Behavior-based Control of the ERS-7 AIBO Robot

Stefan Bie
Johan Persson

Examensarbete för 20 p, Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet.

Thesis for a diploma in computer science, 20 credit points, Department of Computer science, Faculty of Science, Lund University.

Abstract

This report considers behavior-based control for the AIBO ERS-7 robot. This was the first time that the ERS-7 model was used by Team Chaos which led to the construction of a new software framework. The framework contains 5 modules: Vision, Localization, Wireless, Behavior and a common memory area called Worldstate. It became our responsibility to construct the behavior module. During the work with constructing the behavior module we studied several different approaches for decision making for autonomous agents. We start by examining the hierarchal structure of decision trees and then the possibility of using symbolic planning. Finally we examine state machines and hybrid automata. Our conclusion is that the state machine approach best suites the requirements of our solution. The requirements were that the solution would have a modular construction, be reusable and solve the challenges in Robocup 2004. Hybrid automata is only considered as a verification method, since the methods that are developed for hybrid automata can not verify a complex system like an AIBO and its enviroment. The last part of the report is a manual that explains how to continue the work from what we have done.

Sammanfattning

Denna rapport handlar om beteende för AIBO ERS-7 robotar. Detta var första gången som Team Chaos använde sig av ERS-7 modellen. Detta ledde till att ett nytt ramverk för robotmjukvaran skulle utvecklas. Det nya ramverket kom att bestå av 5 moduler: Vision (syn), Localization (lokalisering), Wireless (kommunikation), Behavior (Beteende) samt ett gemensamt minnesutrymme (Worldstate). Det kom under vårt ansvar att konstruera modulen för beteende. Under arbetet med att konstruera beteendemodulen studerade vi några olika metoder för beslutsfattning för autonoma enheter och vi går igenom vissa av dessa i rapporten. Vi börjar med att utreda användandet av en hierarkisk struktur, typ beslutsträd, senare utreder vi möjligheterna att använda sig av planering. Slutligen undersöker vi tillståndsmaskiner och hybrida automater. Vi kommer fram till att tillståndsmaskiner är den metod som bäst tillgodoser de krav som vi hade på vår lösning. Dessa krav var att lösningen skulle vara modulärt uppbyggd, återanvändbar och passa för utmaningarna i Robocup 2004. Hybrida automater behandlas i första hand som ett sätt att verifiera vår lösning, eftersom de metoder som nu finns utvecklade inom detta område inte kan bevisa ett så komplext system, som en AIBO med sin omgivning är. Sist i rapporten finns en manual som förklarar hur någon som vill bygga vidare på vår lösning kan gå till väga.

1 Introduction 3
1.1 Background 3
1.2 Problem Definition 3
1.3 Thesis overview 4
2 The AIBO Robot 5
2.1 Hardware and Peripherals 5
2.2 The Aperios Operating System 7
2.3 The OPEN-R Software Development Kit 8
2.4 Tekkotsu 9
3 Robocup 11
3.1 Four legged Robot League Championship 11
3.2 Four-legged Robot League Challenges 2004 13
3.2.1 The Variable Lighting Challenge 13
3.2.2 The almost SLAM Challenge 13
3.2.3 The Open Challenge 14
4 Team Chaos Challenges Framework 15
4.1 Vision 16
4.2 Localization 16
4.3 Wireless 17
4.4 Worldstate 17
4.5 Behavior 17
4.6 Tekkotsu Interface 17
5 Behaviors 18
5.1 Decision Trees 20
5.2 Planning 21
5.3 State machines 22
5.4 Hybrid automata 23
6 Behaviors in TCC 24
6.1 Using finite state machine approach. 25
6.1.1 States 25
6.1.2 Input alphabet 27
6.1.3 Output alphabet 27
6.1.4 Output function 27
6.1.5 Transition function 28
6.1.6 Start state 30
6.2 Solving Variable lighting challenge Behavior problem 31
6.3 Solving SLAM challenge behavior problem 32
7 Results 35
7.1 Tests 35
7.2 Analysis of the result 36
8 Conclusions and Future Work 37
References 38
Appendix 39
A. Class structure 39
B. States 40
Walk 40
Find 42
Align 43
TurnWithBall 45
ApproachBall 46

TurnToObject ... 47
Kick ... 48
MarkPoint ... 48
C. User's Manual ... 49

1 Introduction

This master thesis was written during spring and summer 2004. It consists of two parts, this report and a program. The thesis is about how to implement behaviors on a Sony AIBO ERS-7 robot [1]. The AIBO robots provide an excellent platform for developing robotic technologies and they are frequently used in research and education all over the world. A popular environment for the robots to act in is the soccer environment, and this is also where our behaviors are intended to operate.

1.1 Background

Robocup is an annual international robot soccer competition promoted by the Robocup Federation [2]. Team Chaos[3] (formerly Team Sweden) is a Swedish national team that has participated in Robocup's Sony legged league since 1999. In this league all teams use the same physical platform, a special version of the Sony ERS AIBO four-legged robot.

Team Sweden entered this years Robocup competition in Portugal in June/July 2004. This year the team included three Swedish academic sites: Örebro University, Lund University, Blekinge Institute of Technology, and the University of Murcia, Spain.

