A Framework for Reusable Robot Behaviors in Human-Robot Interaction

Bachelorarbeit

im Fach Kognitive Informatik
an der Technischen Fakultät
Universität Bielefeld

von: David Klotz
dklotz@techfak.uni-bielefeld.de
Matrikelnummer 1793913
Ravensberger Str. 77
33602 Bielefeld
GERMANY

Betreuer: Ingo Lütkebohle
Lars Schillingmann

6. August 2008
# Contents

1. **Introduction**  
   1.1. Goals of this Thesis ........................................ 2  
   1.2. Overview ..................................................... 2  
   1.3. Existing Soft- and Hardware Environment ................. 3  
      1.3.1. BARTHOC ................................................. 3  
      1.3.2. BWorld .................................................. 3  

2. **Behavior-Based Robotics** ........................................ 6  
   2.1. What are Behavior-Based Robotics? ......................... 6  
   2.2. The Subsumption Architecture ............................... 7  
      2.2.1. Coordination in the Subsumption Architecture ........ 8  
   2.3. Other Behavioral Architectures ............................. 9  
      2.3.1. Coordination with Motor Schemas ...................... 9  

3. **The Behavior Framework** ........................................ 10  
   3.1. Conceptual Design ........................................... 10  
      3.1.1. Behaviors and Resources ............................... 10  
      3.1.2. Outputs ................................................ 11  
      3.1.3. The Arbitration Mechanism ............................ 12  
   3.2. Implementation .............................................. 13  
      3.2.1. Architecture of the Framework ........................ 13  
      3.2.2. The Arbitration Process ............................... 14  
      3.2.3. Exchangeable Algorithms for Arbitration .......... 16  
      3.2.4. Utilities for Realizing New Behaviors and Resources . 16  
      3.2.5. Combining Behaviors .................................. 17  

4. **Experimental Evaluation** ....................................... 18  
   4.1. The Greeting Scenario ...................................... 18  
   4.2. The Pointing Scenario ...................................... 19  
      4.2.1. Evaluating Different Prioritization Policies ........ 20
4.3. Problems Encountered ........................................ 21

5. Conclusion ..................................................... 22
   5.1. Future Work ................................................ 22

A. Erklärung ........................................................ 24

B. Tutorial .......................................................... 25
   B.1. The First Behavior: Looking Up & Down .............. 25
   B.2. Trying to Do Two Things at Once ..................... 27
   B.3. Getting Your Priorities Straight ....................... 27

Bibliography ....................................................... 29
1. **Introduction**

The stereotypical image of a robot today is still that of a seemingly dumb machine, preprogrammed for completing a specific task repeatedly, such as mounting one part to an automobile frame.

Recent developments in robotics are more and more focused on building robots that are different from that: Robots being able to move around freely in a normal home, office or even outdoor environment, robots meant to directly interact with people, helping them in achieving everyday tasks.

An example of this would be a service robot finding his way around in a normal apartment and help an elderly or disabled person to accomplish household tasks they may not be able to do by themselves.

Controlling such robots in their unknown, unstructured and dynamic environments is a task that is completely different from controlling a robot manipulator in a factory. The robot’s reactions to changes in the environment need to be fast. It needs to be able to pursue multiple concurrent goals. Some of those goals (or the means necessary to achieve them) may even be in conflict with each other. The consequence is that robots like that can’t be preprogrammed in a totally deterministic manner.

One approach having proved itself to be well suited for such robots is the concept of *behavior-based control*. This approach takes the notion of a behavior (or behavioral module) from biology and applies it to the controlling structure of a robot.

By not relying on a central planning component, systems built with that approach can react faster and adapt themselves more easily to changes in their environment, without continuously having to synchronize their sensing with an internal model of the world.
1.1. Goals of this Thesis

The aim of this thesis is to design and to build a flexible system based on the concepts of behavior-based robotics, that can be used to try out new ideas and to implement new robotic scenarios easily. Like all behavior-based architectures, this system will take some ideas from the biological world, but only as far as these ideas are helpful in building a usable system. The system will be built with a software engineering approach focusing on trying to make it as easy as possible to program a robot for a new scenario or environment.

