Player [35, 36]. Player is a robot device server that combines a message protocol with a specific device model to give access to the robot’s hardware transparently via network. The communication can happen internally in a single application or over the network, which is the more common use case for Player. The communication protocol of Player 2.0 is based on XDR [37], the External Data Representation Standard. It defines a language based on C structs, enumerations and basic data types to describe messages and types exchanged over the network. The data is transmitted via TCP connections among several Player servers and clients. The device model is based on the Unix philosophy that everything is a file. In terms of Player this means that devices like laser range finders or cameras are treated similar to a file. They can be opened for reading and writing, and data is read and written in form of data and command streams. There is a request/reply mechanism that can be used to configure a device (akin to ioctl()). Similar to programming APIs interfaces specify how data is exchanged with a device. For example a position interface can be used to read positions and write commands where to go. Besides real devices Player can handle abstract devices. These are software components which perform arbitrary calculation, possible enriching data acquired from sensors, deriving new information like extracting the position of the ball from a camera image or driving actuators, for instance a simple navigation driver could plan a path and then send driving commands to the real actuator. A major goal with the communication design was to be able to distribute components over the net-

\[4\text{function to manipulate device parameters to change operating characteristics, cf.}\]
\[http://www.kernel.org/doc/man-pages/online/pages/man2/ioctl.2.html\]
work on different machines and to achieve programming language independence (an API for network programming is the bare minimum needed).

In a typical Player scenario an instance is running on the robot – the server. Embedded as a number of threads are drivers for the robot’s hardware like actuators and sensors. To solve a specific task another software part acts as a Player client. It reads the information from the sensors via the network, decides what to do and orders appropriate actuator actions, again via the network. If multiple robots are in the scenario information can be exchanged among them with Player.

The Player project also drives the development of the Stage and Gazebo simulators. While Stage simulates a population of Player devices in a 2D world Gazebo provides a full 3D simulation with a physics engine.

The Player framework might have been a good candidate for our own endeavor, especially because it is in widespread use and drivers for many hardware components already exist. In Section 4.1 we will argue why Player was not used as the framework.

3.1.7 NaoQi

NaoQi [38] is the robot software framework that comes with the Nao robot by Aldebaran Robotics. NaoQi uses a broker concept where one or more brokers each run a number of modules. Communication between the modules happens via remote method, invocation via the brokers. To do so a module exports a number of methods, other modules can invoke. The modules have well-known names by which a certain feature set is accessed. Proxies are provided for modules for easier execution of methods on other modules. In the default configuration a main broker runs on the robot with a set of base modules required to use the robot like sensor access, image acquisition and actuator control. For the standard modules pre-defined proxies are provided.

Developers can decide to compile their module as a dynamically loaded library which can be run by the main broker. This is how the base modules are integrated. Additionally, a separate process can be created with a child broker that connects to the main broker via a TCP connection. Over this connection, the communication is based on SOAP [31]. SOAP is a protocol for data exchange and remote method invocation. It relies on the Extensible Markup Language (XML) as its message format. In [39] performance issues with SOAP have already been analyzed. Although the amount of data needed on a robot is far away from the domain of high performance computing, our own experiments have shown that the limited computing capacity is saturated quickly with message generation and parsing even for a limited amount of data transferred. Therefore the only possible way of integration with NaoQi is having a

---

module running in the main broker. Using SOAP as a communication channel is virtually impossible due to the resource constraints on the Nao. Additionally no documentation is provided for the SOAP services, method invocation, and broker interaction via SOAP. Therefore a reverse engineering effort would be needed to communicate to NaoQi without any of its libraries. This effort has been done at the University of Bremen, but due to pressure by Aldebaran this code is not released.

NaoQi already provides several modules for sensor and actuator access, motion pattern generation and execution, simple behavior execution, speech synthesis, and image acquisition. The methods and protocols for accessing the hardware by means close to the operating system, like reading and writing from and to a device file, have not been disclosed. Therefore NaoQi currently is the only way of accessing the robot’s hardware, thus making integration with it a mandatory requirement for any other robot framework that needs to control the Nao, cf. Section 4.3.

NaoQi has a very restrictive license. NaoQi not only is closed source software, but its use is restricted to the Nao platform and even to the RoboCup domain alone. This puts uncertainty about the future usability of source code produced for the NaoQi framework. Consequently NaoQi cannot be considered as a general robot software framework.

3.2 Behavior Execution Implementations

Robots need a behavior system of some form that coordinates the actions required to accomplish the envisioned task. This sections gives an overview of some behavior systems that have been described in published papers and articles. A problem with many of the shown approaches is, that the source code to implement the system is not freely available. Thus we reference these for a general overview about the implemented and described approaches.

3.2.1 PRS-lite

PRS-lite [40, 24] is a light-weight reactive controller based loosely on the Procedural Reasoning System [41] that allows for a mixture of goal-directed and reactive activity. Goals and objectives the robot has to accomplish are decomposed into a set of basic behaviours executed by the robot. It is written in Lisp and uses a Lisp-like syntax for the behavior specification.

It combines goal-driven and event-driven aspects of behavior programming. The representational basis of PRS-lite is the activity schema, a parameterized finite state machine (FSM). It is encoded as an ordered list of goal sets. Satisfying these goal sets in the given order will yield the overall objective of the schema. These activities can be enabled, intended in PRS-lite vocabulary.
3.2. Behavior Execution Implementations

Lots of these intentions can run at the same time, allowing for concurrency of the tasks. Special goal types such as waiting for certain conditions permits observing the environment for unexpected deviations and to change the intentions on that event. Overall PRS-lite instantiates a forest of hierarchically structured parallel activities.

3.2.2 Colbert

Colbert [41] is the successor of PRS-lite for the Saphira robot software architecture. Like PRS-lite Colbert uses FSM semantics for the behavior specification and it makes concurrent execution of multiple behaviors possible. The major difference is that PRS-lite used FSMs as the language for specifying the behavior, while Colbert uses a dialect of C with FSM as the underlying semantics. Additionally Colbert added a signalling system for interruption, suspension and resumption of tasks. Finally Colbert used an evaluator to read the behavior scripts and process them. They can be changed at runtime to modify or fix the behavior in case of an error.

3.2.3 URBI

A software system officially supported by Aldebaran Robotics is URBI [42] the Universal Robotic Body Interface. This framework evolves around the URBI language which was developed with low-level actuator control in mind. It features rule-based programming with a syntax close to C++. It supports object-oriented programming including inheritance and polymorphism.

Major features of the language are event-based programming and parallelism at the core of its semantics. For event-based programming URBI provides whenever and at control structures. The former executes a piece of code the whole time a condition evaluates to true, the latter will execute it once when the condition becomes true. Commands can be executed serialized or in parallel. URBI has special operators which cause parallel execution of different code segments. Different strategies to resolve conflicts that occur when a value is set in both threads at the same time are available. In the simplest case commands are enqueued and executed one after another in an unknown order. Other strategies are adding up the conflicting values or averaging them before they are assigned. With the given examples and the feature set URBI concentrates on the low-level aspects of controlling a robot, i.e. coordinating different actuators and execute possibly concurrent movements. The URBI framework features a component framework which allows for the integration of external code written in C++. URBI employs a client/server model. The server at the core there is the URBI engine. It can run on the robot or off-board, for instance on a regular PC. The URBI engine executes the scripts and loads
and integrates the C++ components. Clients can be written in a number of languages like C++ and Java, they connect to the URBI engine as a client and order behavior execution or provide a component externally over the network. Frequently mentioned use cases is the offloading of computational intensive tasks like computer vision for smaller robots.

The source code of later versions of this package is not available and thus one depends on the vendor to supply the adaptation to the robotic system at hand. Therefore it does not meet our requirement of a platform-independent and open behavior system. Additionally it is too much focused on low-level control.

3.2.4 XABSL

One of the behavior specification languages that gained most impact recently in the RoboCup domain is XABSL [43], the Extensible Agent Behavior Specification Language. XABSL mandates a special language to define hierarchies of finite state machines (FSM), called options. Each state of an option is associated with either another option or a basic behavior that the system provides. By associating a state with another option the hierarchy is formed. Following the hierarchy from the top-most option to subsequent options associated with the active states to the leaf state an option activation graph is produced. Options and basic behaviors can be associated with any number of parent states.

The original XABSL language was based on XML. This was later revised with YABSL [44]. It describes a C-like language that allows for behavior definitions with much less overhead and was later made the default language as the release version of the source code suggests. Therefore, when speaking about XABSL, we refer to the C-like syntax. A converter exists to translate between both forms of the language. Both forms of the languages are compiled to intermediate code which is then used by the XABSL engine for execution. Basic behaviors are functions or classes that have to be implemented in C++ and linked to the XABSL engine. These functions or classes provide access to functionality of the software framework to the XABSL engine, for example the walking or driving of the robot.

A plugin has been written for our robot software framework to test the possibilities of integration and to test the XABSL features. It has turned out that especially the need to implement a C++ function for every provided basic behavior and the definition of adapters for every input signal makes it cumbersome to use. Files written in the C-like language have to be translated into intermediate code before they can be used in the framework. This additional step makes the workflow tedious. XABSL would benefit from having support for directly reading the XABSL files, be it the XML or the C-like variant of the language, akin to an interpreter. One of the big advantages of XABSL is its
3.2. Behavior Execution Implementations

being freely available (although the core parts are missing an explicit license) and used by several teams in the RoboCup community.

3.2.5 RoboCup Team Description Papers

We will now summarize information gathered from other RoboCup teams’ description papers that are published by every participating team for each RoboCup to describe the pursued research directions, innovations and achievements. We will concentrate on information regarding the behavior sub-systems.

One of the most notable behavior sub-systems is XABSL, we already described in detail above. It is used by several RoboCup teams like the German Team (Four-Legged League, [44]), Darmstadt Dribblers (Humanoid League, [34]), CoPS Stuttgart (Middle Size League, [43]) and others.

The FUmanoid team from FU Berlin uses a concurrent continuously executed behavior system [45]. Several behaviors run in parallel on different levels. The FUmanoid team uses three levels, a low-level behavior for simple actions like walking and turning, a role behavior that combines the simple behaviors to actions for an attacker or defender, and a strategy layer which decides on the overall strategy, possibly taking into account communication with other robots. This looks similar to the subsumption architecture described in Chapter 2. The CMDash team describes in [46] the necessity of a unified world model. To limit the effects of noisy data, multiple sources are integrated and merged. Based on this world model different game behaviors have been implemented. From our participation in RoboCup we have learned that a unified world model is beneficial [47]. In [48] a qualitative world model was described that allows for formalizing soccer moves.

The purely reactive paradigm has attracted a lot of attention in the past [21, 20]. Elements of this paradigm are still in active use, often in hybrid deliberative-reactive systems. This means that in the reactive part multiple behaviors run concurrently. Action selection or fusion is necessary if multiple behaviors can be active at the same time. As mentioned above in URBI several different strategies can be chosen to solve conflicts if assignments to the same value happen in concurrently executed threads. In [49] Laue and Röfer describe an architecture that uses potential fields\(^6\) for behavior selection. For each behavior activation values are calculated and the most suitable one is executed. The SPQR and ISocRob joint team concentrates on cooperative behaviors [50]. While both parts of the team develop their own underlying system a plan representation and coordination protocol has been developed to support cooperative behavior of the robots. Both teams have used petri nets before

\(^6\)Method used originally in robot navigation that places the robot in a potential field with imaginary attracting forces from the goal and repelling forces from obstacles. The sum at the current point gives the current direction.
for modeling of behaviors and their coordination [51, 52]. This has now been extended to model messages for example to establish or break a commitment.

3.3 Agent Modeling with
UML Statecharts and Hybrid Automata

To ease the development of behaviors and to properly describe it usually a model is applied for formalizing the behavior. We have seen that FSM are used often, for instance in XABSL, PRS-lite and Colbert (cf. Section 3.2). Now we are going to describe one modeling approach in more detail, because it is of particular importance for this thesis (see Chapter 5).

Stolzenburg et al. have used UML statecharts [53] for modeling a robot soccer agent behavior [54]. UML statecharts are based on Harel’s statecharts, a generalization of the concepts of finite state machines [55]. Statecharts are directed graphs that are used to describe the behavior of an object. It explicitly supports building hierarchies and concurrency. In statecharts a state is expressed by a name, and an optional action that is to be executed in that state. In UML entry and exit actions can be specified, which are executed on entering or leaving a state. Transitions are triggered by specified events, and have optional conditions, called guards, which must be fulfilled to execute a transition on the event. For example multiple transitions for the same event can be differentiated by appropriate guards. Opposed to regular state machines in statecharts a transition may occur to multiple states at the same time. For this concurrent states have been added, which has at least two active substates and support synchronization [56]. Stolzenburg et al. used statecharts to model multiagent systems with the specific example of robotic soccer [54]. Each soccer agent is modeled on several levels using the hierarchical aspect of state charts. At the top-level is the so-called mode, which determines the overall program like setup, attack or defend. In the middle is the script level with plan skeletons that use behaviors like kick or dribble defined on the lowest level.

Later Stolzenburg et al. refined the approach by combining the UML statecharts with hybrid automata [57] and modeled the time that synchronisation takes among several agents [58]. A hybrid automaton is a finite state machine which allows for state transitions following differential equations (flow conditions) on the one side, and logical expressions (jump conditions), on the other side. Additionally invariant conditions can be specified. Concurrent machines can be synchronized by events, labels that can be defined for transitions, and if the same label is present on transitions in different hybrid automata the transition will only finish after all automata have passed the appropriately labeled transition. Since to some degree UML statecharts are akin to hybrid automata,
Stolzenburg et al. combined the two approaches [58]. This adds invariant and flow conditions to UML statecharts to model continuous actions like the movement of the robot. Additionally, the time required for the synchronisation is no longer assumed to take zero time, as is done in UML statecharts. This is achieved by introducing a synchronization variable that is shared among several sub-states in a concurrent state and works similar to a semaphore.

As we describe in Chapter 5, we concentrate on a different level of the robot behavior. In terms of the just described papers, we want to model behavior somewhat comparable to the script level mentioned above. Due to this, we will disregard the multiagent and time coordination facilities provided by the introduced approaches using UML state machines for now, and will resort to hybrid automata and extend them for modeling the robot’s behavior in Chapters 5 and 6.
Chapter 4

The Fawkes Robot Software Framework

As mentioned in the previous chapter there is a plethora of robot software frameworks (RSF). There was already an existing RSF called RCSoft5 that was used on our robots. But there were good reasons to believe that a rewrite was necessary to provide all the desired features and to account for the specific constraints of the hardware platform. In Section 4.1 we give details on the decision to start a new framework, and neither use the existing custom framework nor another existing framework. Then in Section 4.2 we describe the framework design and its base components before we present the component configurations for the Nao in Section 4.3 and for the MSL robot in Section 4.4.

4.1 Decision for a new Framework

During RoboCup 2006 it became obvious that the used robot platform based on a differential drive, a directed camera and a laser range finder was not sufficient anymore to be competitive for the Middle Size League. Besides the hardware deficiencies making it virtually impossible to compete with robots employing omni-directional motion and camera systems, severe problems in the software system became evident. For example some components like the laser and localization systems were too slow for the expected game speeds. Also there were fundamental architecture problems like the lack of any synchronization between different components and huge communication delays in the caused by a global mutual exclusion lock acquired for each reading and writing data operation. During a tournament game at RoboCup 2006 the behavior module was accidentally run twice, resulting in a worse performance.

Other frameworks were taken into account for the new system. Middleware-based frameworks like Miro or Orocos were quickly ruled out, because their major feature, the transparent communication over the network was of no
importance and even forbidden for the primary target platform – our soccer robots. Many architectures could not be taken into account, because they were either too old or closed source, making it cumbersome or even impossible to tweak everything to our needs.

So, we ruled out the other possible RSFs, and the remaining candidate to look at was Player. Similarly to the middleware-based frameworks, the major feature of transparent network communication, e.g. for hardware access, was uninteresting for our domain. Likewise the programming language independence was not a major concern, as it was clear that the majority of the code should be written in C++ for efficiency reasons and to be able to re-use existing code. The motor controllers and actuator drivers for the targeted robot platform were not available in Player and required new code anyway. Finally we decided not to go with Player. Only a small fraction of Player would have been used. The drivers and functional components had to be developed from scratch, and with a custom communication infrastructure, trade-offs expected to provide the flexibility we needed, could be circumvented.

Since several of the framework ideas are based on the old framework, knowledge and experience gained over the last couple of years could be transferred to avoid known problems and base on the ones that worked.

The result of the first phase was presented and approved in February 2007 based on a very early prototype of the \textit{Fawkes Robot Software Framework} \cite{fawkes}. The work has been continued since then and Fawkes has evolved into a full-blown robot software framework that is being used on our RoboCup@Home, MSL, and Nao robots, on the latter two as the exclusive software system.

\section{The Fawkes Framework}

Based on the experience and after evaluating existing robot software systems and literature the following structure forms the basis of the Fawkes RSF, as depicted in Figure 4.1, following a \textit{component-based} design (Section 4.2.1). In the default configuration the whole software runs as a \textit{single monolithic process}, subdivided into separate \textit{threads}. A core application initializes and configures base components like the blackboard, a central logging and networking infrastructure. The functional parts of the software like navigation, image processing, behavior execution etc. are provided in extension \textit{plugins}. Simplified, these plugins, provide a \textit{set of threads} (Section 4.2.2) that request needed resources from the core application via \textit{aspects} (Section 4.2.4). If this succeeds, and only if this succeeds, the plugin will run (Section 4.2.5 about \textit{guarantees}). The core application provides a \textit{main loop}, the threads can be synchronized with. The default main loop (Section 4.2.3) is subdivided into certain stages. Roughly, it implements a \textit{sense-think-act loop} with some intermediate levels. All threads communicate with each other via the \textit{blackboard}. It provides data storage for
Figure 4.1: Fawkes Application Structure

data and message passing for commands. Using the central blackboard has several advantages that are described in detail in Section 4.2.6. The base application also provides infrastructure for a centralized configuration (Section 4.2.8) and an extensible network protocol (Section 4.2.9) that is used for remote instruction of plugin load/unload commands, configuration editing and access to the blackboard data. A central clock supports proper timing for real systems and simulation environments (Section 4.2.10). Log output is aggregated and distributed by the framework to support multiple log targets and easy access to the logging information (Section 4.2.11). The FireVision computer vision framework (Section 4.2.7) provides the integration and toolbox for image processing applications. A webview plugin provides access to the internals of the framework via a web interface (Section 4.2.13).

In the beginning of the design phase the decision was made to use C++ for the implementation of the framework. It allows for the required efficiency and is an object-oriented programming language that aids software engineering and structuring. Additionally it lets us re-use code from the former framework that was also written in C++. The RSF has been used and tested on Linux, especially Fedora Linux, and FreeBSD. With the latter, no real robot was used, but the general ability to run on different systems was evaluated. This procedure revealed a few problems and fixing them lead to better and more portable code.
4.2. THE FAWKES FRAMEWORK

4.2.1 Component-based Design

Fawkes follows a component-based approach for defining different functional blocks. The term component has already been defined in Section 2.3.1.

In Fawkes components are logical elements. They manifest in the form of plugins. Generally, a single plugin implements one component. But there are situations, where it is useful to combine multiple components into a single plugin for code efficiency or easier synchronisation. With the blackboard as the communication infrastructure, system components can be defined by a set of input and output interfaces. This allows for easily replacing a component as long as the replacement component requires and provides the same sets of interfaces.

4.2.2 Plugins and Threads

The Fawkes core application only provides the basic infrastructure, the pure framework so to speak. The functionality that makes an arbitrary framework a robot software system is provided by plugins. The plugins are implemented as dynamically loadable libraries – shared objects on Linux systems. They implement a particular interface which gives access to descriptive and dependency information and a set of threads.

Plugins can be loaded and unloaded at runtime. This allows for a fast development cycle. Usually a developer works on one plugin at a time. With the ability to reload only this plugin the program-compile-test cycle can be quicker, because only the changed plugin has to be reloaded, not the whole system.

Threads are one of the key elements of the Fawkes RSF. With the advent of modern multi-core CPUs, even for small and smallest devices as depicted in Figure 4.2, it is considerably worthwhile to provide simple ways to exploit the multi-processing capabilities. With the decision to make every functional entity of plugins a thread, it is reasonably easy to exploit this feature. Threading
is implemented based on the POSIX\(^1\) Threads API [60, 61]. They can operate in two different modes, either in continuous or in wait-for-wakeup mode. In continuous mode the thread runs all the time, until it exits or is torn down by another thread. In wait-for-wakeup mode, the thread blocks, until it is woken up. When woken up, it executes a single iteration.

As already mentioned the most costly operations on the robot are sensor data processing and deliberation. Especially on more elaborated robots where multiple sensors are evaluated running these threads concurrently exploitation of multi-core CPU comes at virtually no cost and development effort. However, to speed up single algorithms the multi-processing needs to occur on a lower level, for example by using OpenMP [62] or similar techniques.

A plugin has a set of one or more threads. In Figure 4.1 Plugin A has 3 threads, Plugin B has only two. When a plugin is loaded the threads are initialized according to its aspects (cf. Section 4.2.4) and start if and only if the initialization was successful for all threads of the plugin (cf. Section 4.2.5).