This was the fifth time that Lund participated. Lund was represented by seven students and one PhD student. Last year a new robot model was introduced, the Sony AIBO ERS-7. In previous Robocup participations Team Sweden used an older AIBO model, but now it was changed to the ERS-7 model since this model was an improvement in terms of size, speed, sensor resolution and more. It was considered to be a better idea to develop a new software framework than to port the old one. This resulted in a natural sequence of work for the project.

1. Exploring the hardware and software systems.
2. Development of the new framework
3. Task specific studies, implementation, testing and documentation.

1.2 Problem Definition

The objective of this project is a design and realization of the behavior-based control for ERS-7 robots. In particular, the design should allow for handling different tasks defined by the user. It may also involve planning capabilities. The particular task to be chosen should fit the description of the challenges of Robocup 2004.

The task definition should be done in a modular and compositional way. One possibility to achieve this goal is to investigate finite state machines as the specification formalism. Implementation should fit the framework of the AI@CS software architecture (tentatively named TCC Framework).

1.3 Thesis overview

The report is divided into structured chapters, which we have tried to put in a logical order. The first two chapters describe the robot and the environment that it acts in. Then the framework that was developed is described. Chapter five and six describes different methods to implement behavior control and the method that we choose. The result and possible improvements are discussed in chapter seven and eight. Finally there is an appendix that contains more detailed information about our implemented solution.

2 The AIBO Robot

AIBO is a four-legged robot that is manufactured by Sony. The first generation AIBO (ERS-110) was launched in 1999. In September 2003 the third generation of the AIBO was released, ERS-7. This is the latest model on the market today.

This document will only refer to the ERS-7 model since the program is developed for that model. ERS-7 was chosen because it is better one of the two models currently used in the Sony legged league, the other one is ERS-210.

2.1 Hardware and Peripherals

AIBO has a wide range of sensors and effectors, which it uses to interact with the environment. The core of AIBO is a 64 bits RISC processor operating at 576 MHz. The robot uses two types of memories, an internal and a removable memory. The internal memory can store 64 MB of data, and the removable memory is a 32 MB Memory Stick. The robot is programmed through the Memory Stick. An illustration of the robot can be found in figure 1.

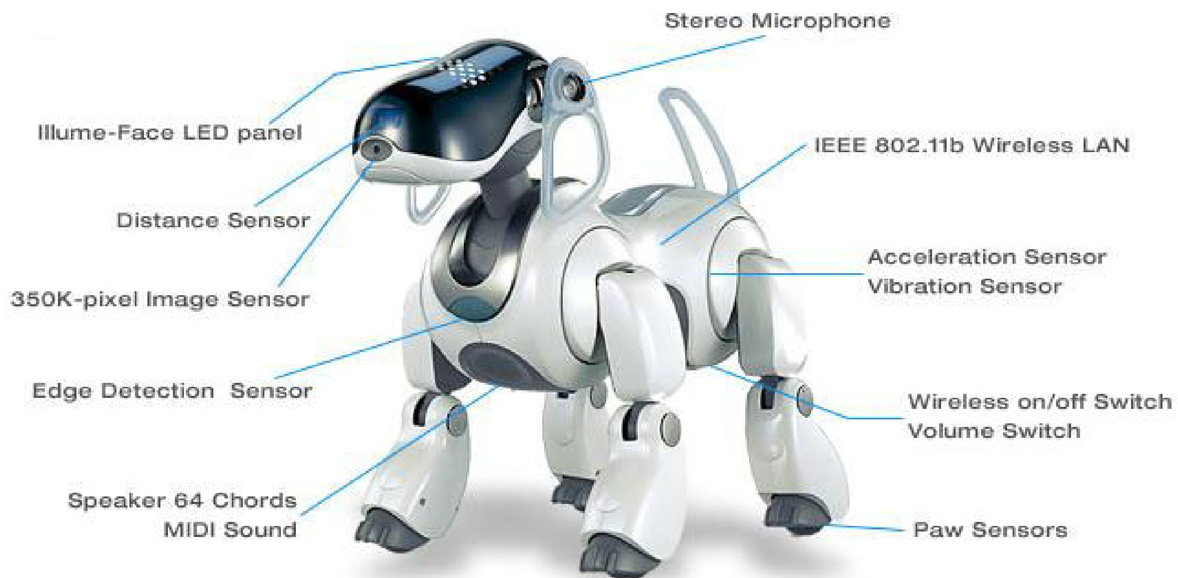


Figure 1. Sony ERS-7.

The sensors are used to get information about the environment. They are updated every 32 ms, and every update is sampled four times. AIBO has several different sensors, which can be divided into three different senses; Feeling, Seeing and Hearing.

- **Feeling**. AIBO has two types of **touch sensors**, electrostatic, and pressure. The electrostatic sensors are positioned on the dogs head and back. The pressure sensors are positioned under the paws and chin. AIBO has **acceleration sensors** that is sensitive to movement in all three dimensions, i.e., left-right, front-back, up-down. The robot also has a **temperature sensor** and a **vibration sensor**.

- **Seeing.** AIBO has a **CMOS image sensor** that is used to get pictures of the environment. The image sensor has 350 000 pixels, and it is capable of taking pictures at 30 frames per second. The camera can work in three resolutions 208x160, 104x80 and 52x40. It has a 56.9 degrees horizontal angle of view, and vertically it has 45.2 degrees angle. AIBO also has distance sensors placed in the nose and on the chest. The distance sensors in the nose consist of a long range and a short range sensor. Together they can operate at distances from 10 cm to about 90 cm.