Specifically, it should be as easy as possible to build a new behavior and integrate it with a system of already existing behaviors. Such a framework provides a fixed structure that, while restricting the possible ways of building the controlling system of a robot, also makes it easier to divide the work this kind of system has to do into separate, independent modules. As a result these modules will be easy to maintain and to debug because of their clear separation of concerns.

The version of the behavior framework developed in this thesis is aimed primarily at controlling one robot, BARTHOC, that is described in section 1.3.1. But in principle it is independent from the specific robot used. Some of the behavioral modules developed using this system may be specific to the actuators or sensors of a robot, but the general arbitration mechanism (described in section 3.1.3) works independent from what types of behaviors are used with it.

1.2. Overview

In the next sections of this chapter, I will give a short introduction to the tools I have used in realizing this system and to the robot platform used for demonstrating it.

In the second chapter, I will present the behavior-based approach to controlling a robot in more detail. I will describe some examples of this approach and explain the differences between it and other approaches to controlling a robot’s actions.

The third chapter describes the system realized, first on a conceptual level, then on the level of the concrete architecture that was built. It gives a short introduction of the interfaces one needs to implement and the utilities that are available to realize new behaviors.

The fourth chapter showcases two example scenarios realized by using the system and tries to experimentally evaluate the influence the choice of the priori-
tization policy for the arbitration mechanism has on the overall behavior of the robot.

The final chapter presents a short conclusion and describes some possible improvements for the system created here and some future work that may be realized on top of it.

The appendix contains a small tutorial to help people to get started with writing their own behaviors using this framework.

1.3. Existing Soft- and Hardware Environment

The system developed in this thesis is initially targeted at the humanoid robot BARTHOC, which is described in the next section. It builds upon the foundation of the BWorld application used for controlling and simulating the robots motions.

1.3.1. BARTHOC

BARTHOC, which is an acronym for “Bielefeld Anthropomorphic RoboT for Human Oriented Communication”, and his smaller cousin BARTHOC junior are humanoid robots designed in close cooperation between the robot manufacturer MABOTIC\textsuperscript{1} and the Applied Computer Science Group at Bielefeld University.

While these robots are not mobile (they are modeled only from the torso upwards), they have enough degrees of freedom to convey believable human-like expressions using their hands, arms, torso, head and face. Both of them have two firewire cameras contained in the steerable eyeballs. Stereo microphones can be used to provide a sense of hearing.

They are used primarily for research in studying embodied interaction and human-robot communication. See [HSF+05] for a description of the platform and its intended uses.

Figure 1.1 shows BARTHOC’s head and upper body.

1.3.2. BWorld

Most of the work of this thesis will be done in the context of an already existing system for the simulation and the control of the robots BARTHOC and BARTHOC junior. This application is called BWorld (which is short for “BARTHOC’s World”), written by Ingo Lütkebohle (see [Lü08] for an introduction). It deals mainly with

\textsuperscript{1}http://www.mabotic.de/
Figure 1.1: BARTHOC

providing a graphical interface for sending motion commands (the desired angles of some or all of the robots joints) to the real robot control hardware and for displaying a simulated robot that can show the results of those commands even when the software is not connected to the real robot.

It can also act as a networked server using the XCF integration framework\(^2\) (described in [WFBS04]) receiving commands from remote clients and relaying them to the robot hardware. Nevertheless this functionality was not used in this work (although separating the behavior framework from BWorld might be an option in the future).

BWorld also provides a mechanism for accessing its functionality through a scripting interface, using the Rhino JavaScript interpreter being contained in the Java platform since Version 1.6. This was used extensively in providing a flexible way to instantiate and set up all the components of the behavior framework (and the individual behaviors using it) while keeping it relatively independent of the rest of the BWorld application.

Figure 1.2 shows the BWorld application running one of the example scenarios realised during this thesis.

\(^2\)http://xcf.sf.net
Figure 1.2.: The BWorld Application
2. Behavior-Based Robotics

This chapter first explains the general concept of behavior-based robotics. It then gives a short introduction to two archetypal behavior-based architectures. First to the subsumption architecture, one of the first architectures to realize this idea, and then to motor schemas, contrasting mainly their methods of coordinating the outputs of different behaviors.