### 4.2.3 Main Loop

The core application provides a structured main loop that is used for synchronized thread execution. Most robotic tasks are recurring tasks executed in a loop, e.g. the sensor data is extracted every time new data is available and the locomotion component executes and monitors actuator commands continuously following an interleaved pattern of monitoring and execution. Therefore a continuously running main loop can be implemented that allows for interleaved or concurrent execution of all tasks. Another observation made in the old RSF is that data is often produced and extracted without any component that could make use of the data fast enough.

This loop is an option for threads and not a mandatory requirement. A thread can be programmed to deliberately run concurrent to the main loop. Threads that are integrated into the main loop operate in wait-for-wakeup mode (cf. Section 4.2.2) and are woken up at the hook they registered for (via aspects, cf. Section 4.2.4). Concurrent threads usually run as continuous threads with their own timing, often accompanied by a second thread that integrates into the main loop that writes the data to the blackboard at an appropriate state. Sometimes concurrent threads also operate in wait-for-wakeup mode, and then wake themselves upon an event like changed data in the blackboard. For instance a component like the localization based on Monte Carlo particle filtering [63] could run only at a low frequency, while tracking based on odometry can be a lot faster.

Different platforms or domains might have different requirements for inter-thread synchronization, execution order, or the required concurrency. For this

\(^1\)POSIX: http://www.opengroup.org/austin/papers/backgrounder.html
reason the main loop can be replaced by a plugin at runtime. In the following we will describe the standard main loop.

The standard main loop is divided into nine stages shown in Figure 4.3. The stages are called hooks, because plugin threads can be “hooked in” at each stage. The pre-loop and post-loop hooks are mainly used for synchronized simulator integration to exactly match one simulation with one application cycle. In the sensor/vision hook sensor and image data is acquired, which is processed in the sensor processing stage. Afterwards data can be integrated into a world model. In the think stage an agent program would run and make decisions about the game play and strategy. It enqueues commands for the behavior engine, which executes the skills. The behavior engine will monitor the skills and call functions lying deeper in the system to trigger actuator base components like locomotion. The actuator stage has been sub-divided into the act and act execution sub-stages. The act stage is an intermediate level that could for example be used by a locomotion component for path planning, which is then finally compiled to motor commands and executed in the act execution stage. Once a loop is finished it is started again, possibly after a short configured delay. Assume the sensor/vision hook would be active at the moment (hence its orange color). If so sensor data acquisition and vision threads will be woken up and the main loop will wait for these threads to finish. A monitoring mechanism has been implemented that reacts to threads that run for too long by excluding them from the main loop. This way a rampage thread does not influence the whole main loop. This is explained in more detail in Section 4.2.12.

Especially the sensor processing and think stages are candidates for concurrent execution. These stages can be quite time consuming, especially on a constrained system like the Nao. Using concurrent processing for the image processing would allow for example the locomotion component to move the robot based on the last decision, avoiding obstacles with the faster ultrasonic sensors. A long-running planning module could deliberate about the overall game strategy during the think stage, while the skills will run a lot faster in the rest of the system following the last determined strategy.

### 4.2.4 Aspects

Threads need access to features the RSF provides, like the blackboard, configuration or logging subsystems (cf. Section 4.2.11). Likewise there have to
be means by which a plugin can provide the new main loop when it wants to replace the default main loop as described in Section 4.2.3. An easy way one might think of is a central hub class, possibly implemented as a singleton, which provides static methods to get access to certain sub-systems. But this would lead to a lot of code put into each thread just for accessing the sub-systems. Additionally the framework could not know what the requirements of the different threads were, and ensure that they were met.

To overcome these shortcomings we borrowed ideas from aspect-oriented programming [65]. We have added so-called aspects to the software. Each aspect provides access to a particular feature of the framework. In our C++ framework each aspect is represented as simple base class. The simplicity is important to be noted here. The classes may not inherit from another class and they are cut to the bare minimum of functionality. In the simplest form they only have one protected member that is initialized to reference a class providing access to the framework feature, which is set in the initialization phase of a thread. Due to the ability of polymorphic inheritance in C++ a thread simply “gets the aspect” by deriving the aspect class. Metaphorically speaking the aspects are wrapped around a thread like the leaves of a tulip’s flower. Using the runtime type information (RTTI) the framework can determine at runtime what features a particular thread needs (or provides), or detect if requirements cannot be met. We will now describe a few important aspects of the RSF.

**BlockedTimingAspect.** This aspect integrates a thread into the synchronized main loop. The aspect is initialized with a particular hook (cf. Section 4.2.3) when the thread should be woken up and run.

**BlackBoardAspect.** Threads that have this aspect get access to the blackboard to exchange data with other parts of the system and to send messages. The blackboard is described in more detail in Section 4.2.6.

**ConfigurableAspect.** This aspect provides access to the central configuration system, which is described in Section 4.2.8.

**MainLoopAspect.** The MainLoopAspect allows for replacing the main loop of the core application. This aspect differs as it is the first aspect we introduce that actually provides a resource that changes the behavior of the system, opposed to requesting a resource from the system. The aspect is implemented by having a mandatory constructor parameter that takes the new main loop which must be a valid instance. This way compliance can already be checked at compile time. The framework ensures that at any time there is at most one thread that replaces the main loop, to avoid competition and race conditions.

**ClockAspect.** The ClockAspect is used to distribute the central clock to all threads. The TimeSourceAspect is closely related to the ClockAspect. It allows

---

2Pattern that ensures that at most one instance of a particular resource exists, cf. [64]
to register a time source with the framework. In a simulation environment this
can be used to distribute the simulated time, which is likely to divert from the
real time.

**VisionMasterAspect.** A thread with the VisionMasterAspect provides the vi-
sion master. This in turn provides the infrastructure for computer vision ap-
lications. It manages access to the cameras thereby allowing multiple plugins
to use the same hardware. A plugin that is being unloaded does not close the
camera that is still being used by another vision application. It also allows vi-
sion threads to register for a synchronized execution. There can be at most one
thread with the VisionMasterAspect loaded at any one time.

**VisionAspect.** The VisionAspect can only be initialized if a plugin with a
thread that has the VisionMasterAspect has already been loaded. The Vision-
Aspect provides access to the vision master which can supply cameras and is
responsible for scheduling the different image processing threads. The initial-
ization will fail if the vision master has not been loaded yet.

**NetworkingAspect.** Threads with the NetworkingAspect can send and receive
data via the FawkesNetworkProtocol. Additionally they can publish services
and browse available network services (of any type). The Webview plugin,
described in Section 4.2.13, for example announces the web server service this
way. More information about the network abilities is described in Section 4.2.9.

### 4.2.5 (Soft) Guarantees

An often recurring problem is to check if required resources are still available,
or to check for errors while using them. In object-oriented programming lan-
guages like C++ this problem has been alleviated by introducing exceptions.
A block of code is encapsulated and if this code causes an exception it will be
cought and handled accordingly. The soft guarantees are similar in spirit. They
should provide a useful exception handling on the framework level. Threads
should be freed from the need to check their resources and to avoid as much
error handling as possible. The guarantees are soft, meaning that there is no
controlling component that ensures the guarantees at any given time. In fact
the conditions are checked during initialization, and before finalization not to
remove resources which are still required.

The most important guarantee implemented so far is the *initialization guar-
antee*. It guarantees that a plugin is either successfully initialized in full, or is
never run. This transcends to the threads of a plugin. Successful initia-
lization means that all aspects of all threads must be successfully initialized, and
no optional initialization routine of any thread reports an error. If only one
thread cannot be initialized (for example having a VisionAspect and no vision
master has been registered), none of the threads will be started, but they will
immediately be destroyed again and an error will be reported stating why the plug-in could not be loaded.

Another guarantee is the single writer constraint for blackboard interfaces described in Section 4.2.6. At any one time there may be at most a single writer for a certain blackboard interface.

### 4.2.6 Blackboard

All data extracted and produced by the system and marked for sharing is stored in a central blackboard. It contains so-called interfaces, a twofold way of exchanging data of a specified type. Each interface has a well-known identifier that is used to form the connections between components. It is uniquely identified by the combination of the type and identifier, which is commonly written as the two values separated by a double colon (::). An interface for one has a shared data memory segment, and for the other a message passing channel is assigned to the interface. The memory contains a specified group of values of simple types like floating point numbers, integers and strings. This value group provides the core of the information and constitutes the basic data that is shared. An interface has two types of accessors, the writer and the reader – denoting the type of access to the memory area. At any time, there may be at most one writer for an interface, but any number of readers. This is to prevent multiple writers to compete for the same interface overwriting each others data. This would lead to confusion (and has indeed led to confusion in the old software framework where this was allowed) when a reader would sometimes get the data of one writer and sometimes of the other. The message channel is intended as a command channel, allowing for commands to be sent. To stress this, sending messages is only allowed from a reader to the writer. In this regard, the reader/writer relationship is akin to a client/server scheme where multiple clients/readers access a single server/writer. The blackboard is accessible locally within the Fawkes process and remotely via the network (cf. Section 4.2.9).

Interfaces are defined by an XML file. The format is defined by a document type definition (DTD) file. It provides means to layout the group of values in the memory section and defining the possible message types for the interface. Values from the data part can be referenced by the messages for easy definition. The format supports enumerations and constants, single values and arrays. In the current implementation, field sizes have to be specified statically. From the XML file a C++ class for easy access to the data and messages is generated. Additionally it supplies input for the Lua wrapper generator (cf. Chapter 6). Interfaces contain a MD5³ hash that is used to check compatibility of interfaces when the blackboard is accessed over the network. This prevents

³Message Digest Algorithm 5 [66]
errors and confusion when an interface is changed on the robot, but not all developers working on the robot have updated to the latest version, yet.

The blackboard supports observers and listeners to directly react to blackboard events. Both are implemented using the observer pattern [64]. They provide specific virtual methods that are called on the appropriate event. Blackboard observers react to global blackboard events, like a created or destroyed interface. It can be used for example to open new interfaces of a certain group, like object position information about obstacles. The listener reacts to events for a specific event, like opening or closing of the interface for reading or writing by another thread, or a modification of the data.

A conceptual view of the blackboard is depicted in Figure 4.4. In the given scenario the blackboard has been instantiated by the core application. Two threads with the BlackBoardAspect access the blackboard internally, a remote application uses the network facilities for external access to the blackboard. There are two interfaces with different unique identifiers, having different types or IDs. The continuous lines denote access to the memory segment, red lines for writing and orange line for reading access. The curly lines denote remote access, but are otherwise similar to the local access. The dotted line shows a connection that would violate the single writer constraint, it would be rejected by the framework. The dashed lines show possible message directions. As we see with Thread 2 and Remote App there can be multiple senders for an interface, Interface A in the example. The writer Thread 1 receives the messages.

An example is an object position interface, which provides access to position information of an object, like the position of the robot itself or the ball on the field. A message could be the explicit setting of the position, for example to provide a hint of the current location to a localization module by the user.

4.2.7 FireVision

FireVision is a computer vision framework (CVF) which has been transferred almost unchanged from the former RSF. Its development had started later than the rest of the old framework and was designed with some of the new ideas in
mind, like better encapsulation and synchronized execution.

It comprises a simple CVF, targeted to robot applications, that provides utilities for the basic needs of image processing like image acquisition, efficient byte format conversions, color learning, feature extraction and filters. Recently adapters have been added to integrate OpenCV [67] into the CVF.

A base vision plugin called \texttt{fvbase} provides the vision master for the system and handles maintaining physical camera access. Vision applications with the VisionAspect, for example for extracting the ball position from the image, then register with the base vision and get access to the camera via shared memory. Depending on the expected duration of processing a single frame, the plugins can be chosen to run either synchronized with the base vision (they are woken up by the base vision and it will wait for them to finish), or concurrent.

### 4.2.8 Configuration

Although an RSF must be configurable, even multiple instances on platforms of the same type, if only to set the role of the robot on the soccer field or to adapt for slightest hardware differences.

Fawkes provides a configuration subsystem that uses a replaceable storage component for the configuration data. The general interface mandates several standard types like integral or floating point numbers, strings and boolean values that the configuration store must be able to handle. The values are referenced by a Unix path-like identifier. This allows for some structuring and grouping of the configuration values. By default the configuration data is stored in a SQLite\textsuperscript{4} database. The configuration subsystem supports an event system, that can inform threads, if the configuration is changed. In this case a component can immediately adapt to the changed configuration values without need to reload it.

### 4.2.9 Networking

For mobile robots transmission of data via both, wired and wireless networks is essential. In the simplest case it is just used to send instructions to the robot to start a task, in more elaborated scenarios the robots communicate with each other over the network or allow for a distributed computing environment to off-load some of the computational burden.

In the Fawkes RSF we concentrate on remote access for commanding and monitoring the robot, and inter-robot communication to exchange information about the world. Distributed computing is currently not a major feature, as in the MSL and SPL off-board computing is even disallowed. However, as described in Section 4.2.6 it is still possible with a small effort.

\footnote{SQLite: \url{http://www.sqlite.org}}
Fawkes provides two basic networking protocols. The first uses a TCP connection for transmission. It provides a generic infrastructure for encapsulating arbitrary payload sent over the stream. The payload is wrapped into messages of a certain size and type. Sub-protocols implemented on top of this is plugin enquiry and loading and remote access to the blackboard, which is the dominant network application at the moment. The second protocol, called world info protocol, is based on UDP and multicast transmission [70]. It is used to send soccer specific information about a robot’s perception to team members during the game. The communication uses symmetric encryption via AES and carries basic information about the robot’s belief of its own, ball and opponent positions. Additionally the protocol is used to communicate game state information, as received from the central referee box program.

An annoying problem for accessing network services is usually supplying appropriate information like host name or even the numeric address and the port number. To avoid this DNS-based Service Discovery [72] over Multicast-DNS [73] implemented via Avahi has been added. For this system information is exchanged via the Domain Name System (DNS [74, 75]) formatted in specified SRV (service information [76]) and TXT (text information) records, which are usually sent to the link-local network via multicast. With this feature graphical tools can just show a list of descriptive names of robots with a running Fawkes instance to the user to select from.

4.2.10 Central Clock

The Fawkes RSF aims to be applicable on the real robot as well as in a simulation environment. Especially for physically correct, more complex, or multi-robot simulations it might run slower than the real time. Often, simulator simulate discrete steps. Consider a simulation that runs in steps of 20 ms and calls the stakeholders for their decision what to do once every simulation cycle. The simulation however might take longer than 20 ms, however, the world has only advanced by 20 ms. To cope with this it is essential that components like prediction models or timed behaviors make use of the simulated time, and not of the actual system time. However, some code fragments should always use the system time, for example network timeouts happen in system time, even in a simulation environment. Hence a central clock is provided by the base system to component threads via the ClockAspect. The clock gives access to the time (which is real or simulated), and explicitly to the system time. Via the TimeSourceAspect a component thread can supply a new source of time

---

5Transmission Control Protocol [68]
6User Datagram Protocol [69]
7Advanced Encryption Standard [71]
8Avahi: http://www.avahi.org
for the clock. This is then used by simulation components and adaptors to provide the simulated time.

### 4.2.11 Logging

One of the drawbacks with the old RSF was that each process logged separately, usually to the standard output, the console. This hindered an effective analysis after a program failure or often resulted in processes being run over a remote console connection to see the log messages. This problem has been overcome by adding a centralized logging infrastructure to the Fawkes RSF. Via the logging aspect component threads can get access to an instance of a logger. Internally the logger is a dispatcher, that distributes the log messages to a number of loggers. This can be a logging to the standard output on the console, to a file, or to listeners connected over the network. Via the LoggerAspect a component thread can even provide new custom loggers to the system. This is done for instance by the webview plugin to cache the latest log messages to present them to a user of the web interface (cf. Section 4.2.13).

### 4.2.12 Main Loop Timing Constraints

The standard main loop has been described in Section 4.2.3. Such a strong synchronization of the different threads poses a serious risk: a single thread which runs for too long (for example encountering a situation unforeseen by the developer) blocks the whole loop, even worse a deadlock could cause the whole main loop to get stuck. During this thesis we have added precautions to the software that are able to detect and mitigate this problem. Therefore, the POSIX barrier\(^9\) has been re-implemented using mutexes and wait conditions, to allow for barriers with timeouts (a feature not available for POSIX barriers). So a configurable maximum run-time per hook has been added. If the time exceeds before all threads have passed the barrier, at least one thread has run for too long. The system is now able to detect this situation and recognize the threads that caused the problem. The loop can continue and the bad threads are not woken up anymore. Periodically bad threads are checked if they have recovered, if so they are woken up again during the loop. Additionally, the total runtime of the loop is monitored. Sometimes all threads stay inside the time, and yet the total loop time is exceeded. This happens because there is no limit for each particular hook, but a global timeout value valid for every hook. But even slight time drifts might have undesired effects. There is nothing that the system can do to avoid this problem on its own, but printing the information and timing information in this case is valuable for the developer.

---

\(^9\)A barrier is used for synchronization of a group for threads. Each thread must stop at the barrier and cannot proceed until all threads have reached the barrier.
4.2.13 Webview Plugin

Although the Webview plugin is a simple plugin generally not more important than others it is noteworthy because it provides access to the internals of Fawkes via a simple web interface. It was developed by us during this thesis to have an easy way to glance over the values stored in the blackboard. An integrated web server reacts to web browser requests and produces pages on the fly with specific information. This could be providing recent log entries or allow for loading and unloading of plugins. It also supports blackboard introspection, that can be used to view the content of arbitrary blackboard interfaces. The plan is to elaborate this plugin into an (optional) infrastructure component. Via a special aspect arbitrary plugins could extend the web interface. The behavior engine might for example provide an interface to execute, monitor, or stop skills.

4.3 Fawkes on the Nao

The Fawkes RSF originally had been developed with multi-core CPUs and omni-directional locomotion and image processing in mind. Now, on the constrained Nao platforms former design decisions had to be reconsidered and the suitability of the RSF for the Nao had to be verified. The initial branch for the Nao was created in February 2008. In absence of real robots the first component developed was an integration plugin for the simulation environment (cf. Section 6.7.3) provided by Aldebaran. It was not before June 2008 that the physical robots arrived and could be used.

Nao Component Configuration

Figure 4.5 shows the component configuration that has been envisioned for Fawkes and the Nao platform. The graphical representation leans on UML [53]. Boxes with rounded corners denote resources, boxes with angular corners and an icon in the upper right corner denote components. Connections with arrows mark flow of data; connections with a circle at the end denote a provided interface of the component, lines connecting a provided interface with an open arc describe a usage dependency. Components with orange borders have received major contributions by the author during this thesis. Those with dashed orange borders have been developed by the author earlier and received only minor updates during the thesis.

The diagram is separated into four columns. From left to right the abstraction level towards the hardware rises. On the left-hand side there are components for low-level hardware access, followed to the right by sensor extraction and actuator driving components. The world model provides the refined
sensor information of the low-level to the high-level components by agglomerating and merging the different data sources. Finally the agent and behavior engine on the right-hand side control the behavior of the robot. The boxes filled orange constitute the behavior generating components and therefore are of special interest for this thesis. The flags mark components that had been planned to be developed by our partners from the ZaDeAt project from their respective country. We will now describe the different parts of the configuration. The behavior components are described in more detail in Chapters 5 and 6 and thus are mentioned here only briefly. The components depicted in the figure are implemented as a plugin each if not stated otherwise in the component description, therefore often leading to the terms component and plugin being used interchangeably.

**naomana/naosim.** On the left-hand side the naomana/naosim component provides access to the robot hardware, real or simulated. The component can be provided by two different plugin implementations. On the real robot the naomana plugin integrates Fawkes with NaoQi. It supports basic access to the sensors and actuators (described in Section 2.1.3) on the lowest level, like writing the raw IMU data into the blackboard and order the execution of joint angle-setting commands. Additionally it allows for accessing advanced NaoQi features like the motion engine and speech synthesis. Currently the NaoQi motion engine is used to provide basic robot movements like walking, standing up, and kicking the ball (see note on naomotion below). Since NaoQi is the only way to access the robot’s hardware the component is crucial. The basis

**Figure 4.5:** Fawkes Component Configuration for the Nao
was developed early during the preparation of this thesis when the robots arrived. During the thesis major improvements have been applied that made the plugin faster and avoided some problems with the NaoQi integration. Additionally new movements have been made available like standing up when the robot is lying flat on the ground or kicking the ball. The movements are based on examples provided by Aldebaran and are implemented by using the low-level part of the NaoQi motion engine (synchronized joint movement).

The *naosim* plugin integrates the RSF with the Webots simulator. It implements a controller for Webots and communicates via the Fawkes network protocol to the naosim plugin running in Fawkes. It allows for strong synchronization of the simulator and Fawkes main loops. It is explained in more detail in Section 6.7.3 and a screenshot is shown in Figure 6.5.

**naomotion.** The *naomotion* plugin integrates two sub-components, navigation and movement execution. In the navigation sub-component commands like "goto position \((x, y, \theta)\) relative to your current position" are processed, where \((x, y)\) is a cartesian or polar coordinate in a system with the robot at its origin and \(\theta\) being the orientation the robot should have at the destination. This command is decomposed into simple movements possibly by path-planning and integrating obstacle avoidance, for example based on the ultrasonic sound sensors. Simple movements are for instance walking straight for a specified distance, turning, walking on an arc, kicking and standing up. Currently, only a very simple navigation method without obstacle avoidance has been implemented – turn towards the destination, walk as far as necessary to reach the destination and finally rotate to the desired target orientation. The monitoring is minimal, as reliable information via odometry or a closed-loop motion is unavailable. The movement execution itself is implemented as a replaceable sub-component. Currently, two solutions are available. One takes pre-generated patterns and executes them blindly – in an open loop without sensor feedback. The second forwards the simple motions to the naomana/-naosim component for execution via the NaoQi motion engine.