- **Hearing.** AIBO can record sounds with its two microphones located at each side of the head. The sound can be recorded in stereo at a sampling frequency of 16000Hz in 16bits linear pulse code modulation.

The effectors are used to interact with the environment. AIBO has several different effectors, which can be divided into two types; movement and communication.

- **Movement**. AIBO has four legs with three joints in every leg (elevate, rotate, knee). It has a head with three joints in the neck (tilt, pan, and nod), a tail with two joints (tilt, pan) and a mouth with one joint. The ears can also move (flick up or down).

- **Communication**. AIBO has three types of communication. visual, audible and wireless.
  - The visual communication is done by LEDs. There are LEDs on the dogs head and back.
  - The audible communication is done by a miniature speaker located on the chest. The speaker can play polyphonic sounds.
  - The wireless communication is done by an IEEE 802.11b wireless Ethernet interface.

2.2 The Aperios Operating System

Aperios [4] is an object-oriented, distributed real-time operating system based on meta-level architecture. It was formerly known as Apertos and exists in many of Sony's products today, including AIBO. It is very small compared to other operating systems as it only takes up about 100 KB of storage.

The only constituent in Aperios is the object. For the object to be free to move around a distributed environment, it is open to evolve and adapt to its environment. Therefore the semantics and property for an object change while it is running and when it evolves or moves. Thus, there is *object/meta-object separation*. An object is just a container of information, whereas a meta-object defines the semantics of its behavior.

Each object has its own group of meta-objects, which gives an object abstract instructions or meta-operations that defines the object's semantics. For example, a scheduler meta-object contains details on how to schedule thread or process objects. Since a meta-object is also an object, there can be meta-objects for that meta-object. That means there is a *meta-hierarchy*, within which an object and its group of meta-objects are defined, as illustrated in figure 2.

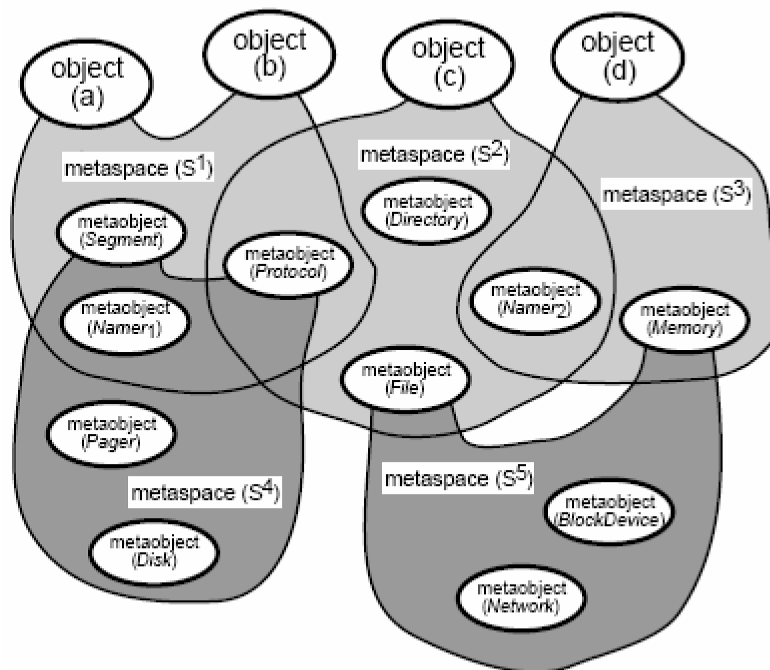


Figure 2. Object/Metaobject separation and metahierachy

The object uses a meta-hierarchy of meta-objects to define its behavior. The set of metaobjects that an object uses is called *meta-space*. If an object cannot continue its execution as it evolves or changes, it can *migrate*, i.e., change its group of meta-objects, to a new meta-space to continue its execution. For example, if it wants TCP/IP communication and its meta-space do not support this, the object can migrate to other meta-space that supports this kind of communication.

At run-time, an Aperios application is supported by one of many meta-spaces. Each meta-space contains at least one meta-object, called the *reflector*. A reflector acts like a gateway; it

intercepts incoming requests to the meta-space and forwards them to the appropriate meta-object. For example, a scheduling request will be sent by the reflector to the scheduler meta-object.

2.3 The OPEN-R Software Development Kit

OPEN-R [5] is a modularized and object-oriented software which has a layered architecture. It serves as an abstraction layer to the Aperios operating system. OPEN-R consists of two layers: a system layer and an application layer.

- **The system layer** contains the necessary services to access the robots hardware. There are three important objects that are provided by the system layer and they all provide different services to the application layer.
    - OVirtualRobotComm – Handles the sensors, effectors (joints and LEDs) and camera.
    - OVirtualRobotAudioComm – Handles the audio communication, both speakers and microphones.
    - ANT object – Handles the TCP/IP communication.

- **The application layer** is where a program is written by the user. It uses the system layer interface to access the robots hardware.

The system layer's services, that get data from the application layer, send an event indicating to the object that they are ready to receive more data once they have handled the last sent data. The services that send data to the application layer send an event indicating to the object that they have sent the required data.