2.1. What are Behavior-Based Robotics?

Behavior-based robotics are an approach to building the controlling system of a robot. The method is called by several different names, including behavior-based control (for example in [Mat97]) and even nouvelle AI (in [Bro90]), but the general concept stays about the same.

One thing all these systems have in common is that they are made up of relatively small, mostly independent behavioral modules listening for input from the world (which may include other modules) and generating some form of output for controlling the robot. The inputs are often called by the biologically inspired term stimuli and can come from sensors measuring properties of the environment or from other modules inside the robot. Only the combination of the different outputs of all these modules is what defines the outwardly visible behavior of the robot.

The inspiration for most of these systems stems from biology, where ethologists like Konrad Lorenz try to model the overall behavior of animals as being composed of different smaller behavioral units (like a fleeing behavior or a display behavior). Most of these behaviors are activated by some kind of external signal, their stimulus. See [Ark98, section 2.4] for a good overview of how animal behavior can serve as a guide when developing behaviors for robots.

Motivation for this new approach came from the apparent lack of progress some researchers thought was being made by classical, symbolic Artificial Intelligence in realizing intelligent systems interacting with the real world. Rodney Brooks
even went so far as to argue that all systems reasoning over some sort of abstract symbols were fundamentally flawed because the gap between the real world and the symbols used to represent it could never be closed adequately (see [Bro90, section 2]).

Among other reasons, this lead to purely reactive systems that directly mapped sensory conditions to output signals to a robot’s actuators. This can be seen as one end of a spectrum of different robot control architectures, the other end being deliberative systems build with traditional AI planners working on symbolic representations of the world. Behavior-based systems usually lie somewhere towards the reactive end of that spectrum, but they are not totally opposed to storing some kind of permanent representation if that is helpful in achieving the behavior’s goal. See [Mat97] for a more thorough explanation of the difference between deliberative, purely reactive and behavior-based systems.

2.2. The Subsumption Architecture

A very influential early behavioral architecture for robotic systems is the “Subsumption Architecture” designed in the 1980s by Rodney Brooks, then at the MIT Artificial Intelligence Laboratory. The first description of it can be found in [Bro86]. His idea (which was radically different from most mainstream artificial intelligence research at that time) was that you didn’t need a central mechanism planning every move of a robot (and its outcome in the world) in advance to get a robot to show intelligent behavior. Instead he proposed (and realized in several robots) a system made up of several small modules. Each of these modules did only very little processing but together they managed to realize several interesting behaviors. Most importantly, there was never any kind of central representation of what the robot “thought” his environment looked like. Instead, most of the modules worked only with the data being available directly from the robots sensors.

He contrasted this decomposition (see figure 2.1) into several independent modules (the behaviors) to the more traditional approach of building several stages being more or less executed one after the other (with perception coming first, followed by some kind of deliberative reasoning and at the end choosing some action based on the outcome of the reasoning process).
2.2.1. Coordination in the Subsumption Architecture

In the subsumption architecture, each behavior is realized as a finite state machine that additionally has an internal clock which enables it to trigger state changes by itself (Brooks calls this an “augmented finite state machine”, AFSM). These behavioral modules are connected using “wires” (sometimes simulated in software, sometimes using real wires) that can pass short numerical messages from one module to another. It is up to each module how this message is interpreted.

Competition between different behaviors is realized by using two mechanisms: Suppression of inputs and inhibition of outputs. The first means that an input wire going into a module (and coming from a different module) can essentially be overwritten by a suppressing wire coming from a third module. This stops any communication on the original wire and replaces the input by that coming from the suppressing wire. The second means that messages on an output wire of a module can be “stopped” by an inhibiting wire. They are not replaced by the messages arriving on the inhibiting wire, their contents are simply lost. Using
these two mechanisms, behaviors can “subsume” the function of other behaviors by manipulating their in- and outputs, which gives the architecture its name.

2.3. Other Behavioral Architectures

There were other robotic systems built after the subsumption architecture also calling their controlling architecture “behavioral” or “behavior-based”, some of them being extensions of the original subsumption architecture, others being new systems built with similar goals in mind. See [Ark98, chapter 4] for a comparison of some of them.

One example is an architecture using so called “motor schemas”, described in [Ark98, section 4.4]. Its foundation is schema theory, which can be seen as a “higher level abstraction by which behavior can be expressed modularly” ([Ark98, p.41]). A schema in this architecture is basically the same as a behavioral module in the rest of this chapter. It is the smallest unit of behavior from which more complex behaviors (called “assemblages” by Arkin) are constructed.