**odometry.** The *odometry* plugin produces information about the relative movement of the robot’s center of mass over the last few loop iterations. This information is especially valuable for refining the robot’s belief of its position by tracking over time or even necessary to update motion models. The data can also be integrated into the locomotion component to adjust the walking parameters based on the observed actual result of the former actuator commands. Originally the odometry component was envisioned as a sub-component of the naomotion plugin, to be implemented by our South African partners. Due to the lack of personnel this has been reassigned to our Austrian partners and is currently work in progress in a master’s thesis.

**fvbase.** The base vision was taken unchanged from the existing software system. It provides access to the cameras of the robot and manages the proper
wakeup of vision threads. Therefore it implements the VisionMasterAspect (cf. Section 4.2.4 and Section 4.2.7).

**naolocalize.** The *naolocalize* plugin obtains information about the position of the robot on the soccer field from camera images. It uses Cox’ algorithm [77, 78] which compares sensor information with a simple map of the environment consisting only of line segments. Given an initial position estimate and the set of recognized field lines, it matches the lines and the map to maximize the correlation, by minimizing the error when assigning recognized lines to their closest map feature. After several iterations the error converges towards a minimum and the position is updated to the new position. Odometry information can be used as input to update the estimates between cycles, which should lead to a noticeable speedup. The lines are detected by using a gradient approach, looking for steep ascents of the pixel intensity on a gray scale image. These steep ascents happen when a white line crosses the green field. The plugin is currently work in progress by our project partners of TU Graz.

**naoball.** The *naoball* plugin is an adapted version of the vision system of the old MSL robots, which had a directed camera on a pan/tilt unit, somewhat comparable to the cameras in the Nao’s head. The picture is scanned on a sparse grid for the orange ball color previously learned with a Bayesian approach. Matching pixels are clustered and a rectangular region of interest (ROI) that encompasses all pixels of a cluster. The largest cluster is assumed to contain the ball (closest object is the biggest in the image, given the same general object size). In the original software, a verification step based on randomized circle detection [79] was executed to match the circle around the ball. This for one provided accurate data of the ball size in the frame for the relative position model, and for the other produced good results even for partially occluded balls. However, this step is computationally too costly for the Nao. As a workaround the ROI is assumed to fit exactly around the ball such that the minimum of the width and height of the ROI is considered as the ball’s diameter. The results are not as good as desired but good enough to work with. A more sophisticated approach will have to be implemented in the future.

**worldmodel.** The *worldmodel* plugin ties together several sources of information and provides a unified model of the world to the higher level components. The different local data sources are the sensor extraction plugins. In the simplest case they are just copied through, possibly filtering the data over time if desired. The world model also receives information sent by other robots over the network. This data can then be integrated to refine the world model [80]. The own belief is sent periodically with about 10 Hz to the other robots via the world info protocol. On the Nao the main function of the world model is the network transmission and reception. But in a later step for example visually detected obstacles could be verified or refined by using the ultrasonic sensor data. For the MSL robot obstacle positions retrieved from the omni-directional
Fawkes on the AllemaniACs MSL Robot

and stereo cameras could be merged, as is done for example for the MSL localization.

The mapping of input to output data and the methods to be used for merging, fusing, or copying the data can be setup in the configuration and does not need any changes to the code, as long as no new method is to be implemented. This is important to note because that allows to reuse the world model with only a changed configuration on both platforms, the Nao and the MSL robot.

**skiller.** The *skiller* component – the skill execution runtime – is the behavior engine part that we will describe in detail in Chapter 6. It is used for executing so-called skills, small reactive behavior entities for specific tasks.

**agent.** The *agent* component is the highest level of robot control. It is the deliberative component in the classical hybrid paradigm (cf. Section 2.4.2). It can be implemented in various ways. Currently, for the Nao this component is implemented by the *luaagent* plugin. It employs techniques similar to the behavior engine’s and is described in more detail in Section 6.8.

We have discussed the component configuration of the Fawkes RSF on the Nao platform. We are now going to discuss the configuration for the AllemaniACs MSL robot, in particular the differences compared to the Nao platform.

## 4.4 Fawkes on the AllemaniACs MSL Robot

Fawkes was originally developed for the MSL platform. Therefore only a small amount of work had to be conducted to adapt the software system to the robot. The basic modules for accessing the hardware already existed prior to starting the thesis. However, there was no behavior system in place and several adjustments and fixes have been applied during the thesis.

### MSL Component Configuration

The component configuration for the MSL robot is depicted in Figure 4.6. The meaning of the different graphical elements is the same as the one mentioned for the Nao component configuration in Section 4.3. Again we have four columns, with an increasing abstraction from left to right and with longer connections and more intermediary components towards the underlying hardware. Compared to the Nao diagram, there are two facts outstanding. First the locomotion component is closer to the hardware and second the part of the configuration for the worldmodel, skiller, and agent components is unchanged, which corroborates the conceptual fitness of the RSF design for a general behavior system for multiple platforms and domains. We will now describe the components specific to the MSL robot. For the *fvbase, worldmodel, skiller,* and *agent* see Section 4.3, as they are the very same components.
kicker. The kicker plugin controls the pneumatic kicking device as described in Section 2.1.4. It allows for executing kicks on any combination of the three kickers, where the middle kick can be controlled in strength by regulating the air pressure applied to the cylinder. Secondary the ball guidance arms can be extended as necessary. It is a structurally simple plugin, which can be regarded as a hardware driver plugin.

navigator. Especially in the very speedy MSL environment the navigator plugin is very important for good game play. The MSL robot plugin combines local and global approaches for collision avoidance and path planning. It employs a landmark-based representation of the environment that contains static landmarks as well as dynamic obstacles detected via sensors. Based on a Delauney tesselation over all obstacles a traversal graph is constructed. This traversal graph is a topological representation of all paths through the perceived obstacles to the target position. On this space A* search [81] is applied to find a short and safe path. The safety of a path is accounted for by incorporating the distances between obstacles passed along the path into the A* cost function [82].

The motor sub-component for hardware access has been directly integrated into the plugin. This makes synchronization easy and fast. For the communication of the path-planning navigation sub-component and the motor-component an internal blackboard interface is used. This allows for sending motor commands from the outside, for example for manual input with a joystick, and peeking into the communication to analyse problems. This also allows to easily split-off the motor plugin, to support several different implementations when the need arises.
4.4. Fawkes on the AllemaniACs MSL Robot

**omni-loc.** The *omni-loc* component is currently work in progress as another diploma thesis [83]. As described in Section 2.2.1 the fields are now truly symmetrical after the last color features that would help to identify the sides of the field have been removed and the localization problem needs to take that into account. The approach pursued is based on Monte Carlo Localization [84]. The field lines provide the necessary static features for the particle filter and odometry information supplied by the navigator is used for the motion model for updating the particle positions in each cycle. A specific problem tackled is the symmetry of the field. The idea is to communicate additional observations by other robots over the network. These are the position estimates of the robots themselves, the ball, teammates, and opponents. This additional information is integrated into the Monte Carlo sample weighting process. Since the goal keeper is relatively fixed on the own team’s half of the field and does usually only move over small distances it is feasible to assume that there is always at least one robot that has good information about the field sides. With these additions the symmetry should be solved without adding another sensor like an electronic compass.

**omni-ball.** Ball detection is particularly easy with an omni-directional camera. The *omni-ball* plugin detects the ball by looking for an orange color blob. Given an a priori learned mapping from image to real world coordinates the position is only a lookup of the pixel coordinates in a map. The basic application had already been developed in the old RSF for a primitive omni-directional camera using a webcam pointed towards a silver-coated light bulb. It has been adapted and improved for new RSF.

**omni-field.** The *omni-field* plugin is used to detect obstacles on and borders around the field. This is done by looking for green to non-green color changes on sparse star-formed scanlines originating at the camera’s center in the image. The obstacles are later clustered and post-processed. The field borders are represented by obstacles, currently no explicit line-detection or reconstruction takes place. The data is used for collisions avoidance and localization.

**stereo-obs.** The *stereo-obs* component uses the stereo camera to detect obstacles and the ball in front of the robot with high precision. Currently, the project is in a prototype stage and no fully-working plugin has been implemented.

We have outlined the component configuration used on the target platforms, the humanoid robot Nao and the MSL robot featuring omni-directional moving and imaging capabilities. As we have seen the basic structure of the configurations is similar, with the major changes regarding the sensors and actuators, as can be expected on two different platforms. This concludes the discussion of the RSF and we will now continue with the behavior system.
Chapter 5

Theoretical Foundations for Skills

In the last chapter we have discussed the underlying Fawkes robot software framework in detail. It is the foundation for implementing robotics applications, but by itself does not offer a convenient way for describing the behavior other than writing a C++ program that directly accesses the sensor and actuator components via the blackboard.

In this chapter we will lay the theoretical foundations for the implementation of a behavior engine, by first explaining the overall architecture in Section 5.1, defining different levels of behaviors in Section 5.2 and modeling skills as reactive behavior entities by describing and extending hybrid automata (cf. Section 3.3) in Section 5.3 including an example definition of a skill. Later in Chapter 6 we will discuss the implementation and usage of this behavior modeling approach.

5.1 Hybrid Behavior Architecture

As described in Section 2.4.1 several execution paradigms have emerged over time. We apply the hybrid deliberative-reactive paradigm introduced in Section 2.4.2. This paradigm already structures the overall behavior execution into the three parts sense, plan, and act and describes the relation between perception, cognition/deliberation and acting. We now want to concentrate on the interaction of cognition/deliberation and acting. The sensed information and inferred artifacts are available to both stages, possibly employing a central world model. We combine aspects of the managerial, state-hierarchy and model-oriented architectures, described in Section 2.4.2, to define our architecture, and define in these terms different levels of behavior.

From the managerial architecture we take the strict ordering. We will form several levels, that provide their services to the higher levels. Orders will only flow from top to bottom. From the state-hierarchy architecture we take the layer classification by the amount and type of state information present in a
layer, as also described in [20]. Finally, with the intent to design a robot that can use decision-theoretic planning based on ReadyLog [85], the model-oriented architecture comes in naturally. We encourage (but not mandate) the use of a central world model for higher behavior levels, for a concise and coherent view on the situation, as we have already described for the Nao and MSL robot component configurations.

5.2 Low-level Control, Behaviors and Agents

We will now define the behavior of the robot in terms of three levels, based on the described hybrid behavior architecture. The levels draw on the level of hardware abstraction, the expected timing criteria, and take the amount of the required state information into account that is described in [20]. After the definition we will describe the influence of this definition on the software architecture.

Definition 1 (Behavior Levels).
Every behavior entity executed by the robot is categorized into one of the following three classes.

Level 0: Low-level Control. Modules on this level run in real-time or close-to-real-time control loops for tasks like motion pattern generation or path-planning and driving. Sensor data acquisition and interpretation usually happen at this level. They have direct or almost-direct access to the hardware and contain no or almost no state information.

Level 1: Skills. Skills are reactive basic behaviors used by the agent as primitive actions. They never access the hardware directly, but only by means of components at Level 0. They need to provide quick execution to be able to operate at soft real-time (about 30 Hz) consuming only a moderate amount of processing power. State information is stored about the immediate past.

Level 2: Agent. Top-most deliberative layer, where global decisions about the robots behavior and the strategic direction are made. There are no strict timing constraints for the execution. Usually state information is generated about the future (planning), but may also be kept about the past.

At each level, execution of behaviors can only be ordered by behaviors which are on the same or a higher level.

Behavior Levels and the Soccer Software Architecture. Modeling the behaviors in certain levels influences the overall software architecture that was
implemented for this thesis. In Figure 5.1 the different layers are depicted as they form the basis for the Fawkes software stack in the robot soccer domains.

At the bottom there is Level 0, containing components for sensor and actuator access and processing with tight control loops which have to run under real-time or close-to-real-time constraints. Notably, this level also contains pure information processing modules, like the world model. Located in the middle is Level 1, containing the behavior engine, the reactive execution and monitoring environment for skills. On the top is Level 2, comprising an agent program, intended as a deliberative component, for example a planner that makes decisions on the overall game play or on the team strategy, possibly communicating and coordinating actions with other robots. In more constrained or simpler scenarios this might well be another reactive component, ordering the execution of skill.

Between Level 0 and Level 1, the lower orange bar, is an abstraction of the low-level components towards the behavior engine. They are encapsulated via the blackboard interfaces, described in Section 4.2.6. This allows for replacing components on this level easily, with no or only small changes on the behavior engine level. Another abstraction layer is between Level 1 and Level 2, the skill set indicated by the upper orange bar. This set of skills depends on the domain and platform at hand. For different platforms the set can easily differ, for instance a robot with an arm can have a “grasp” skill, which a robot without an arm cannot provide. There are two fundamental directions. Services are provided from bottom to top, and orders flow from top to bottom. A module on a certain level cannot instruct higher levels, but it provides its services. Similarly a module does not offer it services to a lower level, but instead passes orders down for execution and expects a response if the execution succeeded or failed.

Against the background of a deliberative approach, where decision theoretic planning should be applied for game play decisions and strategy, one has specific expectations what the behavior engine has to provide – which nevertheless can be applied for reactive decision making. On the higher level strate-
5.3 Modeling Skills with Hybrid State Machines

As mentioned in Section 3.3, Stolzenburg et al. have used UML statecharts to describe the behavior of an agent, especially in the domain of multi-agent systems. Later this model has been extended by hybrid automata. Since we are primarily interested in modeling the behavior of a single robot, we use a simpler model. Skills need a particular programming and run-time environment – the behavior engine. According to Definition 1 the behavior engine is located at Level 1 in our behavior hierarchy. From the initial proposition that skills are reactive execution entities which accomplish simple tasks, i.e. the primitive actions from a higher level perspective, state machines are an obvious and popular choice for modeling the behavior as can be seen by the vast amount of approaches using them [2, 40, 41, 43]. As we need continuous transition conditions between states, we have selected hybrid automata – hybrid state machines (HSMs) to accent the state machine aspect – as the model of our choice. A HSM is a finite state machine which allows for state transitions following differential equations (flow conditions) on the one side, and logical expressions (jump conditions), on the other side.

We will first give the unmodified definition of hybrid state machines, and then augment and extend this definition to be suitable for efficient skill modeling.
5.3.1 Hybrid State Machines

We will now introduce the original definition of hybrid automata [57].

**Definition 2 (Hybrid State Machines – HSM).**

A hybrid state machine is defined as

\[ \mathcal{H} = (G, X, \Sigma, \text{jump}, \text{event}, \text{init}, \text{inv}, \text{flow}) \]

**Control graph.** A finite directed multigraph \( G = (Q, E) \). The vertices \( Q \) are called control modes. The edges in \( E \) are called control switches.

**Variables.** A finite set \( X = \{x_1, \ldots, x_n\} \) of real-numbered variables. The number \( n \) is called dimension of \( \mathcal{H} \). We write \( \dot{X} \) for the set \( \dot{X} = \{\dot{x}_1, \ldots, \dot{x}_n\} \) of dotted variables (which represent first derivatives during continuous change), and we write \( X' \) for the set \( X' = \{x'_1, \ldots, x'_n\} \) of primed variables (which represent value at the conclusion of discrete change).

**Initial, invariant, and flow conditions.** Three vertex labelling functions init, inv, and flow that assign to each local control mode \( q \in Q \) three predicates. Each initial condition init\((q)\) is a predicate whose free variables are from \( X \). Each invariant condition inv\((q)\) is a predicate whose free variables are from \( X \). Each flow condition flow\((q)\) is a predicate whose free variables are from \( X \cup \dot{X} \).

**Jump conditions.** An edge labelling function jump that assigns to each control switch \( e \in E \) a predicate. Each jump condition jump\((e)\) is a predicate whose free variables are from \( X \cup X' \).

**Events.** A finite set \( \Sigma \) of events, and an edge labelling function event : \( E \rightarrow \Sigma \) that assigns to each control switch an event.

A standard example of a thermostat HSM is shown in Figure 5.2 [57]. It models a thermostat of a heater. The variable \( x \) represents the temperature. In the state Off the heater is off and the temperature falls according to the flow condition \( \dot{x} = -0.1x \) (decrease by 10% in each step). In the state On the heater is on and the temperature rises according to the flow condition \( \dot{x} = 5 - 0.1x \).
In the initial situation the heater is off and the temperature is at 20° C. Jump conditions are represented by inequalities on the edges. The jump condition \( x < 19 \) causes the heater to go on as soon as the temperature falls below 19° C. The invariant condition \( x \geq 18 \) asserts that the heater will go on at the latest at a temperature of 18° C.

### 5.3.2 Set of Skills for a Particular Skill Space

The behavior engine should be applicable to multiple platforms and domains. We expect that the available skills will depend on the combination of a particular platform and domain. Therefore we give the following definitions. The domain describes the area of operation and determines the tasks to be executed. The platform describes the used robot system as described in Section 2.1.2.

**Definition 3 (Skill Space).**
The combination of a platform \( P \) and a domain \( D \) with regard to skills is called skill space \( (P, D) \) for platform \( P \) and domain \( D \).

**Definition 4 (Set of Skills).**
The set \( K(P, D) \) is called the set of skills for the skill space \( (P, D) \).

### 5.3.3 Skill Hybrid State Machines

We want to use HSMs to model skills as execution entities. More complex skills often use simpler skills. Therefore we want to provide an efficient way for reusing a skill, which avoids the construction of a state machine that includes both, the complex skill behavior and all of the internal details of the included simple skill. To achieve this we extend the HSMs as follows.

**Definition 5 (Skill Hybrid State Machine – SHSM).**

\[
S = (G, X, D, A, jump, flow, exec, K(P, D))
\]

**Final and failure state.** The control graph of the state machine has only two valid exit states \( Q_{exit} = \{q_{final}, q_{failure}\} \).

**Control graph.** A finite directed multi-graph \( G = (Q, T) \), where \( Q = Q_U \cup Q_{exit} \) are the states (vertices) with \( Q_U \) being the user defined states and \( T \) are the transitions (edges).

**Dependencies.** For hierarchical definition of a skill existing skills can be re-used. These used skills are called dependencies of skill \( S \). Skills that are used in the current skill are called direct dependencies, skills that are used in direct dependencies or their dependencies are called indirect dependencies. A skill may not depend directly or indirectly on itself. For
this we define a set $D \subseteq K(P,D) \setminus S$ of dependencies. Let $D_S \subseteq D$ be the set of skills that the skill $S$ directly depends on. Then the function $\delta : K(P,D) \to \wp(K(P,D) \setminus S)$ with $\delta(S) = D_S \cup \{\delta(d) \mid d \in D_S\}$ gives a set of all direct and indirect dependencies of $S$ and $S \notin \delta(S)$. This can be represented as a dependency graph.

**Execution Function.** A skill is executed with the `exec` function. It assigns values to some variables $x \in X$ and runs the state machine by evaluation of the jump conditions of the current state, possibly leading to a state change. It is defined as $\text{exec}(x_1, \ldots, x_n) \to \{\text{final}, \text{running}, \text{failure}\}$ with $x_i \in X$. The return value depends on the current state after the evaluation.

**Actions.** For the execution of lower-level behaviors and other SHSMs we define a set $A$ of actions. An action $a \in A'$ is a function $a(x_1, \ldots, x_n) \to \{\text{running}, \text{final}, \text{failure}\}$ with $x_i \in X$ that executes a lower-level system behavior (on a lower behavior level). The set $K = \{\text{exec}_d \mid d \in D\}$ is the set of execution functions of dependency skills (on the same behavior level). The set of actions is then defined as $A = A' \cup K$.

**Action Execution.** For each state $q \in Q$ we define a set $E_q \subseteq A$ of actions. Each action is executed when the state is evaluated. The set $E_q$ may be empty.