The system layer contains OPEN-R modules which are called objects. These objects are different from standard C++ objects, but there are similarities. They have the same basic functionality as a C++ object and some extensions, like multiple entry points. The modules communicate through messages. These messages are passed through inter-object communication, which uses pre defined communication channels. The channels are defined in a configuration file. The module that sends the message is called subject and the module that receives the message is called observer. The message can contain any C++ type data and an identifier that identifies which method should be executed when the message arrives. A module can have any number of subjects and observers implemented but it can only process one message at a time. This is because every module in OPEN-R is single-threaded. Thus every module must have its own message queue.

Every OPEN-R program contains one or several concurrently running modules. The fact that the system is modularized makes it easy to replace a module without recompiling the whole program. OPEN-R also includes Remote Processing Utility which makes it possible to execute parts of the program on a different platform, e.g., a desktop computer.

2.4 Tekkotsu

Tekkotsu[6] is an application framework for robotic platforms (at this point AIBO), handling low level and routine tasks. It is written in C++, like the underlying OPEN-R and makes full use of inheritance and templates. Figure 3 gives a schematic overview of the framework.

Tekkotsu is an open source project developed at Carnegie Mellon University USA. The name means "iron bones" in Japanese which is a metaphor for structural framework. The aim of using the Tekkotsu platform was to avoid demanding low-level programming and being able to concentrate on the task of programming sophisticated behaviors instead.

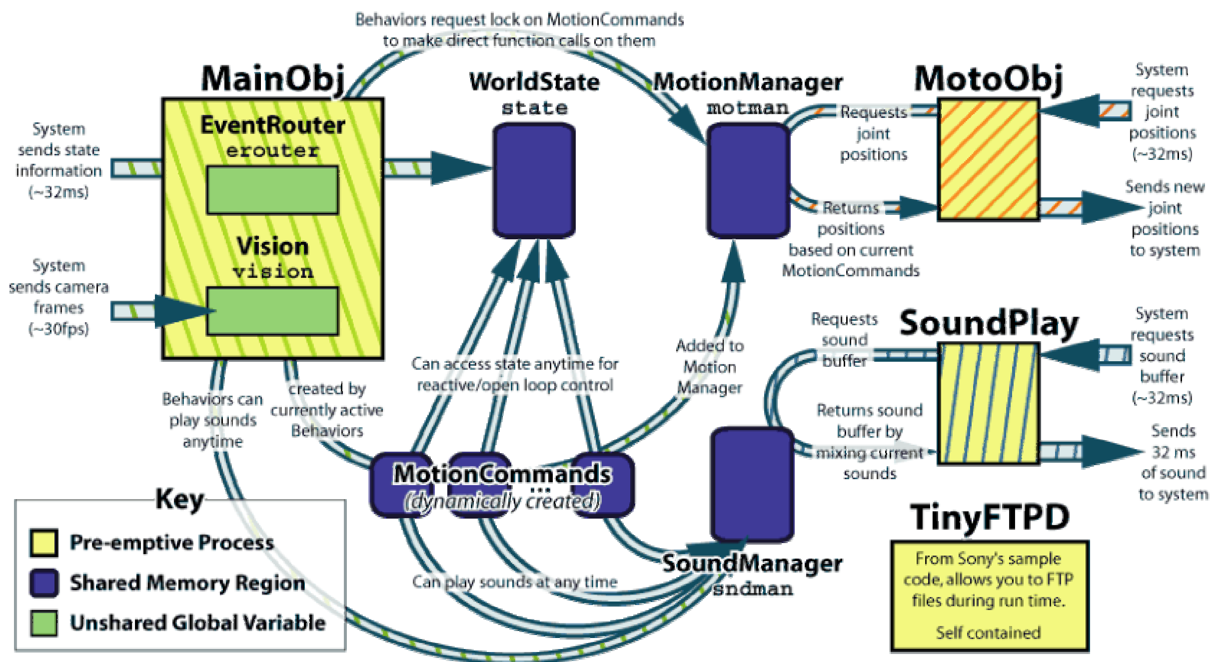


Figure 3. The Tekkotsu framework

Tekkotsu is an object oriented and event passing architecture, similar in design to Java. Three system processes (OPEN-R objects) are being used:

- **MainObj.**
  Most of the bulk processing such as vision, decision making, tracking state of the world.
- **MotoObj.**
  Updating positions of joints and values of LEDs
- **SoundPlay.**
  Mixes and sends sound data to the system to ensure playback doesn't stutter.

In addition to the processes, there are some important structures to consider:

World State is a shared memory region where the state of the dog at the moment is stored. The variables held in World State are joint positions, joint torques, button status, power status, IR distance, accelerometer, temperature readings, LED values, PID settings, and ear positions.

Motion Manager provides simultaneous execution of several motion primitives at the same time. Motion primitives are shared memory regions based on MotionCommand.
The MotionManager handles the work of making sure the Main and Motion processes have mutually exclusive access to the MotionCommands.

Motion Commands are the classes providing functionality. For us the most important classes are walkMC, which provides a certain walking style, and HeadPointerMC which allows us to set the angles of the head. It is also possible to create new MC:s for other applications. Figure 4 illustrates the MotionCommands hierarchy.