2.3.1. Coordination with Motor Schemas

In Arkin’s architecture, each schema has an associated gain value. This value can either remain fixed throughout a run of the robot or be adjusted at runtime using some learning or adaptation mechanism. While the robot is running, each schema continuously generates an output vector that has the same dimensions but can differ in direction and magnitude. This vector is then multiplied by the schema’s gain value and the vectors of all the schemas currently running are added together, producing a combined vector that is then used for controlling, for example, the robots movement direction and speed.
3. The Behavior Framework

This chapter describes the system realized in this thesis. First I try to explain the general ideas and key concepts used in its design, then I describe the architecture and some implementational details of the software realized.

3.1. Conceptual Design

The system’s main building blocks are behaviors, their outputs and resources. Behaviors are the active parts that, in collaboration, control the actions of the robot. They generate outputs that are then sent to a central mechanism that prioritizes them and distributes them to the correct resources (see figure 3.1).

![Figure 3.1: Behaviors, Resources and Outputs](image)

One of the most important points in this concept is the fact that the coordination of the different parts of the system takes place at the level of the outputs, not at the level of the resources or the behaviors. This process is also, in a way, distributed among the resources, because the central arbitration mechanism only prioritizes the outputs, allowing the resources to decide individually what to do with them.

See the next sections for a more thorough description of these concepts.

3.1.1. Behaviors and Resources

Behaviors are the active building blocks from which the system controlling the robot is built. They are the smallest behavioral units in this system. In principle, most behaviors are running continuously and concurrently, but new behaviors can also be started and old ones can be stopped at runtime. This enables the system
to dynamically adapt itself to changes in the environment or in the robots internal state.

Behaviors typically produce some kind of output (see the next section for a description of what characterizes an output) targeted at one or more resources. This implies that outputs can somehow be “applied” to a resource. A resource can therefore either be a real actuator (like a motor or a servo controller directly governing the robots motions) or it can be some other behavior. This means that all behaviors can also act as resources and can also be used as the target of another behaviors outputs.

This hierarchical approach makes it easy to build a new system in a bottom-up fashion. First we can identify which resources can be combined into a behavior controlling them, which in turn can be used by other behaviors built on top of it. Those new behaviors can then just depend on the existing behavior doing its job (or influence it by giving it new inputs). The existing behavior does not have to be modified even when the overall capabilities of the robot are expanded by adding new behaviors.

### 3.1.2. Outputs

An output is a message being sent from a behavior (via the arbitrator) to a resource. See section 3.1.3 for a description of the process happening inside the arbitrator.

This message can be anything from a simple text string to a directional vector or a motor command (specifying the desired angles of different joints). For example, the output of a behavior might be a command like “move servo motor x to position y” (that would be applied to a resource directly controlling the specified motor) or it could be a direction (given as a vector, for example) the robot should be facing. This second output would be targeted at another behavior generating outputs for all the actuators needed to turn the robots head in the right direction.

The general behavior framework is totally indifferent to the specific type of output sent, relying only on it to conform to a specified interface. This interface specifies several attributes all outputs should be able to provide. These attributes specify values for the estimated duration of the action resulting from applying this output, the safety implications of running it, the degree to which they help fulfill the robots current tasks and the knowledge that could possibly be gained by it. The arbitration mechanism described in the following chapter uses these attributes in determining this output’s priority.
3.1.3. The Arbitration Mechanism

One of the main problems that needs to be solved in all behavioral architectures for robots is how the outputs of different behaviors are combined, in effect deciding which behavior gets to control the robot’s actions at each point in time. This process, choosing which output will be applied to what resource in what order, is called arbitration.

In the realized architecture, behaviors submit their outputs to a central arbitration mechanism. This arbitrator then calculates a single numerical priority based on the different attributes of the output (see section 3.1.2 for more information on outputs and their attributes). Afterwards, the arbitrator submits the output and the calculated priority to all resources potentially affected by it.