With the extension of the HSMs skills can be defined more compact. First, we define two possible exit states of the state machine to indicate success or failure. The graph is composed by a number nodes, consisting of the exit states, user-defined nodes, and transitions. Dependencies are defined to ensure that a skill does not call itself and to define the available set of actions. The jump conditions are boolean functions. Conditions are evaluated for the current state only. A skill provides an execution function `exec` which takes a number of variables $x \in X$ which are assigned to the variables in $X$ and then evaluates the graph by checking the jump conditions of the current state. The transition for which the jump condition evaluates to true first is followed. It is expected that the `exec` function is defined in a way that it can be called in multiple iterations, returning `running` in the beginning and then at some point changing its return value to `final` or `failure` when the skill has succeeded or failed. This makes employing a skill in another skill easy, without integrating the whole state machine. It is also used to call a skill from the agent. We expect part of the variables to be modified by exogenous events, for example with new sensing values. Because variables may be modified by exogenous events, the return value of jump conditions can change between two evaluations. Via the $E$ sets actions can be associated with states. They are executed as long as the state remains active. Flow and invariant conditions, and multi-graph synchronization via events have not been used during this thesis. When lifting this concept to Level 2, the agent level, the basic actions are the skills provided by the behavior engine.
5.3. MODELLING SKILLS WITH HYBRID STATE MACHINES

5.3.4 Example Specification for the Nao search_ball Skill

We will now give the formal description of the Nao search_ball skill, which is described in more detail in Section 6.6.1. It is a skill to search for a ball by moving the Nao’s head and turn around its vertical axis. We specify the skill as

$$S_{\text{search_ball}} = (G, X, D, A, \text{jump}, \text{flow}, \text{exec}, K(P, D))$$

We refer to the graph of the graph presented in Figure 6.3 on page 86. We will give the states with the notation $$q_i = \text{NAME}$$ where the name is the one used in the implementation. However, we will use the numerical notation here for a shorter specification of the graph. The skill space is formed by the Nao platform and the robot soccer domain. The flow function is not used.

$$Q = Q_U = \{q_1 = \text{LOOK\_DOWN\_RIGHT}, q_1 = \text{LOOK\_LEFT}, q_3 = \text{LOOK\_UP\_1}, q_4 = \text{LOOK\_RIGHT}, q_5 = \text{CHECK\_TURN}, q_6 = \text{LOOK\_DOWN\_MIDDLE}, q_7 = \text{TURN\_RIGHT\_1}, q_8 = \text{LOOK\_UP\_2}, q_9 = \text{TURN\_RIGHT\_2}, q_{10} = \text{STOP}\} \cup Q_{\text{exit}}$$

$$t_1 = \{(q_i, q_{10}) | i \in \{1, \ldots, 9\}\}$$ (transitions if ball visible)

$$t_2 = \{(q_j, q_{\text{fail}}) | j \in \{1, \ldots, 4, 6, \ldots, 10\}\}$$ (transitions if skill failed)

$$t_3 = \{(q_a, q_b) | a, b \in \{1, \ldots, 10\}, a = b - 1\} \cup \{(q_{10}, q_{\text{final}})\}$$ (skill final)

$$t_4 = \{(q_5, q_{11})\}$$ (transition if turn disabled)

$$t_5 = \{(q_5, q_6)\}$$ (transition if turn enabled)

$$t_6 = \{(q_9, q_{11})\}$$ (transition to repeat)

$$T = t_1 \cup t_2 \cup t_3 \cup t_4 \cup t_5 \cup t_6$$

$$X = \{x_{\text{ball\_visible}}, x_{\text{skill}}, x_{\text{do\_turn}}\}$$

$$D = \{S_{\text{servo}}, S_{\text{turn}}, S_{\text{stop}}\}$$

$$A = \{\text{exec\_servo}, \text{exec\_turn}, \text{exec\_stop}\}$$

$$E_{q_i}^{E_{q_i, j \in \{1, \ldots, 4, 6, 8\}} = \{\text{exec\_servo}\}}, E_{q_j, j \in \{7, 9\}} = \{\text{exec\_turn}\}, E_{q_{10}} = \{\text{exec\_stop}\}$$

$$\text{jump}(t) = \begin{cases} x_{\text{ball\_visible}} = 1 & t \in t_1 \\ x_{\text{skill}} = \text{failure} & t \in t_2 \\ x_{\text{do\_turn}} = 1 & t \in t_4 \\ x_{\text{do\_turn}} = 0 & t \in t_5 \end{cases}$$

$$\text{exec}(x_{\text{do\_turn}}) = \begin{cases} \text{final} & q_{\text{current}} = q_{\text{final}} \\ \text{failure} & q_{\text{current}} = q_{\text{fail}} \\ \text{running} & \text{otherwise} \end{cases}$$

The exec function is assumed to also assign $$x_{\text{do\_turn}}$$ and evaluate the state machine and $$q_{\text{current}}$$ is the current state after the evaluation.
Chapter 6

Implementation of a Lua-based Behavior Engine

In the last chapter we have described the theoretical foundation for proper modeling skills as reactive execution entities with extended hybrid state machines. The base system provides basic actions and skills serve as basic actions to the agent level. In this chapter we describe the specific implementation of a behavior engine we have created, based on the Fawkes robot software framework that has been described in Chapter 4.

First we detail the design decisions made for the implementation in Section 6.1. In Section 6.2 we introduce the Lua scripting language used for the implementation before discussing the actual implementation in Section 6.4. We conclude with a description of the implementation of a simple reactive agent in Section 6.8 that employs the same techniques used for behavior engine, but on Level 2 of our behavior layers.

6.1 Design Decisions

In spring 2008 the Fawkes RSF had matured enough to provide a robust base for a robot system, and research started towards a behavior engine. Around the same time, the decision was made to use Fawkes on the Nao platform. This lead to broadening the scope of the behavior engine to a platform and domain independent behavior execution and monitoring system.

In the old RSF a separation of a reactive execution layer and a deliberative layer based on ReadyLog existed in the fourth generation. The skills were written in C++ and called from the agent by a custom language which formulated each skill as a function. The development flow and tool support had been cumbersome at best. Every modification of a skill required a stop-edit-compile-start cycle, with an especially lengthy compile part, which slowed down the development of the skills. Therefore interpreted scripting languages
were investigated as they promised quicker development because of the absent compile step, and the possibility of automated reloading. Cumbersome programming tasks like memory management can be avoided and scripting languages usually feature more elaborate reflection capabilities and support addition or modification of new code at run-time. This allows for easier supporting the developer with automated code generation and better instrumentation of the code.

The first language considered was URBI, as it was endorsed by Aldebaran for the Nao. But due to its limits and closed nature (cf. Section 3.2.3) it was rejected. Further research quickly led to Lua, as it is widely used in the video game industry to implement artificial intelligence, and as an extension language [86], e.g. in the CryEngine2 game engine for 3D action games [87]. It has also been used in critical applications like the NASA Space Shuttle Hazardous Gas Detection System\(^1\) and off-shore oil platforms [88]. These facts made us confident that Lua was up to the envisioned task. Technical reasons why Lua was chosen over other scripting languages are given in the next section.

An explicit goal for the behavior engine is to allow new programmers to quickly get into programming a robot’s behavior, with a steep learning curve. This is possible with Lua due to its simple syntax, influenced by Pascal and BibTeX and semantics similar to Scheme [88, 86]. Lua has been described as an elegant and easy-to-learn language [89] that should allow newcomers to start developing behaviours quickly. Although Lua poses a different language from the rest of the system, which is written in C++, we see this as strength for the behavior engine. There is a clear context change in a developer’s mind when switching from developing low-level components to behavior programming. During our own development we have seen that this supports faster switching between the two parts of the software, much like it helps to have a native car with the steering wheel on the matching side when driving on the unfamiliar lane in a different country.

6.2 Lua

Lua [90] is a scripting language, designed to be fast, lightweight, and embeddable into other applications. These features make it particularly interesting for the Nao platform. It has been developed at the Computer Graphics Technology Group of the Pontifical Catholic University of Rio de Janeiro since 1993. The latest versions are released under the permissive MIT License\(^2\). Henceforth we describe and use version 5.1 of Lua throughout this thesis.

---
\(^1\)List of Lua uses: [http://lua-users.org/wiki/LuaUses](http://lua-users.org/wiki/LuaUses)
\(^2\)MIT License: [http://www.opensource.org/licenses/mit-license.php](http://www.opensource.org/licenses/mit-license.php)
Lua has several technical properties that make it a good choice for integration into the software framework. The whole binary package takes less than 200 KB of storage. When loaded, it takes only a very small amount of RAM. This is particularly important on the constrained Nao platform and the reason Lua was chosen over other scripting languages like Python\(^3\), that are usually more than an order of magnitude larger [88]. In an independent comparison Lua has turned out to be one of the fastest interpreted programming languages [88, 91]. Another advantage of Lua is that it can interact easily with C/C++, which is important for a seamless integration into Fawkes. Lua is embedded in ANSI C\(^4\) which allows for portability to virtually all operating systems which have at least a C compiler.

Lua was originally developed as an extension language for programs written in C. Although fast, it does not try to mimic C in terms of sheer performance, low-level operations, and integration with third-party software, rather Lua relies on C for these features. What Lua offers is a good distance from the hardware, dynamic structures, no redundancies, and ease of testing and debugging [92]. Lua features a safe execution environment, meaning that errors can be easily detected and will not cause a failure of the whole program. With automatic memory management using an incremental garbage collector (see below) the behavior designer is freed from the most error prone tasks without a too bad performance hit.

We now describe some of the core features of the Lua language and its runtime system, based on [86, 92].

### 6.2.1 Values and Types

Lua is a dynamically typed language. Rather than attaching a type to a variable, types are attached to the value. The variable’s type depends on the value currently stored in the variable. Lua has eight basic types: nil, boolean, number, string, table, function, userdata, and thread. The nil type only has a single value also called nil, and is generally used as a marker for values that do not have a type (e.g. uninitialized variables). Boolean values are either true or false. Numbers are generally double precision floating point numbers. This can be changed when compiling Lua, but has not been changed for this thesis. Strings are arrays of bytes. Tables are associative arrays, described in more detail in the following subsection. Functions are references to Lua or C functions. A userdata value is a pointer to a memory block, supplied by the user and which is not managed by the Lua garbage collection. Finally threads are coroutines, which are part of the cooperative threading features of Lua.

---

\(^3\)Python: [http://www.python.org/](http://www.python.org/)

\(^4\)Standard that defines the minimum environment for C programs. Defined by the American National Standards Institute (ANSI).
6.2. Tables and Metatables

Tables are the only data-structuring mechanism in Lua. They are associative arrays that can be indexed by any value (except nil). Lua tables have two parts, an associative and a number-indexed array part. This is not visible from Lua code, but important to understand to gain the maximum performance. If values are stored indexed by numbers starting from 1 (Lua arrays do not start at 0) and not sparse, they are inserted in an array with constant access time. Arbitrary indexed values or sparse arrays are inserted into the associative part, employing an optimized chained scatter table. This two-fold handling of tables tries to combine high speed and a low memory footprint.

A very important feature of Lua are the so-called metatables. Metatables are ordinary tables with specific entries. They are attached to a value to change its behavior in specified situations. The keys in the metatable are called events and the values are metamethods. An event is stored in the metatable with two underscores prepended to the name. An example for that is the index event, which is stored as \texttt{__index} in the metatable. Let the metatable \texttt{M} be attached to the regular table \texttt{T}. Whenever a value of \texttt{T} is queried, for example with \texttt{T[k]}, and the table has no entry for the given key \texttt{k}, the index metamethod \texttt{M.__index(T,k)} is called. The metamethod can provide the value retrieved from somewhere else, or return nil to indicate its absence. As a specialty for the index event, the metamethod can be another table \texttt{I}. In that case, for keys absent for table \texttt{T}, the appropriate value from the metamethod table \texttt{I} is returned.

6.2.3 First-Class Functions and Proper Tail Calls

Functions are first-class types in Lua. They can be created at run-time, assigned to a variable, passed as an argument, and destroyed. They are regular values, with the property that they point to executable code. Especially the possibility to create functions at run-time is interesting and has been used in this thesis.

Lua functions can use named arguments by using a table as the only parameter. A special syntax for function calls can be used to abbreviate the call. Lines 1–4 in Listing 6.1 give an example of a function call with named arguments. Note the curly braces for the function call of \texttt{n} instead of round ones.

Another important fact is that Lua supports proper tail calls. Consider lines 6–8 of the code in Listing 6.1. In a language like C \texttt{g} would be pushed on the stack and then executed. Especially for more elaborated and deep recur-

---

Listing 6.1: Lua Tail Call and Named Arguments

```lua
1 function n(t)
2 print(tostring(t.x))
3 end
4 n(x=5) --> 5
5
6 function f()
7 return g()
8 end
```
sions this can cause a stack overflow. Lua however replaces \( f \) on the stack by the closure (see below) of \( g \) to avoid this problem.

### 6.2.4 Functables

Functables\(^5\) are not a Lua feature per se, but an expedient combination of basic Lua features. Functables are tables with additional semantics akin to a function. A functable \( T \) could be called as if it were a function with \( T() \). This is useful to keep state information for a function in a table.

A functable is created by attaching a metatable to a table, that has a call metamethod. If the table is called as a function, Lua will execute the call metamethod. Listing 6.2 shows an example. By attaching a metatable to \( T_1 \) it is made a functable and can be called as \( T_1() \).

#### Listing 6.2: Lua Functable

```
local T1 = { x=5 }
function f(t)
  print(tostring(t.x))
end

setmetatable(T1, {
  __call = f})

T1() --> 5
```

Functables are not a Lua feature per se, but an expedient combination of basic Lua features. A functable \( T \) could be called as if it were a function with \( T() \). This is useful to keep state information for a function in a table.

### 6.2.5 Function Closures

With functions being first-class values, that can be created at run-time, a special problem has to be considered. In Listing 6.3 the function \( \text{add} \) takes an argument \( x \) and returns another function, which will add the value of \( x \) to the parameter \( y \). After the inner function has been created and returned, \( x \) does not exist anymore because \( \text{add}(x) \) has ended. To solve this Lua creates a closure of the inner function with a so-called upvalue, a reference to \( x \). If \( x \) is only referenced by a closure the value will be copied. In the example the function \( \text{add2} \) is created that adds 2 to any number passed.

#### Listing 6.3: Lua Closure

```
function add(x)
  return function (y)
    return x+y
  end
end

add2 = add(2)
print(add2(5)) --> 7
```

With functions being first-class values, that can be created at run-time, a special problem has to be considered. In Listing 6.3 the function \( \text{add} \) takes an argument \( x \) and returns another function, which will add the value of \( x \) to the parameter \( y \). After the inner function has been created and returned, \( x \) does not exist anymore because \( \text{add}(x) \) has ended. To solve this Lua creates a closure of the inner function with a so-called upvalue, a reference to \( x \). If \( x \) is only referenced by a closure the value will be copied. In the example the function \( \text{add2} \) is created that adds 2 to any number passed.

### 6.2.6 Object-Oriented Programming in Lua

Because object-oriented programming (OOP) has been applied throughout the rest of the framework, it is natural to keep it for the Lua programming efforts. Classes have to be defined that have member variables and methods and can be instantiated to object instances. These objects have their own state independent of other instances. Inheritance allows for easy creation of sub-classes, possibly supporting polymorphism.

\(^5\)Functables: [http://lua-users.org/wiki/FuncTables](http://lua-users.org/wiki/FuncTables)
Classes are represented as tables. Members are fields, either being variables or functions. For proper instance separation, methods need an identity object (a `self`). This is accomplished by passing a self argument to each method as the first parameter. Lua provides syntactic sugar to automate this. A colon is used to pass an implicit `self` argument of the instance table. New objects are created by calling the `new` method. The class table is set as metatable to an instance table. This way all methods and (default) values of the class are available, and simply setting them to a different value changes the instance, but not the class. Listing 6.4 shows an example of a simple `Number` class. The method `add` adds to the value. The default value is 0, but once modified the value is stored in the instance. Lines 15 and 16 show the creation of two instances. Lines 18 to 21 modify the instances and print their values.

Later in the implementation we make use of duck typing. In this programming style an object’s type is determined by inspection of its method or attribute signature rather than by an explicit relationship to some static typing [93]. It goes back to the phrase if it walks like a duck and quacks like a duck, I would call it a duck. Rather than some explicit inheritance relation, it is enough for an instance to have all expected members, to be regarded as a valid class instance of the expected type. This matches the typing of Lua, and dynamically typed languages in general.

### 6.2.7 Threads and Coroutines

Lua supports cooperative multi-threading by the means of coroutines [94]. A coroutine is a line of execution, with its own stack, local variables, and instruction pointer, but sharing global variables and mostly anything else with other coroutines [92]. Multiple coroutines can be started, but only one coroutine can be running at any one time. Coroutines have to explicitly yield for other coroutines to run. These coroutines are created and managed via the coroutines library. The internal type for a coroutine is `thread`. Recently, true multiprocessing features have been added [95]. For this, processes can be created, which are separate instances of a Lua state. Different processes only communicate via so-called `channels` by passing messages. This also allows for syn-
chronization, because sending a message always blocks and the receiver can deliberately block. The multi-processing features have not been extensively used during this thesis, but might be useful for later extensions.

### 6.2.8 Integration of Lua with C

Lua can easily interact with C. The communication between C and Lua is stack-based. The Lua interpreter instance is called Lua state. It is either created from C when embedding Lua or is passed to a function for C modules (see below). For this stack, a protocol is implemented that allows to use Lua features with only a few dozen functions. As an example consider some function $f(x)$ that only returns the passed argument. To call this function from C, the function $f$ and the argument $x$ are pushed onto the stack. Finally by calling the Lua C-API function `lua_call(L, 1, 1)` the function on the stack is called, expecting the function to consume one argument and return one result. Likewise if a function of a C module is called, it will pop the arguments from the stack and will push the result, if any. As example see the function at line 1 of Listing 6.5.

### 6.2.9 Modules and Packages

Starting with version 5.1 Lua defines a set of policies and API support for modules, reusable software libraries, akin to libraries in C or modules in Python. A Lua package is a, usually coherent, collection of modules. Modules can be written in Lua and C.

Modules written in Lua have their own global namespace. The predefined `module` function sets up the module. It takes the module name and any number of `module initialization functions` as arguments. It will set a new empty table as the global environment. Everything that should be used from the original global environment has to be explicitly assigned to a local variable. The module initialization functions are called with the module table as the only argument. A special example is shown in Listing 6.12.

A C module is implemented as a shared library. The interaction between Lua and C happens via stack-based communication. The C module always defines the functions it should export with the same signature. Each function takes the Lua state as the only argument and returns an integer with the number of

---

**Listing 6.5: Lua C Module**

```c
static int l_add(lua_State *L)
{
    int a, b;
    a = lua_tointeger(L,-1);
    b = lua_tointeger(L,-2);
    lua_pop(L, 2);
    lua_pushinteger(L, a+b);
    return 1;
}
static const struct luaL_Reg clib[] = {
    {"add", l_add},
    {NULL, NULL} // sentinel
};
```
return values. Getting the arguments and returning the result is done via the stack as described in Section 6.2.8. A specific structure in the module registers the C functions with their symbolic names in Lua and is read automatically when the module is loaded. Listing 6.5 shows an example module with a single function that adds two integers.

### 6.2.10 Register-based Virtual Machine

The Lua code is compiled into a special instruction language, consisting of a number of so-called opcodes (operation code). Examples for such operations are ADD or MUL to add or multiply two values, or LT to check if a given value is less than another. For Lua the opcodes are closely related to core features of the language like tables, so there is an opcode NEWTABLE to create a new table.

These instructions are executed in a virtual machine (VM). Lua version 5.0 and later employ a register-based VM opposed to a stack-based VM as used for instance by Java or Python.

For a stack-based VM, which is more common, most instructions have implicit operands. All values are pushed onto a stack, and then an operation pops as many values as it needs arguments. For example a MUL operation expects two values on the stack. These are removed with two pop operations and the result is pushed back onto the stack. For a register-based VM, the instruction also defines the operands within the instruction, no stack operations are needed. The MUL is defined as MUL A B C, where the operands are read from the registers named B and C and the result is put into the register A. For the older stack-based Lua VMs the opcode size was one byte, which has now been increased to four bytes for the register-based VM. However, since less instructions are used the overall code size increases only marginally. The execution speed increases, since less instructions have to be executed. Espe-
cially the stack operations, which would be particularly expensive for Lua, are avoided. A code block like a function is stored with an array of registers in which all local values are stored. These can be accessed in constant time, opposed to the linear time required in a stack-based VM. The number of registers is limited to 200, but this barely poses a problem. Generally speaking code that would need more than 200 local variables should probably either use an array or be split into several functions. Real CPUs also operate register-based. This is exploited by the Lua just-in-time compiler\(^6\) by translating the Lua byte code into native machine code.

Listings 6.7 and 6.8 show the generated byte code in a register-based and a stack-based VM for the simple Lua program in Listing 6.6. This makes it easy to understand, why Lua outperforms many other scripting languages that employ a stack-based VM [91].

### 6.2.11 Incremental Mark-And-Sweep Garbage Collector

Lua uses automatic memory management with a garbage collector. Objects which are not referenced anymore are automatically collected and the associated memory is freed. More specifically, Lua employs an incremental garbage collector. Lua’s collector uses the mark-and-sweep algorithm, consisting of the two phases mark and sweep. In the mark phase all still reachable objects are marked alive. In the sweep phase all objects that have not been marked are destroyed. In traditional implementations however, the interpretation of the program is stopped to perform a whole garbage collection cycle. Stopping the execution of the program for the whole duration of the garbage collection can harm the performance of the system tremendously, especially for a real-time system like the robot’s behavior controller. Lua’s incremental garbage collector solves this problem by interleaving the execution with the garbage collection. Every time the interpreter allocates new memory, a small step of the collector is run. The same algorithm is used but not necessarily in one step. Since the reachability of an object might change during the step-wise execution of the collector, some operations check for dangerous situations and might change the marks applied in the mark phase.