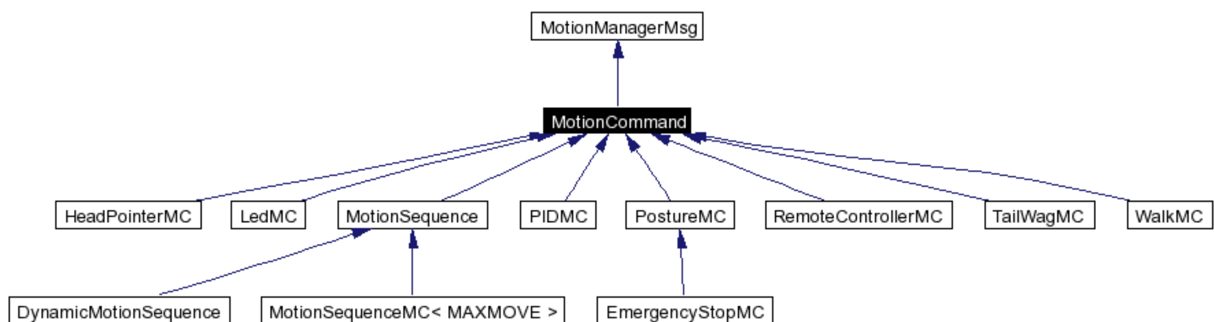


Figure 4. The Tekkotsu MotionCommands hierarchy.

Sound Manager provides easy methods for playing back sounds, either from files on the memory stick, or from dynamically generated buffers.

Event passing. The EventRouter manages distribution of messages to listeners. It is globally accessible so that you can register to listen for events or throw events from anywhere in the code. Timers are simply another event generator, and are handled as any other event.

3 Robocup

RoboCup is an international project to promote AI, robotics, and related field. The competition is intended for a team of multiple fast-moving robots under a dynamic environment. The ultimate goal of the RoboCup project is:

By 2050, develop a team of fully autonomous humanoid robots that can win against the human world champion team in soccer.

To be successful in this competition, a team must incorporate various AI related technologies like: design principles of autonomous agents, multi-agent collaboration, strategy acquisition, real-time reasoning, robotics, and sensor-fusion.
There are three major domains of RoboCup.

- The **RoboCupJunior** consists of a soccer challenge, a dance challenge and a rescue challenge. It is primarily intended for young students competing with simple robots.

- The **RoboCupRescue** is an effort to develop efficient search and rescue robotics in large scale disasters. It demands heterogeneous agents that can perform activities like long range planning and emergent collaboration. There is also a logistic factor in this branch. It is based on two parts:

  - Rescue Simulation League
  - Rescue Robot League.

- The **RoboCupSoccer** contains five different leagues:

  - Simulation League — Computer simulated soccer game.
  - Small Size Robot League — Robots that has a maximum diameter of 18 cm and 15 cm in height.
  - Middle Size Robot League — Robots has a maximum diameter of 50 cm and 80 cm in height.
  - Four Legged Robot League — AIBO robots.
  - Humanoid League — Human-like robots.

This report is only concerned with the Four Legged Robot League so it is the only part that will be discussed in the rest of this section.

3.1 Four legged Robot League Championship

In the Four-Legged Robot League, teams of autonomous AIBOs are competing against each other in a soccer game. The soccer game takes place in a well defined soccer field which is illustrated in figure 5. The field is 4.2m x 2.7m wide. There are four members in each team, one goalkeeper and three field players. A team is always either blue or red and the nets are colored sky-blue or yellow. There is a landmark in each corner, which has different colors depending on in which corner it is. The color of the landmark corresponds to which net it is next to and if it is on the left or right side of the net. This is to make it easier for the robots to determine their position in the field.

A soccer game is divided into two halves with a break between. One half is 10 minutes and the break is 10 minutes. If the game is a draw there will be 5 minutes of extra time where the team which scores the first goal wins (Golden goal). If no team scores the golden goal, then the winner will be determined by a penalty shoot-out. Most of the basic rules in the game are equivalent to the ones in human soccer, but there are some differences. This is because the AIBOs don't have the same physical, mobile and mental, capabilities as humans.
The differences are in movement and ball handling.

- Movement
  - Pushing – It is not allowed to push another robot for more than 3 seconds
  - Obstructing - It is not allowed to intentionally block the way to the ball for a robot of the other team,
  - Penalty area - It is not allowed for a field player to have more than two legs inside the penalty area.
- Ball handling
  - Ball holding – It is when a robot doesn't leave one successive open area that covers more than 180° around the ball. A goalkeeper may hold the ball for 5 seconds as long as it has at least two feet in the penalty area. The rest of the players may hold the ball for 3 seconds.

When a robot commits a foul, it is removed from the field for a period of time, usally 30 seconds.