The resources then can decide what they should do with the output based on the priority. One possibility would be to queue the outputs. This queue can be ordered by the priorities, so that outputs with a higher priority get processed before those with lower priorities. Another option would be to simply replace the current action or state of the resource with a new one based on the newly arrived output if it has a higher priority than the last one.

Outputs could also be merged, resulting in an action that combines the values of several outputs into one. An example would be a resource (or a behavior) that combines several of the directional vectors it received into one direction being the average of all of them.

A resource can also define a threshold and only accept outputs with a priority being above that value.

The algorithm used for calculating the priority from the attributes of the output can be easily exchanged or modified, resulting in a central place to influence the robot’s overall behavior. Depending on how much the different attributes of each output are valued in the calculation of the priority, what the robot does can be influenced just by giving it a different prioritization policy.

The resources can use these priorities in different ways. Some of them may be similar to subsumption (outputs with higher priorities suppress those with lower priorities) while others could use the priorities to give different weights to different outputs, emulating the weighted vector summation in motor schema based architectures described in 2.3.1. You can also combine resources using different approaches in one scenario.
3.2. Implementation

Based on the ideas described in the previous sections a system was realized that can be used to control the actions of a real or a simulated robot in the environment described in section 1.3.

Figure 3.2.: The Main Interfaces of the Framework

This system provides interfaces describing the key concepts of the behavior, the resource and the output. Their relationship can be seen in figure 3.2. A newly written behavior needs to implement the Behavior interface, the outputs it produces need to implement the Output interface and each resource that can be a target of these outputs needs to implement the Resource interface.

3.2.1. Architecture of the Framework

Additionally to these general interfaces, the system also provides an arbitration mechanism and several other classes working with the interfaces creating a framework that can be used to realize new behaviors.

When we take that arbitrator and its prioritization policy (used to generate the priorities described in section 3.1.3) into account and add default implementations for the behavior and the output, we end up with the architecture shown in figure 3.3 on page 14. You can see that a name was also added to the output that can be used for distinguishing between different outputs of the same type.
3.2.2. The Arbitration Process

Figure 3.4 on page 15 gives an overview exemplifying the most important process in this system, routing outputs from a behavior to a resource.

First a behavior creates an output which has to be some class that implements the `Output` interface (see section 3.2.4 for some simple examples). Then it submits that output to the arbitrator, receiving an object of the class `Future` (from the Java framework for concurrency) in return. A `Future` is an object that can be queried for the state and the result of an activity that happens asynchronously in another thread. In this case the asynchronous activity is the output successfully reaching the resource (or resources) it was targeted at.

The arbitrator then (in another thread) gives this output a priority and applies it to all its affected resources. The resources also each give the arbitrator a `Future` object, allowing them to be queried for the completion of processing the outputs they have received. When this is completed, this also propagates through to the original `Future` in possession by the behavior that originally submitted the output.
Figure 3.4.: The Arbitration Process

It is important to note that steps 3 and 5 take place asynchronously, on a different thread and step 2 returns immediately.
3.2.3. Exchangeable Algorithms for Arbitration

The policy used by the arbitrator in generating priorities for the different outputs is also specified using an interface. This means that this policy can be exchanged without having to change anything directly in the arbitrator itself.

![Figure 3.5: Different Policies for Prioritization](image)

This can be used to change the overt behavior of the robot by simply changing its valuing of different outputs. See figure 3.5 and section 4.2.1 for an example of how this can be utilized.

3.2.4. Utilities for Realizing New Behaviors and Resources

Because most behaviors are running in their own thread, implementations of the different methods for starting a behavior, stopping it and checking if it is running tend to resemble each other quite often. To reduce this duplicated code, the framework includes an abstract class called `ThreadedBehavior` that provides a default implementation of these methods using one single thread per behavior. This leaves only the thread’s “run” method to be implemented when writing a new behavior.

Since a lot of behaviors need to produce some kind of output (and all those outputs need to handle their attributes in a similar way), an abstract implementation of the output interface is provided. This already handles all the output’s attributes and can be extended to realize custom forms of output. There is also a generic output that can be used to create an output from any kind of Java object.

This is also where the existing BWorld API comes into play. The `QueueRunner` class from the BWorld source can be used to execute actions in the order of the priorities of the outputs a resource receives and the `MotionExecutor` (a class for
sending joint angles to the robot’s hardware and to the simulator) was retrofitted with the methods needed to use it as a Resource.