### 6.3 Providing C++ Features to Lua

The Lua environment requires access to several features of the framework, like blackboard interfaces or the clock. These are implemented as C++ classes. Utility classes could be rewritten for Lua, but for most of the objects that have to be passed to Lua this is not possible. For instance blackboard interfaces and

\(^6\text{LuaJIT: http://www.luajit.org}\)
the central clock are provided by the framework. For blackboard interfaces an instance is connected to the particular memory block that is to be written or read. The clock is a shared resource with only a single instance. Therefore not only the pure functionality (i.e. getting the time) has to be imported, but rather a specific instance. For this adapters are needed. Generally, when a library written in one programming language has to be made available in another language a binding is required. So we are interested in a binding for framework classes written in C++ for Lua that is able to reference a specific instance. Fortunately Lua is well prepared to provide this. Lua has a special type *userdata* which is a pointer to some arbitrary memory block. Furthermore, Lua has support for loading modules written in C and can call (special) C functions. Finally Lua features metatables, which can be attached to a value to influence its behavior. These elements put together can be used to make C++ class instances available in the Lua environment, preserving the object-oriented properties and syntax.

Figure 6.2 shows an example. The class instance A, with the methods *method_1()* and *method_2()* , is to be made available in Lua. For this a userdata value is created in Lua, pointing to the instance. A metatable is attached to this value. The index metamethod points to another table that contains entries that point to C functions for every method. These functions, when called, get the userdata value pointing to the class instance and any other argument. The C function calls the method on the real object (exploiting the close relation of C and C++) and pushes the return value on the stack, if any.

So Lua provides everything required to make C++ objects available in Lua. However, it is tedious and error-prone to write this manually for every class. Therefore, several tools for automatic generation of such code have been written. We evaluated the most common available generators SWIG\(^7\), and tolua++\(^8\).

---

\(^7\)Simplified Wrapper and Interface Generator, SWIG: [http://www.swig.org/](http://www.swig.org/)

\(^8\)tolua++: [http://www.codenix.com/~tolua/](http://www.codenix.com/~tolua/)
The Simplified Wrapper and Interface Generator (SWIG) is very common to provide bindings of C/C++ libraries to scripting languages like Python and Lua. It takes an interface definition file and then creates code appropriate for the target language. SWIG has a specific shortcoming that hindered its use. It cannot generate code for sub-classes. Since blackboard messages of interfaces are defined as sub-classes, SWIG cannot be used. Toluasp takes a simplified header file as input, and creates the necessary C/C++ source code that can then be compiled into a shared library. Luabind\(^9\) was also considered. It utilizes C++ template meta programming to define the wrapper. It allows for a very efficient specification of the wrapper and supports sub-classes, but it would need manual code creation for every class again and thus was dismissed. Therefore tolua++ was chosen.

For blackboard interfaces, the interface generator was modified to create an appropriate input file for tolua++, besides the C++ code. For the most important utilities simplified headers have been created manually. The build system was extended with appropriate rules to allow for simple building of Lua wrapper modules. Wrapper modules were built for the required utilities.

### 6.4 Behavior Engine

In this section we describe the implementation of the behavior engine as a single plugin, called *skiller* (skill execution run-time), for the Fawkes RSF. The skiller plugin has been implemented agnostic for the approach used to model skills. We explicitly want to leave the door open for further experiments with different modeling approaches.

First we describe the implementation of the skiller plugin in Section 6.4.1. In the Section 6.4.2 the embedded Lua environment is described. Section 6.4.3 explains the general structure of a skill module followed by a discussion of the skill execution, sandboxing and communication with the agent in Section 6.4.4. Finally Section 6.4.5 explains a predicate system written in Lua.

The skiller has been implemented in a way that it can be used unmodified for multiple platforms and domains in general, and the Nao and MSL robots in particular for this thesis. The skill space (cf. Definition 3) determines the required interfaces for reading data and sending commands and a Lua configuration file defines the available set of interfaces.

#### 6.4.1 Skill Execution Run-Time Plugin

The skill execution run-time (skiller) plugin embeds a Lua interpreter into the Fawkes process and conducts the communication with the other components,
6.4. Behavior Engine

Listing 6.9: Skill Set Definition for the Nao Platform

```lua
local skillset = {
  "generic", -- Generic skills for all platforms
  { "relgoto", "goto", "turn", "say", "serialexec" },

  "nao", -- Nao specific but general skills
  { "servo", "head_pantilt", "led", "beckon",
    "getup", "park", "standup" },

  -- Soccer skills, generic and Nao specific
  "generic.soccer", { "intercept_ball" },
  "nao.soccer", { "kick", "search_ball" }
}
skillenv.use_skills(skillset)
```

especially with the agent component.

The plugin consists of a single execution thread which has aspects to hook up with the main loop and to gain access to the configuration, logging facilities, the blackboard, and the central clock. The thread runs at the skill hook (cf. Section 4.2.3). In the following we describe certain aspects of the plugin in more detail.

**Interface Configuration**

The skiller plugin requires a certain set of interfaces, depending on the skill space. These are used to get information from other components, especially the world model, and to send commands, e.g. to actuator components. The set is defined in a sub-space of the configuration as a number of key/value pairs. The key is the configuration path, which determines the opening mode of the interface (reading or writing) and the name of the variable in the Lua environment. The value defines the unique identifier of the interface. The identifier is a patterns possibly containing the wildcard characters ? (a single character) or * (any number of characters, including none). In that case all interfaces matching the pattern are opened and an observer is installed, that observes the blackboard for new interfaces with an identifier matching the pattern. It automatically opens any new interface. This is for example useful, to open all object position interfaces for obstacles. New ones may appear at run-time, for example if a human steps on the field. All opened interfaces are pushed into the Lua environment and automatically updated in every cycle.

Two interfaces are always opened for writing. One is the SkillerInterface, the other is the SkillerDebugInterface. The former is used to provide status information and accept execution orders, the latter to provide information about the internal state. They are discussed in more detail in Sections 6.4.4 and 6.7.
Skill Set Definition

Depending on the skill space a certain set of skills is available. This set is constrained by platform and domain, but may also be manually reduced, for example to provide a simpler or constrained environment for students. In the current implementation, there is one initialization file written in Lua, that is loaded at the beginning. It defines the available skills by requesting them from the skill plugin. The plugin then loads the appropriate skill module and verifies that the minimum requirements for that skill are met, i.e. that all skill and interface dependencies are fulfilled. Listings 6.9 and 6.10 show the skill set definitions for the Nao and MSL platforms respectively that were implemented and used during this thesis.

Automatic Reloading

For reasons of efficiency plugins can be reloaded without restarting the whole process. For even quicker development cycles, support for automatic reloading of the script files has been added to the skiller plugin. For this inotify [96] has been used. This is a facility provided by the Linux kernel for automatic notification about changes to a file system. Using inotify all script files are monitored and when files are changed, removed or added the Lua environment is reloaded. A new Lua context is created and initialized, and only if this succeeds, the old context will be replaced, otherwise an error will logged and the old context will stay in use.

6.4.2 Lua Environment

We have discussed the integration of the Lua environment into the framework. Here we discuss the internals of the Lua environment itself.

Certain core features have to be made available to the Lua code. In particular, these are the logging and configuration subsystem, as well as the central
clock. The standard print functions for text output have been re-implemented to use the logging facilities. Therefore, all regular Lua code, using the standard Lua output facilities will write to the logging sub-system. This ensures that all messages are available at a central place, and over the network (cf. Section 4.2.11). The configuration sub-system can be used for instance to configure parameters for skills. The central clock is used in the skills for example to wait for a specified time or to implement timeouts. It is essential to use the central time for the reasons outlined in Section 4.2.10. The configuration subsystem allows skills to read specific parameters, the intercept skill might for instance need the ball size and desired distance from the ball to achieve, which depends on the domain at hand.

A start script is executed which performs initialization functions to setup the Lua environment, open the blackboard interfaces, and prepares the skill environment for the configured skill space.

### 6.4.3 Skills

#### Skill Module

From the implementation perspective a skill is a Lua module, which provides certain variables and functions for interaction. The implementation as a module has several advantages. The skill module is properly encapsulated with its own global environment, which does not interfere with the base system or other skills. A skill is free to implement anything required in this module. Since a module is just a table, it can be manipulated by the skill environment, for instance by setting a metatable. Skills can be grouped to packages, e.g. for a particular platform. A skill has the following basic variables and functions.

- **name**: The name of the skill. It does not necessarily have to match the module name to allow for different implementations of a skill, only one of which can be included for a skill space at a time.

- **documentation**: A string describing the skill. Specifically it should contain the purpose, preconditions and arguments of the skill.

- **depends_skills**: A list of skill names this skill depends on. If any of the dependencies is not available the skill will not be loaded and an error will be thrown.

- **depends_interfaces**: A list of interfaces required by the skill. Interfaces are defined by an associative array with three elements. The field \( \nu \) denotes the variable name of the interface, by which the interface should be accessible and the field \( \text{type} \) defines the type of the interface. The third field \( \text{id} \) is optional and if given will be the identifier of the interface (which
forms the unique identifier in combination with the type). The variable name corresponds to the name in the configuration. This is to allow for fast switching between two component interfaces.

**init()** This function is executed once when the module is initially loaded. It is usually used to initialize state structures.

**execute()** This function triggers the actual execution of the skill. More precisely it executes one step of the skill (cf. Section 6.4.4 about skill execution). Any parameters can be passed, named parameters are preferred. Especially for the SHSM modeled skills this is important to allow for automated binding of variables. The execute function returns any number of values, but the first value must be the status code of the skill. This status is either S_RUNNING if the skill is still running, S_FINAL, or S_FAILED if the skill succeeded or failed respectively.

**reset()** This function is executed when a skill has finished execution or has been stopped. It is usually used to reset internal state structures.

### Skills with Finite State Machine (FSM)

Since we aim to provide specific support for modeling skills by hybrid state machines the code for basic functions described above is usually the same. The **init()** and **reset()** methods reset the state machine, erasing the internal state and on the first run preparing the FSM for execution. Thus for skills with HSMs, or any form of FSMs, the optional **fsm** variable, described below, can be used as a shortcut.

**fsm** If this field is set to a FSM providing **reset()** and **loop()** functions, the basic skill functions become optional. On initialization and reset **fsm:reset()** is called. An **execute()** wrapper function is created and installed in the module. It assigns arguments to the **fsm.vars** field and calls **fsm:loop()**. Likewise **reset()** is added calling **fsm:reset()**.

### Skill Functable

To the agent and other skills a skill is available as a *protected functable*. A metatable is attached to an empty table. The call metamethod is set to an anonymous wrapper function, the execution function. This function calls skill environment functions for housekeeping, for example to evaluate the return value of the skill’s **execute()** function. The code of the wrapper generator function is shown in Listing 6.11. This way every exported function and value of the skill is accessible in the execution sandbox. But it cannot be modified, because a writing access will only modify the functable, but not the skill module.
6.4. Behavior Engine

Listing 6.11: Skill Wrapper Generator Function

```lua
function create_skill_wrapper(skill_module)
    local t = {};
    local mt = {
        __call = function(skill, ...)
            skill_loop_begin(skill.name)
            rv = {skill.execute(...)}
            skill_loop_end(skill.name, rv[1])
            return unpack(rv)
        end,
        __index = skill_module
    }
    setmetatable(t, mt)
    return t
end
```

6.4.4 Skill Execution, Sandboxing and Agent Communication

Skill Execution

Skills are ordered for execution via the SkillerInterface. First, one of the interface readers acquires the exclusive control over the execution thread using a subscribe mechanism via the blackboard interface. Only after it successfully acquired the control it can send commands to the skiller. Successive subscription requests or command execution from other threads are denied. It is to ensure that there is only one instance giving orders, to guarantee that there are not two or more commanding instances competing and interfering with each other.

Skills are ordered for execution by forming a valid Lua string, called the skill string. This skill string may contain anything that is provided in the sandbox environment (see below), especially skill functables. The skill string can call multiple skills or contain conditionals. However, each skill may be executed at most once. This does not only account for skills explicitly executed in the skill string, but for all dependencies of these skills. This constraint has been added to keep the execution overhead low (skills do not need to be instantiated and state information does not have to be copied) and since usually it is not very useful to execute the same skill multiple times at the same time (skills usually order an actuator as effect, which is a resource available usually only once). A valid complex skill string could look like the following:

```
    search_ball(); say{text="Searching for the ball"}
```

This would search for the ball and at the same time say “Searching for the ball”. The skill string is compiled into a function, called skill function, concealed into a sandbox and then executed. When the execution is stopped all involved skills are reset using their reset() functions. Skills, that are explicitly called in the skill string are called top skills. Only these skills define the overall return status.
Skill execution happens step-wise, interleaved with the rest of the system. The execution thread is woken up at the skill hook and at each wake up the skill function is called, if any. Skills can be executed once or continuously. For a one-shot execution the skill function is executed exactly once and the reset happens immediately in the same cycle. For a continuous execution, the skill function is compiled, and then executed in the current and consecutive loops, as long as no new skill string or a stop command is received. The reset is executed when the skill function is stopped. During the continuous execution the internal state of the involved skills is preserved.

Most of the time the skill function is executed continuously to allow for internal state over multiple loops. The execute() functions of the top skills are executed by the skill function in every loop. The function performs a single step and immediately returns. It may not run until the skill has finished. We considered using Lua coroutines for skills, enforcing that each skill is a coroutine. But for the chosen modeling approach, this would just have caused overhead. However, a skill might still be implemented using coroutines. In this case the execution function could resume the skill coroutine, and when it yields the execution function returns.

**SkillerInterface Messages**

As described above, the skiller accepts messages for acquisition and release of exclusive control, to execute skill strings once or continuously and to stop a continuous execution. Therefore the following command messages have been defined for the SkillerInterface. The messages are only accepted by the exclusive controller unless stated otherwise.

**AcquireControlMessage** This sends a request for subscription as the exclusive controller of the skiller. The request is accepted from any sender if no other exclusive controller registered before.

**ReleaseControlMessage** This causes the release of the exclusive control. The skiller registers as a blackboard listener for the SkillerInterface. If the reader, that subscribed as the exclusive controller, leaves the subscription will be revoked automatically. This is done to cope with plugins improperly written that do not release the control. Still it is important to have an explicit release control message, for example to be able to use debug tools that show the internal state. They must be able to acquire the control (to execute skills) and to release it to show the internal state and yet allow another controller (like an agent) to take over.

**ExecSkillMessage** This orders the one-time execution of a given skill string.

**ExecSkillContinuousMessage** This orders the continuous execution of a given skill string.
6.4. Behavior Engine

StopExecMessage This stops the current continuous execution of the skill function if any, and causes a reset.

RestartInterpreterMessage Restart the Lua interpreter. This is useful if the automatic reloading on a file change has been disabled or is not available, which is likely to be the case during a tournament game.

Sandboxing

Lua is a comprehensive programming language, that supports operating system functions like command execution and file handling. This poses a potential threat. On the Nao for instance, there is only the root super user, which can do anything on the system. Therefore the Fawkes process has the ability to make use of all operating system features, including erasing any file on the system with a simple command. If the Lua environment was unconstrained, the system could be severely damaged. Besides such an offensive and malicious behavior, in a more likely scenario a programmer might cause side effects with a skill string by unwillingly modifying the Lua environment harming the performance of consecutive skill execution. If all skill strings were executed in the very same environment, the first skill string could for instance assign nil or a new custom function to a skill. The next skill string that includes this skill would then be unable to call it.

To avoid this problem, for every new skill string a sandbox is created in which the skill function is executed. The sandbox contains only selected functions and packages, and newly created functables for the skills. Lua already provides the \texttt{setfenv} function, to set a function environment, that replaces the global environment by a table that contains global variables, functions etc. available to the function. Starting from an empty table, items from a template table are copied. Finally, fresh functables for all available skills are generated and added. The created table is then assigned as the new environment for the skill function. Henceforth, the execution of the skill function can only use the features that have been made available, thus protecting the skill execution from unwanted side effects.

Agent Communication and Skill Status

The communication of the skiller with the agent involves a way of sending execution orders from the agent to the skiller, and status reporting from the skiller to the agent. The order messages have been described in Section 6.4.4.

The agent needs information if a skill string is still being executed, or if it has succeeded or failed. Therefore, the skiller writes the \texttt{skill status} to the SkillerInterface. A skill string might involve several skills to be executed at the same time. However, the skill status is only determined by evaluating the
CHAPTER 6. IMPLEMENTATION OF A LUA-BASED BEHAVIOR ENGINE

return values of the top skills (cf. Section 6.4.4). It is a skill author’s responsibility to ensure that a skill properly handles and communicates return values of sub-skills. The following list gives all possible skill status values and the conditions when they are returned.

**S_INACTIVE** No skill string is currently being executed.

**S_RUNNING** A skill string is currently being or has been executed, and neither has any top skill failed nor have all skills succeeded.

**S_FINAL** A skill string has been executed and no top skill is running or has failed but all top skills have succeeded.

**S_FAILURE** A skill string was executed, but at least one top skill failed. The execution has been stopped.

This aggregated return value allows an agent program to monitor the execution of the skills. If the agent needs to know the return value of a particular skill, it either needs to decompose the skill string into a serialized execution of single skills, or write a more complex skill string that handles the return value of the particular skill.

When an agent plugin is loaded, it acquires the exclusive control over the skiller. If that failed the agent should report an error, as it cannot operate properly. Based on its decisions it orders the execution of the skill by forming an appropriate skill string and sending it to the skiller. Commonly an agent will choose continuous execution of the skills, to allow skills to keep state information. A one-shot execution would trigger an immediate reset erasing state information. Most skills, especially more complex skills, will need state information to yield the expected result. The agent continuously evaluates the situation and makes decisions about the next action. If a new skill string is to be executed, the agent will simply overwrite the old one by sending a new message. Sometimes, an explicit stop may be needed, for example if the referee ordered an immediate freeze.

### 6.4.5 Predicates

For short, a predicate is an operator in logic which returns either true or false [97]. We are interested in expressing specific boolean properties of the world situation in the current loop, for instance “ball is in own half”. This is especially interesting because boolean functions are used as jump conditions for skills modeled as SHSMs (cf. Section 6.5), thus predicates and jump conditions fit together nicely. Therefore we implemented an efficient way for developing and using such predicates by so-called *predicate library modules* (PLM). Since predicates are potentially used many times in a single iteration in different
jump conditions, a caching algorithm should be used to avoid multiple calculations of the same value. Likewise a calculation of all predicate values no matter whether they are used or not should be avoided. An assumption that makes this possible is that the situation does not change within a single loop iteration. This is true because the blackboard interfaces are only updated once at the beginning of the loop.

The base module `fawkes.predlib` provides the tools required to build a PLM. Each predicate is defined as *predicate function* which returns either `true` or `false`. Since the value should be cached, some effort has been made to reach an efficient implementation that avoids the evaluation of the predicate function every time the value is required. The `fawkes.predlib` module provides a function used in the PLM as module initialization function. This function creates a *predicate table*, containing the predicate names as keys and predicate functions as values, and sets a metatable on the module, which makes the ordinary module a PLM. The declared predicate functions are not stored in the module, but rather in the predicate table. The actual predicate name remains unused in the module and is provided by an index metamethod. Predicates are accessed as if they were boolean *predicate variables* of the module and not as functions. Although this can be seen as unexpected behavior, it is necessary for automated caching and to minimize the amount of code required to define and use the predicates. The function also registers the PLM with the base lib. The metatable has metamethods for the index and newindex events.

The *newindex metamethod* is called whenever a new value is set on the module table, i.e. when a new non-local function or any other value is defined. The only non-local values a PLM may have are predicate functions. Any other new value is rejected with an error. Helper functions and internal variables may still be stored in local variables. Predicate functions are added to the predicate table. The *index metamethod* is called the first time a predicate value is requested. It executes the appropriate predicate function, and assigns the value to a field in the module table with the predicate name bypassing the newindex metamethod. Any consecutive usage of the value will cause the value to be returned immediately from the module table, without calling the index metamethod. Thus the value has been cached and the predicate function is evaluated at most once. Additionally, unused predicates are not calcu-
ated. Hence any number of predicates can be defined without a performance penalty as only the ones used are actually calculated.

At the end of a loop, the `fawkes.predlib.reset()` function is called to reset all predicates by setting all used predicate values to nil. Predicates can easily be used as jump conditions by utilizing the text based format of jump conditions (see above).

Listings 6.12 and 6.13 show the definition of a PLM, and the usage thereof. In Listing 6.12 lines 1 and 2 initialize the module as a PLM. Lines 4 to 6 define a predicate function, which determines if the current day is Sunday. Since the module is a PLM, it is made a predicate. The module does not have a function `sunday()`, rather it has the predicate value `sunday`, because the function is only stored in the predicate table. Line 1 in Listing 6.13 loads the PLM and assigns it to the local variable `p`. Lines 2 and 3 are used to print the value of the `sunday` predicate, but only line 2 triggers the evaluation of the predicate function. In line 3 the cached value is used.

### 6.5 Skills Employing Skill Hybrid State Machines

In the previous section the general behavior engine framework, its implementation in Lua and how a skill is operated has been described. Now, based on the theoretical foundations described in Chapter 5, we describe the implementation of Skill Hybrid State Machines (SHSMs) in the Lua-based behavior engine.

#### 6.5.1 Finite State Machine Driver and Basic States

Since finite state machines (FSMs) are a recurring pattern that has been modeled and implemented in several different ways, the basis for our SHSMs forms a generic FSM driver class. For its basic functionality only very few functions are necessary. The actual specifics of a particular FSM implementation are pushed into the implementation of the state class. First we describe the `State` class that forms the superclass for all states supported by the FSM driver, detailed thereafter.