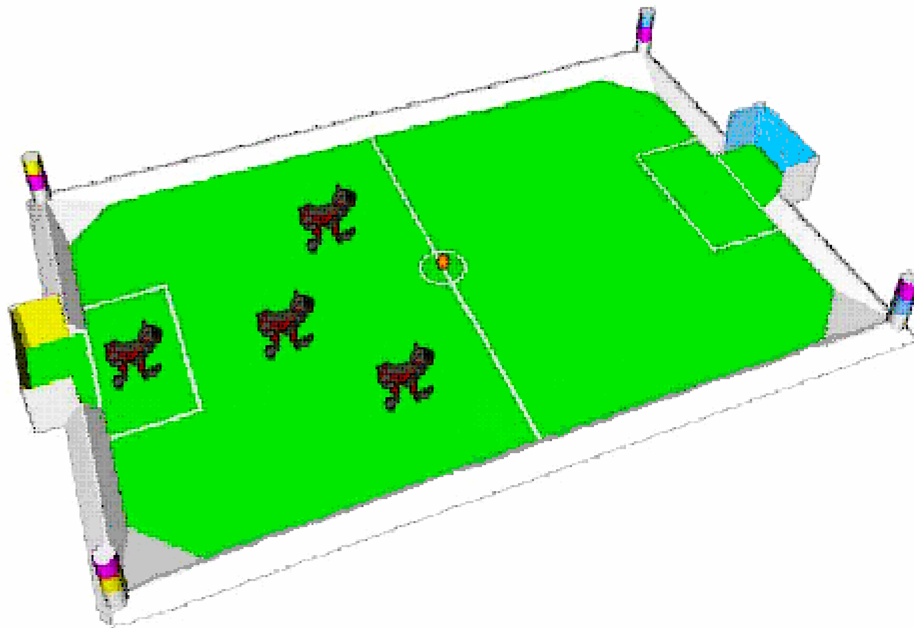


Figure 5. Four-legged Robot League soccer field.

3.2 Four-legged Robot League Challenges 2004

In addition to the Championship there are three different challenges for the teams to participate in. Each of these challenges targets a specific problem area within AI robotics, such as vision, localization and cooperation.
The challenges in 2004 were:
1. The Variable Lighting Challenge.
2. The almost SLAM Challenge.
3. The Open Challenge.

The sole purpose of these challenges is to take RoboCup to the next level, which means that all efforts to reduce the difference between real soccer and RoboCup soccer will be tried in a challenge before they are included in the game.

3.2.1 The Variable Lighting Challenge

This challenge is intended to encourage teams to increase the robustness of their vision to lighting changes. It is based on a penalty shoot out. The robot must score as many goals as it can into the yellow goal in three minutes. The team that scores the most goals wins.

As well as the single blue robot, there are two red 'opposition' robots on the field. Both of these robots are paused; frozen in the 'UNSW stance'. One robot is placed in a goalie position on one side of the yellow goal. The other is placed in the third of the field nearest the yellow goal, at least 30cm away from the edge. There is a single ball upon the field. Initially it is placed in the centre kickoff position. Upon each score, the ball is moved back to the centre kickoff position. The robot is not moved by the referee and must make its own way back to the centre of the field to reach the ball again. The lighting for this challenge is different from standard robocup lighting. Some additional lights are brought in to supply variable lighting. These additional lights are of deliberately unspecified color temperature. Lights may also be covered to achieve variable lighting conditions. The changes include periods of relative stability in the lighting, periods of slow change in lighting and periods of fast change in lighting. It is envisaged that the additional lighting will be non-uniform across the field and hence the lighting changes will be non-uniform.

3.2.2 The almost SLAM Challenge

The almost SLAM challenge is intended to help the league move away from strictly defined beacons (which are the ordinary landmarks in the field) to more generic localization information such as the various stands in a soccer stadium. In order to achieve this, additional landmarks are placed around the fence on a robocup field. The challenge is divided into two stages. In the first stage, all normal landmarks and the additional landmarks are uncovered. The robot has then one minute to explore the field. In the second stage, the normal beacons and goals are covered up or removed, and the robot must then move to a series of points on the field. When it is close to a point, the robot will pause itself and indicate that it believes it to be near a point (by wagging its tail). The second stage ends when the robot has had two minutes, or when it has stopped five times. Between the two parts of the challenge, the referee covers or removes all of the normal landmarks and goals.
At the end of the second stage, the points are calculated. The teams get points for accuracy and speed.

3.2.3 The Open Challenge

This challenge is designed to encourage creativity within the Legged League, allowing teams to demonstrate interesting research on autonomous systems. Each team is given three minutes of time on the Robocup field to demonstrate their research. The winner will be decided by a vote among the entrants.

4 Team Chaos Challenges Framework

The new AIBO model had several features that were different from the old model and these made it necessary to develop a new software framework. This new framework should have several key features.

- **Exchangeable modules** – It should be easy to replace a module with another one.
- **Fast** – The pictures should be processed at a rate of at least 10 frames per second.
- **Robust** – Frame rate, delays, speed of the robot, etc should not affect the framework.

The framework that was designed is based on the OPEN-R SDK, Aperios Operating System and Tekkotsu. The relationship between these is illustrated in figure 6. Tekkotsu was used because it provided a simple interface to the robotic hardware. Tekkotsu also had several walking styles and communication implemented.

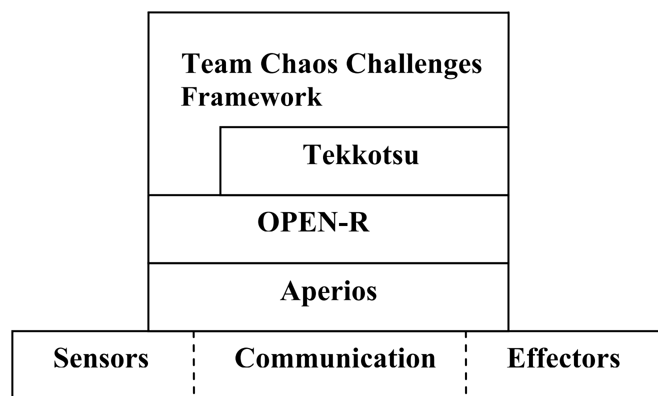


Figure 6. Relationship between TCC framework, Tekkotsu, OPEN-R, Aperios and AIBO.