3.2.5. Combining Behaviors

To combine several behaviors into a useful scenario, they can be instanced, configured and started from a file containing JavaScript code. This file can be loaded and run in BWorld using the built-in scripting interpreter. The tutorial in appendix B contains an example script used in this way.
4. Experimental Evaluation

This chapter describes two scenarios that were realized to give an example showing what can be done using this framework. The first scenario highlights the general interaction of several behaviors and their outputs while the second scenario shows how a different prioritization policy can change the overt behavior of the robot.

4.1. The Greeting Scenario

The first demonstration scenario was realized to show the general process of writing new behaviors communicating with each other in terms of their outputs and the priorities assigned to them. It made the robot look around idly in random directions as long as nothing else happened. If a new visitor arrives, the robot turns its head in the direction of the visitor and greets him by waving using his right arm.

To keep this as simple as possible, sensory input in this scenario is just simulated by telling the greeting behavior of newly arrived “visitors” (and their locations) using the scripting interface.

![The Greeting Scenario Diagram]

Figure 4.1.: The Greeting Scenario

The behaviors created for this scenario are called ViewDirection, ArmGestures, IdleBehavior and GreetBehavior. It also uses a MotionExecutor resource that
sends motion-commands directly to the robot. See figure 4.1 for an overview showing these behaviors and the flow of outputs between them.

The ViewDirection acts as a resource that can receive three-dimensional vectors describing the direction the robot should be looking into. It transforms these vectors to the two angles (pitch and turn) needed to control the orientation of the robot’s head. These angles are then encapsulated in an output being sent to the MotionExecutor resource. This behavior is only activated when it receives an input, otherwise it does not generate any output of its own.

The ArmGestures behavior also acts as a resource. It receives simple textual messages telling it which gesture to activate. At the moment only one gesture, called “wave”, is supported, but in principle multiple gestures could be activated using different messages. The gestures are composed of sequences of postures (which are just collections of joint angles) that are sent directly to the MotionExecutor. The ArmGestures also are only active after having received an input. Figure 1.2 on page 5 shows the greeting gesture from this scenario in the BWorld application.

Every second the IdleBehavior randomly chooses a new target vector to look at. It then generates a sequence of vectors interpolated between the last target vector and the new one and sends this collection of outputs to the ViewDirection, which in turn interprets it as a path that should be followed.

The GreetBehavior waits until it is told about new visitors using the scripting interface. When this happens, two outputs are generated. One is targeted at the ArmGestures, telling it to activate the “wave” gesture, the other one is targeted at the ViewDirection, telling it to look in the direction of the new visitor. Both of these outputs have some of their attributes set to elevated values.

The result is that the outputs of the GreetBehavior are assigned higher priorities than the outputs of the IdleBehavior, thereby interrupting the path generated by the latter to instantly look in the direction of the visitor.

4.2. The Pointing Scenario

To show the influence of different prioritization policies, a second scenario was created that highlights a difference in the overt behavior of the robot resulting only from using a different prioritization policy. See figure 4.2 for an overview of the behaviors in this scenario.

In this scenario, one behavior (the PointBehavior) tells the robot every once
in a while to point in a random direction. Another behavior (the `RestBehavior`) monitors the target angles of the joints in the arm using a simulated sensor that produces an increasing level of “strain” as long as the arm is in an upright position. Both of them send their outputs to a third behavior (the `PostureBehavior`) that uses them and their assigned priorities in deciding when to send which posture to a `MotionExecutor` resource. Another separate behavior (the `ObstacleAvoidance`) also monitors the target angles of several joints trying to keep the robot’s arm clear of a hypothetical obstacle. When it detects angles that would put the arm in a dangerous position, it immediately takes countermeasures by generating outputs aimed at the `MotionExecutor` trying to steer clear of the obstacle.

### 4.2.1. Evaluating Different Prioritization Policies

In this scenario, two different prioritization policies were tested while keeping all the behaviors and resources configured exactly the same.

One policy makes the robot regain its resting position as soon as possible. It does this by giving low priorities to outputs that are aimed at making the robot fulfill its intended task. Instead it gives higher priorities to outputs that try to put the robot in a safer position or otherwise prevent damage being done to it. This policy is called the “lazy” prioritization policy, because that’s how the effect on the robot’s overt behavior looks from the outside.