**Basic State**

The `State` class is the superclass for any state that can be used with the FSM. True inheritance can be used for writing subclasses, but implementing a different type of state by duck typing (providing all required members, cf. Section 6.2.6) is sufficient. The `State` class provides the following basic members, as expected by the FSM. We have to anticipate knowledge of the following sub-
section about how state transitions are indicated by returning another state object in a function.

**do_init()** Called when the FSM enters the state by executing a transition that leads to this state. A state may trigger an immediate state change by returning a successor state. Any values returned by the predecessor state’s `init()` or `loop()` method that triggered the transition are passed as arguments, without the state object itself.

**do_loop()** Called in every cycle when the state is the current active state of the FSM. A state change is indicated by returning the successor state. Additional return values are passed to the successor state’s `init()` method.

**do_exit()** Executed just before a state is left, i.e. by returning another state in `init()` or `loop()`.

**reset()** Executed during a reset of the FSM this state is associated to.

**prepare()** Executed exactly once when the FSM has been initialized and is reset for the first time.

The `do_` prefixed functions are implemented in the particular state class. The unprefixed versions may be overridden for a particular instance of a state class. When the basic `State` class is used, the unprefixed methods get the same arguments and return the same values as the `do_` prefixed methods. For different state implementations (as described for instance in Section 6.5.2) the expected return values or passed parameters may differ. Since they are only passed between states and regarded by the state’s base class implementation, the FSM can handle the state as long as the `do_` prefixed methods adhere to the described contract.

**Finite State Machine Driver**

The FSM driver is implemented in the `FSM` class. It provides methods to add and remove states. States are associated with a specific FSM and can only be used once. The FSM has exactly one *current/active state*. State transitions are not added explicitly to the FSM, rather the `do_init()` and `do_loop()` methods of a state may return a successor state, which will cause a transition to this state. If nil is returned, no state change is performed. This allows for different concepts how state transitions are formulated, and moves this part of the design to the state implementation.

The core of the `FSM` class are the `loop()` and `trans()` methods shown in extracts in Listing 6.14. The `trans()` method executes a number of transitions. Given a valid `next_state`, the `do_exit()` method of the current skill is called and the next state is set as new current state. The `do_init()`
method of the new active state is called. This might return a successor state and additional arguments. Therefore trans() is called recursively possibly executing a number of transitions immediately. As Lua supports proper tail calls, there is always only a single trans() execution running, thus avoiding a stack overflow. In the full code a maximum number of transitions per loop can be configured. The loop() method is called once per main loop cycle. If no current state has been set, the FSM had not been run and a transition to the start state is executed. The do_loop() method of the current state is called, we say the active state is executed, and the result is passed to trans(), as it might trigger a transition. The reset FSM:reset() is called to reset a FSM. The current state is left and set to nil. The reset() method of all associated states is called. On the first call of the FSM lifetime the prepare() methods of associated states are executed. The FSM class has a member table called vars. In this table, variables of the FSM can be stored. This is especially helpful to implement the set $X$ of variables of HSMs. It is cleared during a reset.

The FSM driver provides a very flexible system for implementing different types of FSMs, the SHSMs being only one particular model. During the RoboCup 2008 in Suzhou, the “Who is who” challenge\textsuperscript{10} of the RoboCup@Home League was implemented using the FSM driver and only basic states as a first proof of concept and ran successfully.

\textsuperscript{10}The robot explores the arena detecting, recognizing and remembering persons. Later the persons pass the robot and he has to recognize them.
6.5.2 Jump States and Jump Conditions

To implement HSMs a state class called JumpState has been implemented. Combined with the FSM class this constitutes a HSM. Jump states got their name from the jump conditions, which are used in HSMs for defining conditions for state transitions as boolean functions.

A transition of a jump state is defined as a tuple of a successor state and a jump condition. In each cycle, the jump conditions for the current state are evaluated. If they evaluate to true they will cause the associated transition to be executed, we say they trigger or fire. If multiple jump conditions would fire, only the first transition will be executed, in the order the transitions have been added. The init(), loop() and exit() methods can freely be implemented. For jump states any values returned by these functions are ignored. Therefore, a state change cannot be triggered by returning a target state. Rather a jump condition must trigger. Transitions can be marked as a precondition. They are then evaluated during the do_init() method before init() is called. This can be used for instance to check if a certain interface has a writer to immediately go to the failure state if it does not, or to implement invariant conditions.

Three different ways of defining a jump condition are supported by the JumpState:add_transition() method. A jump condition can be either a regular boolean function, or the value true which causes an unconditional transition. The third option allows to pass a string of valid Lua code, which must form a boolean expression. This string is compiled into a boolean Lua function. The latter method is especially useful when using predicates. Since the predicates themselves are used as boolean values rather than functions, they need to be wrapped to be used as jump condition, which can easily be accomplished by passing the predicate as a string.

6.5.3 Skill Hybrid State Machine

Skill Hybrid State Machines (SHSMs), based on the theoretical foundations from Chapter 5, are implemented in a SkillHSM class and a number of derivatives of the JumpState class.

The SkillHSM class extends the FSM class by convenience utilities that support the developer when implementing the proposed model. It supports the automatic creation of the FINAL and FAILED states, which map to the $q_{\text{final}}$ and $q_{\text{failure}}$ states respectively. They are automatically mapped to the appropriate skill return values.

The two SkillJumpState and SubSkillJumpState classes inherit from the JumpState class. The most notable differences are the support for sub-skills and an (optional) automatic creation of transitions to another state based on a sub-skills status value. The SkillJumpState is more generic. A num-
number of sub-skills can be added to a list, which are automatically reset when entering and leaving the state. The SubSkillJumpState takes one sub-skill and successor states for succeeded and failed sub-skill execution. The skill is executed automatically in each loop. The init() and loop() methods can be implemented, and are mostly useful for generating appropriate parameters for the sub-skill.

The SkillHSM class has a method add_transitions(), which allows to add a number of transitions defined in a single table. Additionally, any state that does not yet exist is automatically created. This vastly reduces the amount of code required to write a skill.

6.5.4 Implementation of a Skill Using a SHSM

In the previous sections we have shown the elements of the behavior engine plugin, the Lua environment and skill definition. In the preceding part of this section we described the combination of a generic FSM driver with jump states to create HSMs and an extension thereof to streamline the development of a skill. Now we describe how these elements are put together to implement a skill based on Skill Hybrid State Machines.

A skill is a Lua module with specific variables and functions. In the case of a skill that employs some form of FSM the functions are optional as long as no special operations are necessary (Section 6.4.3). So for SHSM skills we need to set the name and documentation appropriately, define the required sub-skills and interfaces via depends_skills and depends_interfaces and assign an instance of SkillHSM to the fsm variable. The skill can now already be added to the skill space.

Skills are modeled as SHSMs. Therefore, we need to add the states and transitions. For convenience, to add skills and transitions sequentially the SkillHSM:add_transitions() method can be used. It takes a single table that defines all transitions, and creates all states as necessary (Section 6.5.3) and automatically creates the FINAL and FAILED states, therefore creating the graph $G = (Q,T)$. The set $X$ of variables is represented by the fsm.vars table. It contains an entry for each variable. The auto-generated execute() function automatically assigns all parameters to the fsm.vars table. Jump conditions are represented as boolean functions, that can be defined in multiple different ways. In the implementation the basic actions $A'$ are the messages that can be sent via the blackboard to the low-level subsystems, therefore the set $A'$ is defined by the depends_interfaces fields of skill modules. Additionally, there is one special action which can be defined by implementing the loop() function of a state. The set $D$ of dependencies of a skill is defined by the depends_skills field. The $k$-functions are the functables of the dependency skills. Invariant conditions can be implemented using precondi-
6.6 Skills for the Target Platforms and Domain

6.6.1 Nao Skill to Search for the Ball

In the previous sections we have discussed the execution environment for skills, how a skill describes its minimum requirements and what it has to provide. Now we describe the search_ball skill as an example. The skill is specific to the Nao platform and used during a robot soccer game to find the ball. To achieve this the robot employs the mouth camera. It first looks down to the right, and then turns its head slowly to the left to find a ball that is somewhere close in front of the robot. If the ball cannot be found the robot will raise its head and look right, to detect the ball in a larger distance. If the ball is still not visible, the robot will turn two times around its vertical body axis. On the first turn it looks for a ball in a close range and in the second it looks for more distant ball sightings. If the ball is found, the robot will stop immediately and will return S_FINAL. If the ball is not visible after all the movements S_FAILED will be returned. In this case the agent could choose to search again for the ball or to walk somewhere else on the field to look there.

Listing 6.15 shows the implementation of the skill. Lines 2 and 20 are used to initialize the module as a skill module. The initialization function skillenv.module_init() will add some basic utilities provided by the skill environment, like the SkillHSM class and skill jump states. skillenv.skill_module() will do most of the initialization work. Based on the basic skill variables defined in lines 7 to 17 the skill module is initialized (cf. Section 6.4.3). The fsm variable is initialized with a SkillHSM instance, the servo and turn skill are required as sub-skills, and a HumanoidMotionInterface with the variable name naomotion is requested. The skill_module function ensures, that the minimum requirements are met. In line 4 a predicate library is loaded. Here these are the general soccer predicates, from which the ball_visible predicate will be used in line 48. Most of the states are simply SubSkillJumpState instances. In lines 28 to 42 transitions are defined, which do not have explicit jump conditions, but monitor execution of a sub-skill. If
-- Initialize module
module(..., skillenv.module_init)

local p = require("predicates.soccer.general")

-- Crucial skill information
name = "search_ball"
sfm = SkillHSM:new{name=name, start="LOOK_DOWNRIGHT"}
depends_skills = {"servo", "turn", "stop"}
depends_interfaces = {}
documentation = 

Parameters:
no_turn if true no turning is performed

-- Initialize as skill module
skillenv.skill_module(...)  

local BODY_ANGLES = { -- omitted -- }

-- States
fsm:add_transitions{
closure={p=p},

{"LOOK_DOWNRIGHT", "LOOK_LEFT", 
skill=servo, args={body_angles=BODY_ANGLES, time_sec=2.0}},
{"LOOK_LEFT", "LOOK_UP_1", 
skill=servo, args={head_yaw=0.8, time_sec=4.0}},
{"LOOK_UP_1", "LOOK_RIGHT", 
skill=servo, args={head_pitch = -0.25}},
{"LOOK_RIGHT", "CHECK_TURN", 
skill=servo, args={head_yaw=-0.8, time_sec=4.0}},
{"LOOK_DOWNMIDDLE", "TURN_RIGHT_1", 
skill=servo, args={head_pitch = 0.45, head_yaw=0.0}},
{"TURN_RIGHT_1", "LOOK_UP_2", 
skill=turn, args={angle = -2*math.pi}},
{"LOOK_UP_2", "TURN_RIGHT_2", 
skill=servo, args={head_pitch = -0.45}},
{"TURN_RIGHT_2", "LOOK_DOWNRIGHT", 
skill=turn, args={angle = -2*math.pi}},
{"CHECK_TURN", "LOOK_DOWNMIDDLE", "vars.do_turn", 
precond=true},
{"CHECK_TURN", "LOOK_DOWNMIDDLE", "not vars.do_turn", 
precond=true},
{"STOP", "FINAL", skills={{stop}}},
-- default transitions
{"STOP", cond="p.ball_visible"}
}
the sub-skill succeeds, a transition to the given target skill will be executed, otherwise the FSM will go to the **FAILED** state.

### 6.6.2 MSL Goalie Skill

The goalie skill places the robot in front of the goal, and watches the game. It has been developed for the MSL league, which means that no camera movement is necessary. Therefore, the robot can just wait until the ball is visible. The robot has three positions in front of the goal, left, middle, and right. Initially the robot assumes to be at the middle position. When the ball is left of the robot, i.e. the relative angle \( \theta \) from the robot’s forward direction to the ball is greater than 0.35 the robot will move to the left, when \( \theta < -0.35 \) the robot will move rightwards. The skill has been developed quickly to test the general ability of the state machines. For a real game the goalie would need a continuous range of movement and not only three discrete positions.

The source code of the skill is show in Listing 6.16. The initial setup is similar to the Nao skill. In this example however we use custom jump conditions,
to check for the ball sectors to the left and right. Still the `p.ball_visible` predicate is used to check for the visibility of the ball.

**Listing 6.16:** Simple Goalie Skill for the MSL Robot

```lua
-- Initialize module
module(..., skillenv.module_init)

local p = require("predicates.soccer.general")

-- Crucial skill information
name = "goalie"
fsm = SkillHSM:new{name=name, start="GOALIE_MIDDLE"}
depends_skills = {"transrot"}
depends_interfaces = {
  {v="wm_ball", type="ObjectPositionInterface", id="WM Ball"}
}
documentation = "Goalie skill. No parameters."

-- Initialize as skill module
skillenv.skill_module(...) 
p.setup{ball_left_angle=0.35, ball_right_angle=-0.35}

fsm:add_transitions{
  "GO_MIDDLE_LEFT", "GOALIE_LEFT",
  skill=transrot, args={vx=0.0, vy=2.0, time_sec=0.8}),
  "GO_MIDDLE_RIGHT", "GOALIE_RIGHT",
  skill=transrot, args={vx=0.0, vy=-2.0, time_sec=1.1}),
  "GO_LEFT_MIDDLE", "GOALIE_MIDDLE",
  skill=transrot, args={vx=0.0, vy=-2.0, time_sec=1.1}),
  "GO_RIGHT_MIDDLE", "GOALIE_MIDDLE",
  skill=transrot, args={vx=0.0, vy=2.0, time_sec=0.8}),
  "GOALIE_MIDDLE", "GO_MIDDLE_LEFT", "p.ball_left"),
  "GOALIE_MIDDLE", "GO_MIDDLE_RIGHT", "p.ball_right"),
  "GOALIE_LEFT", "GO_LEFT_MIDDLE", "p.ball_right"),
  "GOALIE_RIGHT", "GO_RIGHT_MIDDLE", "p.ball_left")
}
```

### 6.7 Skill Development Tools

Developing a robot’s behavior is tedious, error-prone and time consuming. Therefore good tool support can help immensely to speed up the development. We have always kept this in mind while developing the behavior engine, by minimizing the code required to get things done. We will first describe the support integrated into the Lua environment to create graphs of state machines in
Section 6.7.1. Section 6.7.2 describes the SkillGUI, a tool to trigger and monitor the execution of skills. Finally in Section 6.7.3 we describe the simulator integration that has been developed for the Nao platform and preparations for a simulator integration for the MSL.

### 6.7.1 Graph Generation via Graphviz

A finite state machine can easily be visualized as a graph, in which the states are represented as nodes and the transitions are the edges. This is indeed the basis for the definition of HSMs (cf. Definition 2 on page 55).

One of the most popular graph visualization libraries is Graphviz [98, 99]. It was originally developed by AT&T and is now available as Open Source software. It is a collection of graph drawing libraries and tools that evolve around a common graph description language, often referenced as the DOT graph language.

Graphviz provides Lua bindings for the graph library offering an API for generating graph descriptions using the DOT language. However, a custom Lua module has been written for graph generation for more flexibility and because the library sometimes caused problems on the robot. Using this module graphs can be generated automatically for finite state machines built with the generic FSM driver. The FSM class keeps track of graph modifications in the current cycle. If it was changed, the grapher creates a new representation of the graph. Since the graph generation itself takes time and may influence the whole system it may be turned off.

The graph is published via the SkillerDebugInterface, which has an 8 KB text buffer. Since multiple skills may be active at a time, it defines a message for setting the graph. A list of available graphs can be produced in the text buffer on request. The graph contains annotations for marking the active state, highlighting potentially problematic transitions (transitions leading out of an exit state), and following the execution trace (see below). Additional annotations can be set per state and transition and are added verbatim to the graph.

### 6.7.2 SkillGUI

The SkillGUI is an application with a graphical user interface (GUI) that connects to a Fawkes instance and communicates to the skiller via a remote blackboard connection using the SkillerInterface and SkillerDebugInterface.

The SkillGUI has two major views. The first view shows the log messages of the connected Fawkes instance. For this it subscribes to the logging subsystem (cf. Section 4.2.11) over the Fawkes network protocol. Henceforth all log messages are received by the SkillGUI and are shown in the log view. Entries
The second tab is the graph visualization view. The DOT graph is read via the SkillerDebugInterface. The Graphviz visualization library is used for the automated layout of the graph. Graphviz itself has a plugin infrastructure, where different layout and drawing components are encapsulated as shared libraries, e.g. for generation of PostScript files. A patch has been contributed to the Graphviz project that extends the plugin mechanism to allow for the registration of plugins at run-time, avoiding an external shared library. This allows for a closer coupling of the application, graphviz and the visualization plugin. The graph is drawn using the Cairo 2D Graphics Library. Graphviz decomposes the graph drawing into simple operations like drawing bezier curves and polygons. It then calls the appropriate callback functions of the plugin. When Graphviz is called, after the layout phase several callbacks of the visualization plugin are called, which create appropriate objects in the scene graph, which in our implementation draws the graph on the screen using Cairo.

Figure 6.4 shows a screenshot of the GUI. The active state is marked by a filled orange node. Double circled octagonal nodes mark exit states. For a skill these nodes are the FINAL (green) and FAILED (red) nodes. The orange bordered nodes and transitions have already been visited during the execution of the skill. The numbers besides orange elements denote the order in which the elements were visited. Dotted transitions lead to the failure state. The thick light orange bordered note is the starting node. Having this visualization is very valuable for developing, testing, and debugging skills. It makes the behavior comprehensible and traceable and easier to understand a partic-

Figure 6.4: SkillGUI: a GUI to execute, visualize and debug skills

---

11Cairo 2D Graphics Library: http://cairographics.org/
ular decision of the robot. The SkillGUI can subscribe as exclusive controller. Then skills can be executed by entering the skill string at the lower part of the window. Skills can be executed continuously or one time and the execution can be aborted at any time. The SkillGUI can release the exclusive control to only monitor the FSMs of skills while an agent is running. The SkillGUI can also visualize the graph of the Lua agent described in Section 6.8.

A visual programming approach for behaviors was not a goal of this thesis, and is probably not feasible in the domain of RoboCup. However, with SHSMs as the modeling approach the SkillGUI could be extended or a new application written to support this.

### 6.7.3 Simulator Integration

Testing on the robot is invaluable. But it is also time consuming and has constraints like battery run-time and availability of an adequate testing environment, i.e. a soccer field of the size stated in the rules and with a useful carpet. Especially the latter has turned out to be a problem for the current NaoQi motion engine, which does not use the sensors for a closed loop walking and does not provide all the parameters necessary to create a sufficiently stable walking behavior on different floors. Therefore, a simulation environment is a valuable tool when developing software for a robot.

We discuss the simulation environment for the Nao robot in more detail below. For the MSL robot no fully working simulation environment has been created to date. However, during this thesis a plugin was written to integrate Fawkes with Player (cf. Section 3.1.6). It connects to a Player server and implements translation modules, that connect Player and Fawkes interfaces. With this plugin and work done for the old robot software framework [100], an integration of the Gazebo 3D [101] or Stage 2D [102] robot simulation environments should be possible with a reasonable amount of work.

From the very beginning Aldebaran provided the commercial Webots 12 mobile robotics simulation software. It was used for first experiments on the Nao platform even before the real robots were delivered. The integration is twofold. A Webots controller integrates into the simulation environment. The naosim plugin is loaded in Fawkes. It connects to the controller via a network connection employing the Fawkes network protocol. A custom component protocol was written that provides all hardware data to the plugin and accepts orders, e.g. for joint movements, from the plugin. Additionally, it is used to synchronize the simulation with the Fawkes main loop. For each simulation loop exactly one iteration of the Fawkes main loop is executed. On a modern PC the simulation for a single robot runs in about real time. A fast forward

---

mode can be enabled that makes the simulation faster, but omits the visualization.

The Webots controller is written in C++ and uses Webots’ API to access the simulated robot hardware and environmental data. It can embed a NaoQi instance to provide advanced features in the simulated environment such as the motion engine. Since NaoQi acquires exclusive control over the simulator once started all commands have to be piped through it causing minor performance degradation. For experiments with the behavior engine this does not pose a major problem. Most of the time we are only interested in walking and specific motion patterns currently implemented on the NaoQi level, like kicking or standing up.

The naosim plugin provides access to the simulated hardware within the Fawkes environment. This plugin was the first module developed specifically for the Nao. Figure 6.5 shows a screenshot of four robots playing in the simulator. For most of our experiments however, only a single robot was simulated, because the simulation of multiple robots on a single machine runs slowly, and setting up a multi-robot scenario with multiple machines is beyond the scope of this thesis. The small images within the screenshot show the robots’ visual perceptions. When running in the simulator the speech synthesis can be provided by plugins that integrate Flite [103] or Festival [104] to have a fully comparable environment, for example to program robot demonstrations. Informational voice output is a useful feature during testing to have another communication channel more natural to humans than reading log files.
6.8 Reactive Lua Soccer Agent for the Nao

An agent program is required to make the robot a useful player in a soccer game. It takes information gathered via the robot’s local perception and received from other robots via the network and makes overall game play decisions, i.e. which skill to call next.