The implemented framework consists of six modules: Tekkotsu, Vision, Localization, Wireless, Behavior and Worldstate. It can be seen in figure 7. Tekkotsu, Vision, Wireless and Behavior operate in a Token Ring architecture and these modules exchange information from Worldstate. The Localization object is separated from the control loop and it is accessed at a slower rate than the rest of the framework. This is because it does not need to be active all the time, since the robot can not move more than about 2 cm during a turn in the framework

These modules are implemented as three OPEN-R objects, Tekkotsu, Vision, Wireless and Behavior as one object, Localization as one object and Worldstate as one. This is to minimize the message passing between the objects, since this slows down the framework.

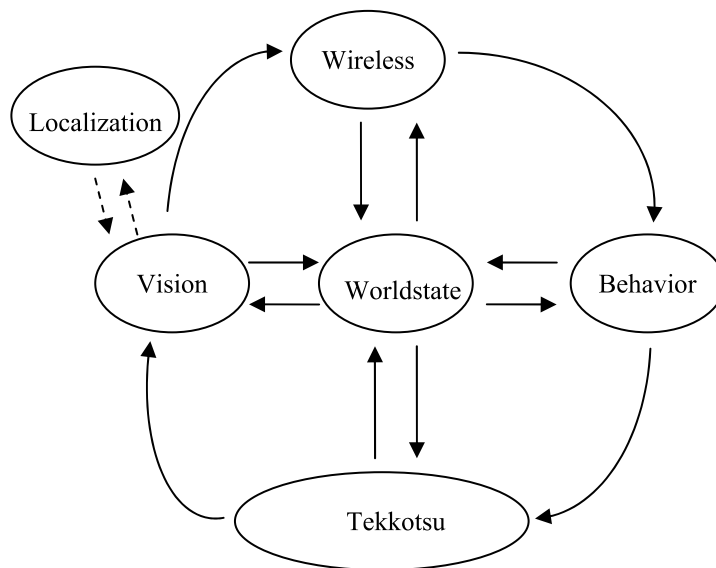


Figure 7. The Team Chaos Challenges framework and program structure of the AIBO.

4.1 Vision

This module gathers information from pictures that the camera takes. A picture is at the highest resolution 208x160 pixels. Vision is a pipeline that starts with a raw picture from the camera and ends with a list of objects that it has found in the picture. The vision pipeline consists of two parts, segmentation and object recognition. First in the pipeline is the segmentation, where every pixel in the picture is segmented according to its color. The segmentation uses a predefined color table which holds the different values a specific color can have. The color tables are made from pictures taken on different object in the environment. These tables are very sensitive to the lighting conditions in which they were made. If the lighting has changed too much since then, the segmentation will not work anymore, or at least not as good. In most cases there are some pixels that are segmented wrong. This happens because it is almost impossible to make a perfect color table. The segmented picture results in a table with small blobs containing different colors. Object recognition grows these small blobs that were generated by segmentation into bigger ones by comparing it to its neighbors. When there are no more blobs with a similar colored blob next to it, the blobs are sorted into different objects. This is done by looking at the different attributes that an object can have. Attributes that are used are color, shape, size, position and color-neighbors. The identified objects are then updated in the WorldState.

4.2 Localization

Localization keeps track of where the robot is in the field. This is done by triangulation based on the distances and angles to the landmarks and nets. These objects are used, since they have fixed position. The robot has to see at least two objects to be able to locate itself. The distance and angle measurements are not very accurate, and can vary even though the robot does not move. This results in instability of the localization. To reduce this problem a history of the

dogs' position can be implemented and by filtering these values a more stable estimation of the position can be done.

4.3 Wireless

This module is responsible for the communication between AIBO robots. It can also communicate with an external computer. This makes it possible to send pictures from the camera, both raw and segmented, and the values of the variables in the Worldstate to a separate computer. Also it is possible to set variables in the Worldstate from a computer. This is very useful during the construction and testing process. The communication uses TCP/IP, which makes it possible for two hosts to exchange streams of data. The module can send one message per connection and frame.

4.4 Worldstate

This is where all common information is stored. It is a global shared memory area that every module has a pointer to, and hence can access it. Worldstate contains the latest picture, both raw and segmented, a list of objects that might appear in the field, communication information and information about movement, e.g., heading or which kick should be performed. Objects that are interesting are robots (self, team members and adversary), the nets, landmarks and of course the ball. Every item in this list has attributes that describes the object, e.g., position (global, and local) or distance, angle, etc.. These variables are updated by other modules.

4.5 Behavior

The behavior module is the "brain" in the framework. This is where all decisions are made except for the head movement that is decided in the Vision module. This is also where outputs to the joints and LEDs are being set. In every cycle, considering the world state, Behavior tries to calculate what the next world state should look like, getting one step closer to the desired goal. The goal is arbitrary and decided by the present application. There are many ways for the "brain" to figure out how to achieve the goal. Some approaches to this problem are presented in chapter five. The implemented approach is discussed in chapter six.

4.6 Tekkotsu Interface

This is where the loop starts and ends. The Tekkotsu Interface is the interface to the Tekkotsu framework. Tekkotsu, in turn, is the indirect interface to the hardware of the robot. The interface writes and reads variables from Worldstate and updates Tekkotsu. Information that is exchanged is the picture, motion information, LEDs and joints information. When the loop starts, Tekkotsu updates the picture in the Worldstate and passes the control to Vision. When the control returns to Tekkotsu, the Worldstate in most cases has been modified. The Interface then reads Worldstate and updates Tekkotsu.