The second policy does almost the exact opposite, giving higher priorities to outputs with higher values of their “TaskFulfillment” attribute and lower priorities to outputs aimed at increasing the robot’s safety. This policy is called the “eager” prioritization policy, because it makes the robot appear more eager to complete the task it was given.

As can be seen in figure 4.3, the result is that the average time it takes from
starting to point to regaining the resting position almost doubles with the eager prioritization policy as compared to the lazy prioritization policy.

4.3. Problems Encountered

One of the problems encountered very early in developing scenarios for trying out this framework was finding the correct ranges of values for the different attributes of each output. In the beginning, almost all were unbounded integer values, but in the end I settled on discrete named levels because that makes setting them less arbitrary.

While developing the second scenario ("pointing"), another difficulty became clear: The prioritization policies are free to choose the priorities from the whole integer range. But they don’t necessarily generate values covering that complete range. This makes it hard to use the priorities for giving weights to different outputs, because one does not know the upper or lower bounds the priorities can reach. Section 5.1 presents an idea how this problem might be solved.
5. Conclusion

This system developed here hopefully lays the groundwork for building a library of reusable behaviors that can be quickly combined into more and more different scenarios necessary for the day-to-day research work being done with BARTHOC and other robots.

Care has been taken to make writing a new behavior as easy as possible. There is just one interface that has to be implemented. By extending the default, threaded behavior implementation, this can be reduced to one method that has to be written. Utilities for easily producing and using different kinds of outputs are also provided. This seems to be in line with the goals set at the beginning.

Although it does not completely free the developer from fine-tuning the interaction of the different parts of a robot’s control system, the behavior-based approach implemented here feels like a very natural way of programming a robot to show some desired behavior.

The scenarios implemented up to now have been relatively simple demonstrations. Already more complex interactions are being developed using this framework, exploring its applicability to real problems encountered in real usage situations.

5.1. Future Work

The problem of figuring out the range of values (mentioned in section 4.3) for the priorities could be solved in different ways. For example each prioritization policy could be given methods to query it for explicit upper and lower bounds of the values it generates or one such range could be mandated that all policies would have to use.

As was mentioned in section 1.3.2, the XCF integration framework could be used for communication between the behavior framework and the BWorld application. It could also be used for communication between the arbitrator and the different behaviors, thereby enabling them to be run on different networked computers.
One idea helpful in furthering the goal of a library of reusable behaviors would be a better integration of this framework into the user interface of BWorld, enabling users to control the running behaviors, to start new ones or to make adjustments without having to resort to the scripting interface.

Unfortunately, due to several malfunctioning joints, some behaviors (for example the waving using the “ArmGestures” from the scenario described in section 4.1) could not be tested on the real robot at the time of this writing.
A. Erklärung


Bielefeld, 6. August 2008, 

______________________________
B. Tutorial

This appendix contains a small tutorial to help anyone interested in getting started with writing behaviors for use with this framework. A behavior is a small independent unit that tries to achieve a sub-task of controlling the robot’s actions. A number of different behaviors will normally be running in parallel, each one playing a small, but important part in shaping the overt behavior of the robot.

In this tutorial we will create three simple behaviors, all three trying to control BARTHOC’s head in some way. All three will be creating outputs targeted at the same resource, a class called “MotionExecutor”. This resource receives “motions” (which are just collections of angles for the joints of the robot) and sends them directly to the robot using the BWorld API.

B.1. The First Behavior: Looking Up & Down

The first behavior we are going to create will just make the robot look up and down. To create a new behavior, you have to create a new class that implements the interface de.unibi.agai.behavior.Behavior. Because we don’t want to implement so many methods on our own, we choose to extend the abstract class ThreadedBehavior instead, which leaves us only with one unimplemented method:

```java
public class LookUpDown extends ThreadedBehavior {
    @Override
    public void run() {
        // We need to do something useful in this method
    }
}
```