It is possible to implement this as a super-skill, which calls all other skills as the sub-skills. For a better structure of the software and to test the communication between the skiller and an agent, a full Lua agent has been implemented in the \textit{luaagent} plugin. It provides a Lua environment similar to the skiller. Instead of multiple skills, there is always exactly one agent module with the same basic functions as a skill module (cf. Section 6.4.3). It is executed continuously as long as the plugin is loaded.

The implemented agent employs a hierarchical HSM and models a robot player. The HSM has received modest extensions in the \texttt{AgentHSM} class to aid the agent development. A hierarchy of HSMs is build by executing another HSM for a specific state of the parent HSM. To facilitate the development of hierarchical state machines the \texttt{SubFSMJumpState} class has been implemented. Skills can be called using the \texttt{AgentSkillExecJumpState} class, which triggers and monitors the execution of a number of skills by the skiller plugin via the \texttt{SkillerInterface}.

The basic state machine of the agent implements the game process described in [105]. The state machine is shown in Figure 6.6. The graph has been auto-generated from the HSM specification written in Lua, depicted in Listing 6.17. The \texttt{INITIAL}, \texttt{READY}, \texttt{SET}, \texttt{PLAYING}, \texttt{PENALIZED} and \texttt{FINISHED} states are the game states. During the game there is a referee box, a computer with a special program, that sends information about the game and commands. This complements the human referee to communicate with the robots and eliminates the necessity to implement recognition of human referees’ gestures and commands. The referee box is a common concept among different robot soccer leagues. Therefore, we have created an abstraction for game states that differentiate common situations like kick off, normal game play or frozen game (no robot is allowed to move). A repeater communicates with the referee box on the one side, and our team of robots on the other side and translates the league-dependent protocol into our abstract world info protocol. Game states are written to the \texttt{GameStateInterface}. In the graph, there are currently two transitions triggered by this interface, one is labeled “GS Kick Off” and initiates the kick-off behavior and “GS Play” starts normal game play. To speed up testing on the field, game states can be changed by presses of the chest button. In the graph, \texttt{SB} and \texttt{LB} denote short and long button presses respectively (a short button activation is between 0.05 and 0.5 seconds, a long button press is between 0.5 and 1.2 seconds). Keeping the button pressed for more than 1.2 seconds will cause the robot to collapse. This is used as an emergency stop, for
Figure 6.6: Nao soccer Lua agent graph with the sub state machine for the attacker example if the robot is executing potentially harmful movements.

Other states have been added to implement the custom agent logic. The PLAYING state employs another SkillHSM instance for the game play behavior. The chosen HSM depends on the role of the robot, which can be attacker, defender, or goalie. The depicted graph shows the attacker behavior in the grey attacker_play box. Currently, this is a simple reactive behavior to show the general applicability of the approach on the agent level. The HSM is executed while the outer game state HSM is in the PLAYING state. The robot will immediately start searching for the ball, using the search_ball skill. When the ball becomes visible to the robot, it will execute the intercept_ball skill to walk to the ball. If this succeeds, the robot will try to kick the ball. After the ball has been kicked, the robot searches for the ball again. In Section 7.3.3 we discuss the run-time behavior of the agent.
6.8. Reactive Lua Soccer Agent for the Nao

Listing 6.17: Code for the Nao soccer agent

```lua
local np = require("predicates.nao")
local sp = require("predicates.soccer.general")

local ROLE = "attacker"

local playmod = {}
playmod.attacker = require("agents.naosoccer.attacker_play")
playmod.defender = require("agents.naosoccer.defender_play")
playmod.goalie = require("agents.naosoccer.goalie_play")

-- Setup FSM
fsm:add_transitions{
  closure={np=np, sp=sp},
  {"INITIAL", "READY", "np.short_button or sp.gs_kick_off", desc="SB or GS Kick Off"},
  {"READY", "SET", "np.short_button", desc="SB"},
  {"READY", "PLAYING", "sp.gamestate_play", desc="GS Play"},
  {"SET", "PLAYING", "np.short_button", desc="SB"},
  {"PLAYING", "PLAYING", fsm=playmod[ROLE].fsm},
  {"PLAYING", "PENALIZED", "np.short_button", desc="SB"},
  {"PENALIZED", "PLAYING", "np.short_button", desc="SB"},
  {"PLAYING", "FINISHED", "np.long_button", desc="LB"},
  {"PENALIZED", "FINISHED", "np.long_button", desc="LB"},
  {"FINISHED", "INITIAL", "np.long_button", desc="LB"}
}

-- Below is the attacker HSM from agents.naosoccer.attacker_play
fsm = AgentHSM:new(name="attacker_play", start="SEARCHFOR")

local function kick_params(state)
  local leg = "left"
  if wm_ball:bearing() < 0.0 then leg = "right" end
  state.args = { kick={leg=leg} }
end

fsm:add_transitions{
  {"SEARCHFOR", "INTERCEPT", fail_to="SEARCHFOR",
    skills=({"search_ball", (no_turn=true)})},
  {"INTERCEPT", "KICK", skills=({"intercept_ball"})},
  {"KICK", "SEARCHFOR", skills=({"kick"})},
  {"SEARCHFOR", init=kick_params},
}
```

94
Chapter 7

Evaluation

In this section we present results that have been acquired during the evaluation phase of this thesis. First we explain the particular challenges in Section 7.1. Then we evaluate the Fawkes RSF in Section 7.2 and finally the behavior engine in Section 7.3.

7.1 Evaluation Challenges

As this thesis covers only the base framework and the behavior engine, we rely on other parties for components like navigation, motion and localization. As described in Section 4.3 the localization for the Nao is developed in Austria and the motion component in South Africa. For the localization there is a prototype that is lacking documentation and is not considered to be stable enough for actual game play by its developer. The motion engine is in a very early prototyping stage. The NaoQi motion engine produces acceptable results on certain floors only, and the available green carpet in the lobby of the computer science building E2 of the RWTH Aachen is not such a surface, because it is too soft and a rolling motion of the feet ankles cannot be configured. On the MSL robots the localization is in a later development stage but it still does not provide the required robustness and timing. The motion component of the navigator also has problems with the given carpet. More development and optimization are required to achieve the desired speed and robustness.

To cope with these challenges we have adapted our plan and environment. We rely on local relative perception, meaning that the data is not transformed into the global soccer field coordinate system, eliminating the need for the localization component. The Nao experiments have been carried out in a regular office with a suitable floor for the NaoQi motion engine. For the MSL robot we developed a goalie skill, that does not require lengthy navigation operations. We have developed components for ball recognition and integration of the NaoQi motion engine in large parts during this thesis.
7.2 Fawkes Robot Software Framework

In Section 2.3.2 we have presented several characteristics of robot software frameworks (RSFs). In Section 7.2.1 we discuss how these characteristics apply to the Fawkes RSF as a qualitative analysis and then describe performance measurements that have been conducted for basic CPU and memory assessment in Section 7.2.2 and run-time analysis in Section 7.2.3.

7.2.1 Fawkes Software Characteristics

Fawkes is running on three different platforms, the Nao SPL robot, the MSL robot and our RoboCup@Home League robot. The hardware abstraction is accomplished by encapsulating different hardware components into separate plugins. For example for the MSL platform we provide a plugin for accessing the motor, the batteries and the kicking device. Data is shared and commands are sent via the blackboard. Camera access is provided by the FireVision framework within Fawkes, that supports a variety of cameras which are all accessed via an unified interface, buffers are converted to a unified format by default, if not explicitly disabled. Therefore, adapting to a new platform is done by implementing and loading the set of plugins that matches the chosen hardware. On the SPL and MSL robots Fawkes is the exclusive software system. It is able to drive all the hardware available on the robot.

Extensibility is accomplished by a component-based approach. Different functional blocks are encapsulated in plugins and interconnected via the blackboard. This makes it easy to use existing components and add new ones. For Fawkes scalability is especially targeted towards singular machines. In the primary application domains all of the computation happens on the robot. Since Fawkes makes use of multi-threading at its core, there is a high potential for exploiting today’s multi-core machines, although one has to be careful because of memory throughput constrains discussed later. For instance, on the MSL robot multiple vision processes run at the same time, without any explicit programming. However, on the Nao platform the RSF must perform well in a single-threaded environment. Components can be distributed over multiple hosts via remote blackboard access over the network.

The run-time system itself has been written with a low computational overhead in mind. Conditional checks, for instance to meet the soft guarantees, are evaluated only when they are first encountered and not in every cycle. When no threads are hooked into the main loop, it will be paused until such a thread is loaded. This avoids unnecessary CPU wake-ups and conserves energy, an important factor on a battery powered robot.

The actuator control model that is implemented by a plugin can be freely chosen. For the overall framework the hybrid deliberative-reactive paradigm is used. This is reflected for example by our blackboard architecture for infor-
mation sharing and the way the standard main loop is composed. Especially in the behavior engine and the simple agent we implemented this paradigm.

The Fawkes RSF is complete in the sense that it provides the basic infrastructure to implement and interconnect a set of components to control a certain robot. For a particular platform and domain the applicable components must be developed and integrated. We have made an effort to make it as simple as possible to integrate new functionality as plugins. Especially the aspects are a useful tool to expose framework functionality without cluttering the code of a plugin. The correctness is not automatically verified for example by using unit tests. For several parts of the software quality assurance applications have been written, that can be used to manually test parts of the software. To achieve consistency functional blocks have been bundled into appropriate libraries. The “worse is better” paradigm [19] has been applied, for example when integrating the NaoQi motion engine to be able to walk, although in a quite unstable fashion, until our own motion engine is ready. This might not provide the best results for locomotion, but at least it does provide some results.

Tool support is very important when developing software in general, and for complex systems like a robot in particular. We have carefully designed the software to interact well with debuggers and performance analysis tools. The software can be compiled with the GNU C++ Compiler and the Intel C++ Compiler on Linux and FreeBSD. Although these systems are similar, using multiple tools and operating systems has revealed several cases where modifications result in better and faster code more compliant to standards like POSIX. To improve the code quality we have setup a continuous integration process [106] based on Trac\(^1\) and Bitten\(^2\). For each new revision committed to the Subversion\(^3\) repository the whole software is built and documentation generated. Failures or missing documentation trigger an email sent to all developers.

The framework provides tools for inspection of all data that has been stored in the blackboard by means of a web interface. It is currently lacking a powerful desktop application that can monitor and graph the data in real-time. The run-time environment constantly monitors the execution time of plugin threads and can cope with threads that run for too long and might slow down the overall software system.

Documentation is provided for several different audiences. Documentation of all public APIs is enforced in the continuous integration process. A missing or incomplete documentation is regarded as an error similar to a failure to compile the software. This way a good documentation coverage of the base framework has been achieved. A wiki provides documentation how to setup Fawkes and start developing. It is extended as the development goes on.

\(^1\)Trac software project management: http://trac.edgewall.org
\(^2\)Bitten continuous integration Trac plugin: http://bitten.edgewall.org
\(^3\)Subversion version control system: http://subversion.tigris.org
7.2.2 CPU and Memory Assessment

For the interpretation of the run-time evaluation results we need to investigate the basic capabilities of the robot platforms in terms of CPU speed and memory throughput. Robot programs and algorithms have similar requirements as tasks accomplished in high performance computing (HPC). In HPC, hybrid systems that exploit shared memory parallelization on multi-core hosts and distributed memory programming across multiple machines at the same time become more common with the rise of modern multi-core CPUs [107]. With these CPUs being available even in mobile computers, this aspect of modern HPC can be applied to robotics software. On the robot, many calculation and modeling techniques to implement sensor processing and decision making are implemented similarly to common HPC problems, although much smaller in scale. For HPC, computation is limited not only by the speed of the CPU, but also by the memory bandwidth and latency. We have conducted benchmarks to evaluate the capabilities of the MSL and Nao computing systems described in Section 2.1.2.

Throughput of MSL and Nao Robot Computers

The memory of the computing system is divided into several stages, the main memory (RAM) and several levels of cache within the CPU. The examined CPUs have two levels of cache, called Level 1 (L1) and Level 2 (L2). L1 cache is quite small and divided into instruction and data cache. With respect to computational speed the L2 cache is more relevant. The AMD Geode in the Nao has only 128 KB, while the Intel Core 2 Duo of the MSL robot has 4 MB (shared by both cores). We have applied a benchmark\(^4\) to test the throughput of the different memory types. The averaged results of 10 runs on both, the MSL and the Nao platform, are shown in Figure 7.3(a). All benchmarks have been performed with 32-bit code, even on the Core 2 Duo. With the exception of the AMD Geode RAM, for a certain level reading from memory is faster than writing to it. The speed decreases slightly from L1 to L2 cache, and then falls off for the RAM. As to be expected the Core 2 Duo has a much higher bandwidth which peaks at about 7.5 GB/sec for reading from L1 cache, and at about 5.6 GB/sec for L2. From the RAM it can still transfer about 2 GB/sec. The AMD Geode maximum throughput is about 1.7 GB/sec for L1 and 1.5 GB/sec for L2, and drops to only 240 MB/sec for reading from RAM.

CPU Speed of MSL and Nao Robot Computers

To measure the CPU speed an artificial CPU benchmark that implements a matrix-vector multiplication has been used. It implements the calculation of

\(^4\)Bandwidth benchmark: http://home.comcast.net/~fbui/bandwidth.html
\textbf{Lstg. 7.1: Row Matrix-Vector Multip.}

\begin{verbatim}
#pragma omp parallel for private(i, j)
    for (i = 0; i < m; i++) {
        a[i] = 0.0;
        for (j = 0; j < n; j++)
            a[i] += b[i*n + j] * c[j];
    }
\end{verbatim}

\textbf{Lstg. 7.2: Column Matrix-Vector Multip.}

\begin{verbatim}
#pragma omp parallel for private(i)
    for (i = 0; i < m; i++)
        a[i] = b[i*n] * c[0];
    for (j = 1; j < n; j++)
        #pragma omp parallel for private(i)
            for (i = 0; i < m; i++)
                a[i] += b[i*n + j] * c[j];
\end{verbatim}

$A = B \cdot C$, where $B$ is a Matrix of dimensions $m \times n$, and $A$ is a vector of size $m$ and $C$ a vector of size $n$. The multiplication is executed with different values for $m$ and $n$ ranging from $100 \times 100$ to $2000 \times 2000$. However, we are not so much interested in the dimensions of the matrix, but rather in the size of the memory it takes to store this matrix, ranging from 0.08 MB to 32 MB. We have implemented two variants of the multiplication. The first multiplies each row of the matrix $B$ with $C$, allowing for a sequential access to the matrix memory. The second multiplies each value of a column of $B$ with the appropriate element in $C$. This requires access to values spread throughout the whole chunk of memory. The implementations are shown in Listings 7.1 and 7.2. It has additional code to support parallelization via OpenMP.

Figure 7.3(b) shows the results given in million floating point operations per second (MFlops) with double precision. The OpenMP variant has been omitted for the Nao platform, because without any hardware support for parallel execution OpenMP at best provides the same performance and usually even decreases it slightly. As expected, the MSL computer is more than an order of magnitude faster than the Nao’s with peaks at about 1200 MFlops for a single core and about 2400 MFlops for two cores. For the MSL robot we clearly see that the performance is a lot higher as long as the matrix memory footprint is smaller than the available amount of cache. In this situation, the CPU does not have to wait for new data to process from the much slower RAM. In this area of up to 4 MB the computing performance doubles with the usage of OpenMP that distributes the calculation over two threads. As soon as the matrix does not fit into the cache, the performance drops and the parallelization gives only a speedup of about 10\% to 20\%. This is probably due to false sharing [108]. The two separate threads overwrite each others entries in the cache, causing a performance penalty. The column-wise multiplication gives a much worse performance. This is because matrix accesses are distributed over the whole memory in a each loop iteration. Even for small matrices that fit into the cache the performance is bad, because with a cache line of 64 bytes only a fraction of the bytes is used, before the next line already needs to be transferred from L2 to L1 cache. If the problem size is larger than the available cache, basically all data must be transferred from and to the RAM during the calculation.
7.2. Fawkes Robot Software Framework

On the Nao, the row-based implementation has an almost constant speed of about 40 MFlops for any matrix size. This indicates that the memory throughput is high enough to keep the CPU busy all of the time as long as the memory can be read optimally. This changes drastically for the column-based implementation. For the smallest matrix the speed is comparable, as the problem of 0.08 MB still fits into the L2 cache of 128 KB. But for larger matrices the computation speeds drop to 10 MFlops and below. Here we see that the bad throughput from the CPU to the RAM causes the computation units of the CPU to wait for new data most of the time.

7.2.3 Run-Time Measurement on MSL and Nao Platforms

An important metric for the evaluation of a computer program is the time it takes to accomplish a certain task. For the contemplated robot platforms and the soccer domain we are interested in the main loop frequency that can be achieved while still maintaining an acceptable quality of the environment perception, decision making and action execution. In the MSL, robots are easily moving with 3 m/s. If the main loop would run at only 10 Hz between two perceptions a robot would have moved 30 cm. A ball that is several times faster than this when shot has moved even further. In the robotic soccer domains the achievable frequency is usually determined by the speed of the main sensor. On the MSL and Nao platforms, these are the main cameras. At the maximum frame size they can provide images at 30 frames per second. Therefore we aim at an overall main loop frequency of 30 Hz. Locomotion components usually need to run faster than this to control the motors and joints. They are therefore split off in separate continuous threads, which run concurrent to the main loop at a higher frequency of 50 Hz. In the case of the Nao this thread runs in the NaoQi software at this time.

Data Acquisition Method

We have written a plugin that replaces the standard main loop (cf. Section 4.2.3) with a loop that executes the threads in the same way, but measures the time that is used for the execution of the different hooks. After two seconds of runtime the loop times are averaged and saved to a log file. This is done to minimize the influence of the writing of the log file on the results.

During testing, both robots ran a ball perception vision module, the world model, the skiller and luaagent plugins, and a navigation component. The MSL robot additionally ran an obstacle detection vision module, for collision avoidance on the field. The Nao robot only has a very simple navigation component, that does not provide obstacle avoidance at this time. With this set of plugins there is generally one thread per hook, with the exception of the MSL image processing, were two threads are running. Additionally, the NaoQi soft-
**Figure 7.1:** Timing of the Main Loop Hooks of the Nao SPL Robot

**Figure 7.2:** Timing of the Main Loop Hooks of the MSL Robot

**Figure 7.3:** CPU and Memory Benchmarks for MSL and Nao Robot Platforms
ware was running on the Nao, to provide access to the hardware and to use
the motion engine. NaoQi by itself uses about 20% of the CPU constantly at all
times, no matter if the robot is moving or standing still. On neither robot the
localization component was used. The components for both robots are work in
progress in separate diploma theses. At this time they are in a prototype stage
and are not optimized for efficiency. For instance the MSL localization has an
average run-time per loop of about 180 ms. Because this is just an intermediate
state we omitted these components from the measurement as not to influence
the other results. For the Nao the Lua agent described in Section 6.8 was run.
On the MSL robot a simple test agent was run that executed test skills, but
movements were executed manually.

Run-Time of Fawkes and its Plugins on the Robots

The results for a test period of about 6 min are shown in Figures 7.1 and 7.2.
They show the run-time per hook in differently colored areas. The horizontal
axes are the timeline for the tests, the vertical axes give the run-time per loop
in seconds. The times spent per hook are stacked onto each other. The ver-
tical extent of a colored region therefore gives the run-time of the particular
hook and the all areas combined give the total run-time for the hooks. The
purple lines marks the total run-time per loop as recorded by the framework,
including its own overhead.

As a first result we see that the total time spent for the hooks and the total
time recorded by the framework are about the same. If the run-time coordi-
nation of the framework would consume a lot of time by itself, we would see
a white area between the accumulated run-time of the hooks and the purple
line. The fact that we do not see this white area most of the time indicates that
the run-time coordination causes only a minimum small overhead.

Image Processing. Sensor processing often is a dominant factor for the overall
run-time of a loop iteration. On the examined platforms this means to analyse
images with computer vision methods. The large blue areas in the Nao graph
show this effect. Images require large chunks of memory to hold the images.
The current ball detection plugin evaluates pixels at the crossing nodes of a grid
with a certain cell size. It classifies the grid pixels to belong to a certain object
by retrieving a value from a color look-up table that has been learned. Orange
pixels for example are classified as ball pixels. Several adjacent pixels of the
same classification are grouped. One out of possibly many groups is chosen by
certain criteria and assumed to contain the ball. A position is estimated with a
position model.

This procedure has drawbacks from a memory throughput point of view.
Two different chunks of memory need to be accessed, the image and the look-
up table, and only a few bytes are required per cache line because several pix-
els are skipped due to the grid. Irritatingly the smallest images that the Nao
hardware can provide are 320 × 240 pixels in size encoded as YUV 4:2:2. Images with these dimensions are too large to fit into the L2 cache of 128 KB. An average run-time of 65 ms per frame was achieved using this image size. Based on this observation, we reduced the resolution in software to 160 × 120 pixels and reduced the grid cell size. Although about the same amount of pixels were analyzed the time was reduced to about 30 ms. With the modification the image and the look-up table both fit into the L2 cache. The number of processed pixels could even be increased with only a small addition in time, but the sparse grid provided results good enough for soccer playing.

On the MSL robot the omni vision camera operates on images of 1000 × 1000 pixels with a size of about 2 MB. These images still comfortably fit into the L2 cache including the color look-up table. The mirror calibration look-up table which maps image pixels to real-world coordinates has a size of 8 MB. But it is used only for a very small set of points and therefore does not harm the performance. The two vision plugins for ball and obstacle detection share the same input image, therefore avoiding false sharing as long as no large internal buffers are used by the plugins. This is the case for the two vision modules with disabled debugging options.

At a first glance the vision seems to take less time when the navigation (red) requires computing power. This caused by the camera, which can deliver images only at rate of 30 frames per second. Because the operating system blocks the read operation until a frame is available, the vision plugins are waiting most of the time for new data to arrive. The two plugins combined require about 10 ms. As seen in the graph after about 210 s the total run-time increases over the average because the navigator demands more computing power while the vision has already reached its minimum.

Motion. For the MSL robot much of the computation power is used by the navigation component (red). When driving it requires about 20 ms of run-time. When the robot is standing still the time drops to almost zero. The chart shows that the navigation algorithm requires optimization. The localization will probably take more than 10 ms and the navigator and vision cannot run in parallel (the navigator needs input from the vision for obstacle avoidance). Thus the sum of both components’ run-times must fit into the desired loop time of about 33 ms. Part of this optimization could be to cluster obstacles to reduce the number of objects the obstacle avoidance has to consider.

On the Nao currently there is no obstacle avoidance. Therefore the navigation itself only determines a series of appropriate turn and walk operations to reach the destination. The actual execution of these simple motion patterns currently happens in the NaoQi motion engine, and is therefore not shown in the chart and the actuator hook seems to consume no time. NaoQi has a constant CPU utilization and there is no way to determine what the time is spent for, be it communication with the hardware or motion generation.
World Model. The run-time of the *world state* hook, which runs the world model thread, is negligible at the moment. At this time it copies blackboard interfaces and asynchronously transmits data over the network. When more capable ways of integrating multiple data sources are implemented this will probably change.

Skiller and Agent. The skiller and Lua agent threads only require a limited amount of run-time. On the MSL robot this time is negligible and on the Nao the agent and skiller require a run-time of about 1 ms and 2 ms respectively. The peak in the Nao chart (Figure 7.1) after about 160 s is caused by a modification of a Lua script causing an automated reload of the scripting environment. The overall run-time of the Lua-based plugins could possibly further decreased by thoroughly benchmarking the different parts of the Lua system. But given the ratio of the Lua-based plugins’ run-time to the other components optimizing the other plugins looks more promising. If automatic graph generation is enabled the run-time increases noticeably by about 10 ms on the Nao. This is caused because large data structures of the FSMs need to be traversed and a lot of text processing happens that is quite expensive and memory intensive. Therefore graph generation was disabled while measuring the time and will be disabled during real games.

Compared to other tasks the threads perform simple operations like checking jump conditions and executing transitions. But not many calculations or memory operations are performed. The run-time of the threads directly depends on the run-time of the implemented agent or skills. But since skills and Lua agents are envisioned as reactive execution entities, we do not expect a major increase in their run-time.

By optimizing the Lua environment the run-time has been reduced to a third of the original time. The skill string is compiled only when it changes and the result is stored and used for multiple iterations. The skiller has a longer run-time because it usually executes more transitions than the agent and the infrastructure is more complex, because the agent only runs a single instance while the skiller can execute multiple skills at the same time.

Summary. In summary the graphs show that the desired main loop frequency of 30 Hz can be reached on the MSL platform and almost be reached on the Nao platform. However, we have reduced the frequency on the Nao to 20 Hz to cooperate with NaoQi. If Fawkes consumes too much CPU time the motion engine for example cannot interact fast enough with the robot causing erratic behavior. On both platforms the localization systems need stabilization and optimization before reliable numbers can be produced. The Lua-based skiller and agent plugins operate fast enough with the current set of skills and the current agent programs. Optimization is required in other areas like image processing and navigation. This makes Fawkes a RSF applicable to both platforms, the multi-core MSL robot as well as the constrained Nao platform.
Chapter 7. Evaluation

<table>
<thead>
<tr>
<th>Skill</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto</td>
<td>Go to a position ((x, y, \theta)) given in global field coordinates.</td>
</tr>
<tr>
<td>relgoto</td>
<td>Go to a position ((x, y, \theta)) or ((d, \phi, \theta)) given in cartesian or polar coordinates relative to the robot.</td>
</tr>
<tr>
<td>turn</td>
<td>Turn by an angle (\theta) around vertical center axis of the robot.</td>
</tr>
<tr>
<td>transrot</td>
<td>Skill for wheeled robot to set translation and rotation values</td>
</tr>
<tr>
<td>say</td>
<td>Use speech synthesis to make the robot talk.</td>
</tr>
<tr>
<td>servo</td>
<td>Nao specific skill to control the servos, especially to power them on or off.</td>
</tr>
<tr>
<td>standup</td>
<td>Skill for humanoid robots to stand up when sitting or lying.</td>
</tr>
<tr>
<td>search_ball</td>
<td>Nao specific skill to search for the ball by moving the head and turning around (cf. Section 6.6.1).</td>
</tr>
<tr>
<td>intercept_ball</td>
<td>Skill to move the robot close to the ball.</td>
</tr>
<tr>
<td>goalie</td>
<td>MSL robot specific skill that implements a simple goalie behavior (cf. Section 6.6.2).</td>
</tr>
</tbody>
</table>

Table 7.1: Selected skills implemented for the Nao and MSL robots

7.3 Lua-Based Behavior Engine

A major goal of this thesis is to develop a behavior engine which is applicable to multiple platforms and domains. For the evaluation we have chosen two different platforms, the MSL and SPL robots, operating in a common domain, robot soccer. We have done so to evaluate the general applicability on different platforms on one side and explore the possibilities for code sharing and convergence of code for a common domain on the other side. During the development we have concentrated on the Nao platform and used the MSL platform as a reference and secondary evaluation platform. In the previous section we described the applicability of the Fawkes RSF as a base for the target platforms. It forms the foundation for the behavior engine.

7.3.1 Skills for the Nao and MSL Robots

For both platforms a set of skills has been developed that allows for basic autonomous soccer playing of the robots. On the Nao robot, we concentrated on the field player that walks to the ball and kicks it, on the MSL platform we developed a goalie skill and there is an ongoing effort to create a Lua-based
agent for the MSL. The rules for the MSL however are more complex and thus the MSL agent is more comprehensive and was not developed during the time frame of this thesis.

Table 7.1 shows a selection of skills that have been developed during this thesis (see Listings 6.9 and 6.10 for the definition of all available skills). The first seven are basic skills, directly using sensor data and ordering execution of basic actions in the framework. The relgoto, turn and transrot skills send appropriate commands to the navigation component. The goto skill makes use of the global position estimate for the robot and then use the relgoto skill to go to the desired position. The servo skill uses the NaoQi integration to move the servos. The standup skill causes the robot to stand up using the NaoQi motion engine via the abstract humanoid motion interface, which is implemented by the NaoQi motion engine integrator at the moment. The search_ball skill is currently only implemented for the Nao. It uses a combination of servo and turn skill calls to move the robot such that the vision can detect the ball. The intercept_ball has the precondition that the ball must be visible, otherwise it fails immediately. It uses the relgoto skill to go close to the ball. Later the skill will be extended to adapt the target location while walking towards the ball, to cope with a moving ball and because the vision can provide better results when closer to the ball. At the moment this is not possible, because due to an error in the NaoQi software the head cannot be moved while the robot is walking, and therefore the ball cannot be focused all the time. This problem should be fixed in the upcoming NaoQi release.

The skills currently rely on a complaisant environment as they do not cover many error conditions. Making a skill robust in many situations requires a immense amount of time on the soccer field and experience of running them at a RoboCup event. Nevertheless the skills worked reliably in our tests in the office for the Nao and on the soccer field in the lobby for the MSL robot. For our tests the full system with all components was running for the first time. Problems that stem from the integration of the components have to be solved and optimizations applied to computationally expensive components.

An important result is the brevity of the specification of the behavior when using the modeling approach with HSMs. At the same time we managed to maintain the freedom for very complex skills, either using the extended HSM API, or by implementing the basic skill interface (cf. Section 6.4.3) and employing another modeling approach. With this combination the behavior engine can evolve and new modeling approaches could be tried if this becomes necessary.

The modeling approach using Hybrid State Machines does work well for skills. Especially the jump conditions are a useful tool to model conditions for the change of the robot’s behavior. Features like invariant and flow conditions have not been used. Flow conditions could be used to model for example the movement of the ball when it is occluded by other robots. But it is more likely
that this kind of functionality will be implemented directly in the world model, such that it is available to all components. We did not exploit the multi graph capabilities of the modeling approach, that could be used to model multi-agent behavior. This however is probably more useful on an agent level, and not for skills. Two agents could coordinate and one agent runs a pass skill while the other executes a behavior to receive the pass. The extension of the hybrid state machines by skill dependencies makes it easy to detect problems at runtime. Encapsulating a skill by an execution function provides a very efficient way to call one skill by another and hide all the execution specialties of the skill, allowing the user to concentrate on the skill’s semantics. Additionally, it provides a convenient interface for agent programs to call skills.

7.3.2 Code Sharing Possibilities and Limits

One of the benefits we expect from a common behavior engine for the different platforms is the ability to share code. Ideally, we want to build up a repository of behaviors to choose from for the particular platform and domain. The code that implements the behavior engine is already a shared code base that hides a lot of differences when operating several platforms. But we also want to explore the possibilities to use the same skill on multiple platforms or in different domains.

For the skills developed during this thesis we see that there is an overlap especially for basic skills like locomotion and ball interception. However, many skills are intrinsically tied to a particular hardware platform, or a group of platforms. Several skills implemented for the Nao robot assume a humanoid platform with specific properties. This ranges from skills like the servo skill that is specific to the Nao platform to skills like standup or park, that could be applied to other humanoid robots as well. Although both, the Nao and the MSL robot, operate in the same domain – robotic soccer – even basic soccer skills like searching for the ball are different. On the MSL platform with the omni-vision system the ball is almost always visible, as long as it is in sensor range. There is no need for the robot to turn if the ball is not visible, rather it has to drive to different positions on the field to look for the ball. For the Nao robot, with a directed camera, to find the ball the head or the full robot must move to look in different distances and angles. On the current small soccer field it is unlikely that a roaming behavior as described above for the MSL robot is required on the Nao in the near future. But even for these diametric platforms with different ways of locomotion we implemented a common ball interception skill. With the precondition that the ball is already visible it is independent of the particular method used to search for the ball. Given the relgoto skill it can simply move to the ball, possibly adjusting the final desired position while on the way. For a better behavior it might become necessary
to develop separate intercept skills, that account for particular features and deficiencies of the platforms.

An important observation is that skills can be made platform independent more easily if the underlying interface to the base system is well defined. For instance in terms of a general goto skill a navigation interface exists that defines a set of basic platform independent motion commands, like “go to the position \((x, y)\) relative to the current position”. Therefore it is beneficial to carefully design the component interfaces. Abstraction on the interface level to a certain degree can improve the platform (and possibly domain) independence of skills and chances are that skills can be used for multiple platforms.

We have done some tests in a very early stage of the development in the RoboCup@Home League. From that experience we expect that different domains will cause more divergence in the set of skills and thus in the code base. This has to be investigated in more detail as the framework is applied to the service robot.

This thesis concentrates on the creation of a robust and reliable framework to run a robot and execute a robot’s behavior. With more advanced skills in the future, like a soccer skill “attack over the wing” that would dribble the ball over the wing to close to the goal, and then pass to another robot in the center, we see more potential for code sharing. On this level the hardware dependent basis is hidden in basic skills and more abstract behaviors can be created, much like well-defined component interfaces allow for more general base skills.

### 7.3.3 Tests with the Nao Soccer Agent

We have developed a simple reactive soccer agent for the Nao. It is used to evaluate the integration of the skiller and agent in the software framework, the execution of several skills in combination, the run-time behavior, and the communication between an agent and the skiller. The agent is described in Section 6.8. For the MSL robot a goalie skill (cf. Section 6.6.2) has been developed in short time during the preparation of the qualification material for RoboCup 2009.

Figure 7.4 shows a run of the Nao soccer agent in the office. On the left-hand side is a picture from a video (see figure caption for the URL of the full video) and on the right-hand side is the relevant excerpt from the agent graph (shown in full in Figure 6.6, source code in Listing 6.17). It shows the sub-HSM for the attacker’s active game play. The node filled orange is the active state. Orange edges are transitions that have been followed. The numbers next to these edges indicate the order in which the transitions were executed. The graph layout was generated automatically and saved using Graphviz and the SkillGUI (cf. Section 6.7.2).

---

5MSL Qualification Video: [http://robocup.rwth-aachen.de/msl/quali2009](http://robocup.rwth-aachen.de/msl/quali2009)
(a) Nao initially looks for the ball by moving its head.

(b) Nao intercepts the ball by walking close to it.

(c) Nao kicks but misses the ball. It is not close enough (imprecise ball recognition).

(d) Nao adjusts by intercepting again and walking closer.

(e) Nao kicks again, now with the right leg, and hits the ball.

Figure 7.4: Nao test run. Graph matches the situation in the image. Full video available at http://www.niemueller.de/uni/thesis/naosoccer.avi.
In the \texttt{SEARCHFOR} state the agent orders the execution of the \texttt{search\_ball} skill. This will move the head of the robot to find the ball (cf. Section 6.6.1). When found, the HSM changes state to the \texttt{INTERCEPT} state, which executes the \texttt{intercept\_ball} skill to get close to the ball. If the ball is visible and close, the skill will succeed and a transition to the \texttt{KICK} state will be made. The \texttt{kick} skill is executed using the motion engine of the humanoid robot and then the state is changed again to \texttt{SEARCHFOR} to start over. The outer state machine is only used to get into the playing mode. First the robot searches for the ball by moving its head (a) and optionally turning. Once the ball is found, the Nao intercepts the ball by walking close to the ball (b) and tries to kick it (c). In this case, due to an imprecise value from the vision, the robot is too far away from the ball and misses it. The robot notices this and walks closer to the ball (d) and hits it on the next kick (e). The test has been carried out multiple times. The major problem encountered during this time is the unstable motion engine. Especially when kicking, the robot has the tendency to fall because it does not use the inertial measurement for keeping its balance. The state machine has later been extended to cover this case and making the robot stand up when lying on the ground to continue with the game.

The communication is encapsulated for the agent in an utility class. Passed an array of skills and parameters a skill string is formed and ordered for execution by the skiller. The skill status is monitored to execute transitions based upon failure or success of the skills. We execute only basic skills and do not form more complex Lua strings. This is the expected use case for most applications as the agent probably wants to closely monitor the skill execution.

The skiller and agent plugins will be extended, used and tested even more thoroughly at the German Open 2009 competition held in late April in Hannover. There, the complete systems for both, the Nao robot and the MSL platform, will operate under competition conditions for the very first time.

We have described a selection of implemented skills and the executed tests. The behavior engine is capable of controlling the Nao as well as the MSL robot. There is a potential for code sharing, which is improved for base skills, if the base system provides its functionality via well-defined interfaces with a certain level of abstraction from the platform. A test agent has been written for the Nao, that is able to execute a basis for a soccer playing behavior. A goalie behavior was developed for the MSL robot that showed that the system can be adapted in a short time frame.
Chapter 8

Summary and Future Work

In this chapter we conclude the thesis and give a summary emphasizing our contributions in Section 8.1. We will give an outlook to possible future work in Section 8.2.

8.1 Summary

In this thesis we have designed and implemented a behavior engine that can be employed for multiple platforms and domains. The used platforms are the humanoid Nao and the wheeled Middle Size League (MSL) soccer robots. The Fawkes robot software framework provides a unified base for both platforms. The software was implemented for the MSL robot and has been adapted to the specific constraints of the Nao platform during this thesis. On top of this a common behavior engine for reactive behavior execution has been developed, employing the Lua scripting language and using an approach based on hybrid automata for modeling these behaviors.

Fawkes on the Nao. As the first step Fawkes was integrated with the simulator for the Nao robots and interfaces were defined to interact with the Nao hardware through the Fawkes system. Later Fawkes was integrated with the NaoQi software to access the real Nao hardware and make use of advanced features like the motion engine and image acquisition. Since the custom motion engine from the University of Cape Town is not ready, yet, the NaoQi motion engine was used to control the robot and we implemented advanced motions like kicking and standing up when the robot is lying on its back. However, no sensory input is used, yet, to stabilize these behaviors. The ball vision was ported from our old MSL robots and adapted to the constrained Nao computing platform. A noticeable speedup could be reached by considering the robot’s processing and memory constraints and reducing the amount of data and simplifying the processing algorithms. Finally the behavior engine
8.1. Summary

could be developed as a Fawkes plugin, interacting with the base framework through the blackboard shared memory and messaging system. The software framework by itself uses only a very small amount of system resources, making it suitable for the Nao platform.

Fawkes now provides a robust base system that is applicable to different robot platforms, in particular for the humanoid robot Nao and the wheeled MSL robot. This provides the base on which the behavior engine can be based.

Lua-based Behavior Engine. We have implemented the behavior engine as a middle layer between the lower level system and the high level agent. It executes basic reactive behaviors called skills, which fulfill a specific task without changing the overall strategic goal. This is defined by the agent program on the higher level. Especially with planning in mind, skills are the primitive actions to the agent. We have extended the formalism of hybrid automata to allow for an concise specification of skills. Therefore we introduced dependencies of skills and defined an execution function which encapsulates the whole extended hybrid state machine (HSM) of the skill as a simple function call. With these additions hierarchical structures can be built with skill calling other skills to fulfill their task. For example, the ball interception skill employs the go-to skill to reach a position close to the ball.

The behavior engine uses the Lua scripting language to create the environment in which skills are executed and to implement skills. We exploit advanced Lua techniques to support efficient skill execution and make the programming environment easy to learn. We have implemented a set of skills for both platforms to move the robot and execute simple tasks related to robotic soccer. An agent program was written for the Nao, based on the Lua environment for skills. It uses the same formalism of extended HSMs to model an agent. There we chose hierarchical structures by stacking multiple HSMs, as the agent is a larger execution entity that does not change at the rate of a skill. The implemented agent program supports basic soccer playing based on local perception information. The robot searches for the ball, walks close to it and kicks it.

A benefit of using the same behavior engine on both platforms is the ability to share code. For one the code of the base platform and behavior engine itself, and for the other also in terms of skills. We have seen that well defined interfaces to the lower level system makes it easier to use the same skills for multiple platforms. We expect more complex skills for the same domain to share more code because the simple skills act as another level of abstraction.

Summary We have demonstrated that the Fawkes robot software framework is capable of driving the humanoid robot Nao and the MSL robot. It operates with the required computational and memory efficiency to work on the con-
strained Nao platform as well as on the MSL robot with a multi-core CPU and faster memory. The behavior engine has been applied in both, the Nao and MSL robot soccer leagues. It provides a unified programming environment and should allow behavior designers to switch between different platforms more easily. Additionally code can be shared to a certain degree among platforms and domains. The systems will operate under tournament conditions soon at the German Open 2009 in Hannover.

8.2 Future Work

In the near future several components need to be finished, integrated and optimized. Most notable these are the localization, motion and navigation plugins for both contemplated platforms. The localization plugins for both platforms currently exist in a prototyping stage, but they both need optimizations to use the available resources on the robot more efficiently. The motion engine for the Nao does basically not exist, yet. A lot of work needs to be put into this component to support robust walking and other motion patterns, using sensory input to keep the robot in balance.

The framework could be applied to new robot platforms, real or simulated. The AllemaniACs’ RoboCup@Home robot currently uses the old software system. First components have already been moved to Fawkes, but several are still to be transferred. Starting to import these could result in a lower entry barrier for newcomers as there is only one unified software system to be learned. The framework is currently lacking a thorough integration with simulators other than Webots for the Nao. Adding support for other simulators, for example Gazebo, would make it possible to add a simulation environment for the MSL robots, and to extend the framework more easily for different platforms. It could also make it easier to introduce new students to the robots as they could first get familiar with the system in the simulator before using the real robot.

More skills and more elaborated agents need to be developed using the current modeling approach to support a more complex soccer behavior. Currently the implemented skills depend on a complaisant environment. This needs to be addressed to allow for a faster and more reactive and robust game play. Additionally the modeling of skills by HSMs could be extended with multi-agent capabilities, by using the multi-graph and event capabilities of the underlying formalism. Because of the general nature of the behavior engine implementation, different modeling approaches could be tried using the existing base system.

Agents employing planning approaches could be written. For example the ReadyLog [109] deliberative programming environment, which utilizes decision theoretic planning for real-time systems, could be integrated into the
Fawkes robot software framework. For this scenario skills are the primitive actions used by the planner. The communication between the behavior engine and the agent has been implemented and tested with the reactive soccer agent written in Lua. Therefore the infrastructure is prepared for deliberative agents. The luaagent plugin could be extended, to import agent programs written in XABSL. Since the XABSL and our model for behaviors should be compatible, XABSL option graphs could be mapped to HSMs. Parsing Expression grammars implemented in Lua [110] could be used to automatically generate the state machines from XABSL files. This would make the framework a possible alternative for the RoboCup teams currently employing XABSL.

The software will be released soon as Open Source Software under the GNU General Public License. This should allow others to join the development effort to support more platforms and to extend the framework.
Bibliography