Since we want to send outputs to a MotionExecutor, we need to know an object of that class. We choose to just pass it into the constructor of our newly created behavior. We also implement the run method with a loop that continuously sends outputs that tell the MotionExecutor to change the position of the “head_pitch” joint:
```java
public class LookUpDown extends ThreadedBehavior {
  private final MotionExecutor actuator;
  // ...

  public LookUpDown(MotionExecutor actuator) {
    this.actuator = actuator;
  }
  // ...

  public void run() {
    boolean stopped = false;
    while (!stopped) {
      increaseAngle(); // Increases the currentAngle
      // Create the output with the angle and the joint name
      PostureBuilder builder = actuator.createPostureBuilder();
      builder.add("head_pitch", currentAngle);
      MotionOutput output = new MotionOutput(builder.
                                           createPosture("UpDown"), actuator);
      // Submit the output to the arbitrator
      Future status = arbitrator.submitOutput(output);
      // We try to wait until our output has been applied
      status.get();
      // ...
    }
  }
}
```

To try this newly created behavior, we just create a JavaScript file that can be used by the scripting interpreter of BWorld:

```javascript
// ... // Create the arbitrator and start it
var arbi = Arbitrator()
arbi.start()

// Create the resource for controlling the robot
var actuator = MotionExecutor(robot)

// Create the behavior and start it
var lookUpDown = LookUpDown(actuator)
lookUpDown.setArbitrator(arbi)
lookUpDown.start()
```

Running this file in the interpreter should make the robot look up and down. Congratulations, you just wrote your first behavior!
B.2. Trying to Do Two Things at Once

Since running just one single behavior is a bit boring, let’s create a second behavior.
We will call this new behavior “LookLeftRight”, since it will tell the robot to do just that. It will run in parallel to the already existing first behavior. The only difference between this behavior and the first is that it uses the “head_turn” joint instead of the pitch.

```plaintext
// ...
builder.add("head_turn", currentAngle);
// ...
```

Since we haven’t set any of the parameters of both behavior’s outputs, they will probably receive the same priorities by the arbitrator. This makes the MotionExecutor just execute them in the order it receives them. Since it receives both outputs telling it to look up and down and outputs telling it to look left and right, this creates the “emergent” behavior of making the robot look in a zig-zagged pattern (or even a circle, depending on the timing).

To try this out we can just add this behavior to the end of the script after the first one:

```plaintext
// ...
lookUpDown.start()
// Create the second behavior and start it
var lookLeftRight = LookLeftRight(actuator)
lookLeftRight.setArbitrator(arbi)
lookLeftRight.start()
```

B.3. Getting Your Priorities Straight

To make this even more interesting, we add a third behavior, that every now and then tells the robot to look straight ahead for a longer duration.
This (again) works almost the same as in the two previous behaviors. We just add a long delay at the beginning of the main loop (since we just want this behavior to be active every once in a while) and give the MotionOutput another delay (since we want to look straight ahead for that duration).

We also set one of the attributes of the output to a higher value. While this depends on the concrete prioritization policy currently in use, we hope that this change will make the arbitrator give the output a higher priority before applying it
to the MotionExecutor so that our prolonged looking ahead will not be interrupted by the outputs of the two other behaviors (that should get lower priorities).

```java
public class LookStraightAhead extends ThreadedBehavior {
    private final MotionExecutor actuator;

    public void run() {
        boolean stopped = false;
        while (!stopped) {
            // We start with waiting 10 seconds
            Thread.sleep(10000);
            // ...

            // Just look straight ahead
            PostureBuilder builder = actuator.createPostureBuilder();
            builder.add("head_turn", 0.0f);
            builder.add("head_pitch", 0.0f);

            // Create the output and change its attributes
            MotionOutput output = new MotionOutput(builder.
                createPosture("StraightAhead"), actuator);
            output.setDuration(2000l);
            output.setTaskFulfillment(TaskFulfillmentLevel.
                BENEFICIAL_FOR_TASK);

            // Because of the long delay, we don’t wait for the output
to complete
            arbitrator.submitOutput(output);
        }
    }
}
```

This behavior can be added to the script file just like the last two:

```java
// ...
lookLeftRight.start()

// Create the looking ahead behavior and start it
var lookStraightAhead = LookStraightAhead(actuator)
lookStraightAhead.setArbitrator(arbi)
lookStraightAhead.start()
```

Running this creates the same overt behavior like the last example, except that every ten seconds, the robot looks straight ahead for two seconds.
Bibliography