5 Behaviors

What is a behavior? What a behavior is lies to a great extent in the eye of the beholder. A behavior can be taking a step, or it could be walking toward an object. It can even be walking to the coffee machine, pouring up a cup of coffee and walking back to the chair again. Or just not doing anything at all. A problem is to determine how abstract we want a behavior to be.

In other words there are meny meanings of the word beavior. In the encyclopedia it is divided into two categories, the physiological and the zoological aspect. In our case the zoological aspect fits better. It explains the word by: ”The term generally refers to the actions or reactions of an object or organism, usually in relation to the environment or surrounding world of stimuli”… ”the sum of all motions, gestures and signals, both visible and unvisible, created by the organism”. The environment for the AIBO is the soccer field and possibly also the surrounding area, while the stimuli is the information given by the camera and sensors. An action, or reaction, is for example to move a leg or, more sophisticated, to walk.

For the AIBO the behavior expression corresponds to something meaningful. An action like move a leg a few millimeters can hardly be called a behavior if it does not accomplish something. A behavior in our case might be an action but is more often represented by a sequence of actions. An example of such a sequence could be finding the ball somewhere in the field, which requires the actions of moving the legs in a specific way and turning the head, scanning for the ball. However we do not want it to be too complex either.

The challenge with programming behaviors is to decide under what circumstances a specific behavior should or may be used, or which behavior should be used under certain circumstances. Since it would be impossible to consider all possible reactions to different situations, we prefer the first alternative. Thus, we have several behaviors to deal with and we must decide when to use them.

There are several different methods to use in decision making for autonomous agents. The methods differ in complexity, speed and robustness. We have looked at a couple of them and implemented the one that we thought was the best given the circumstances.

Zoological (and robotic) behaviors can be divided into three categories, which also might be viewed as levels of abstraction. Reflexive[7] behaviors are directly generated on the basis of the sensor input, without volition or conscious control. An example is the automatic shrinking of the pupils of the eye in strong light. Also reactive behaviors are used without conscious thought, but unlike reflexive behaviors these are learned. Activities like biking and skiing are examples of learned behaviors that we don’t bother our minds with when performing, however it is not a reflex. Third and most complex category is the conscious behaviors. As one can tell by the name these behaviors are performed after consideration, like for example when playing a game or writing a paper. Combining other behaviors is also an example.

Robotic paradigms

Historically, different approaches have been used to simulate intelligent behavior in robots. Three more important structures or “paradigms” have been used since the birth of AI for robotics, namely hierarchical, reactive and hybrid deliberative / reactive, see figure 8.

The *hierarchical paradigm (a)* was the first to occur in the late sixties. It was based on the then leading theory of how people think. “I see the coffee machine, I plan how to get to it, I walk”.

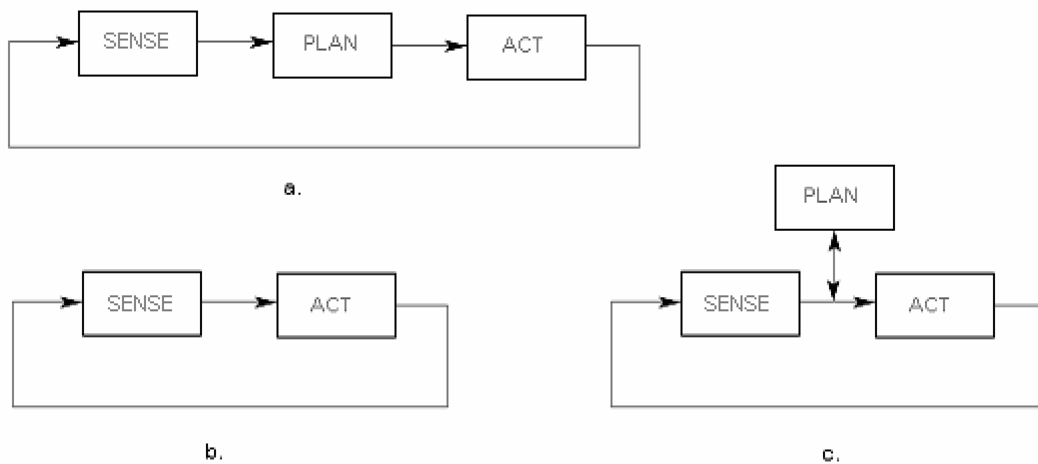


Figure 8 Three paradigms: a) Hierarchical, b) Reactive, and c) Hybrid deliberative / reactive.

The main drawback with the hierarchical approach is that the planning part tends to become too time consuming, slowing down the whole cycle, which can be devastating for real time systems. To cope with this problem the *reactive paradigm (b)* in 1988 [8] threw out planning altogether. The approach directly translates sensor inputs into activities, which is very effective. By combining readings from several sensors by different actions, a complex behavior can be achieved. But throwing away planning was a bit drastic and 1990 a new approach called *hybrid deliberative/reactive paradigm(c)* was introduced. Under the hybrid paradigm, the robot first plans how to best decompose a task into subtasks and then what are the suitable behaviors to accomplish each subtask. Then the behaviors are executed as per the reactive paradigm. Planning, which is the computationally expensive part is executed much less often than the reactive part.

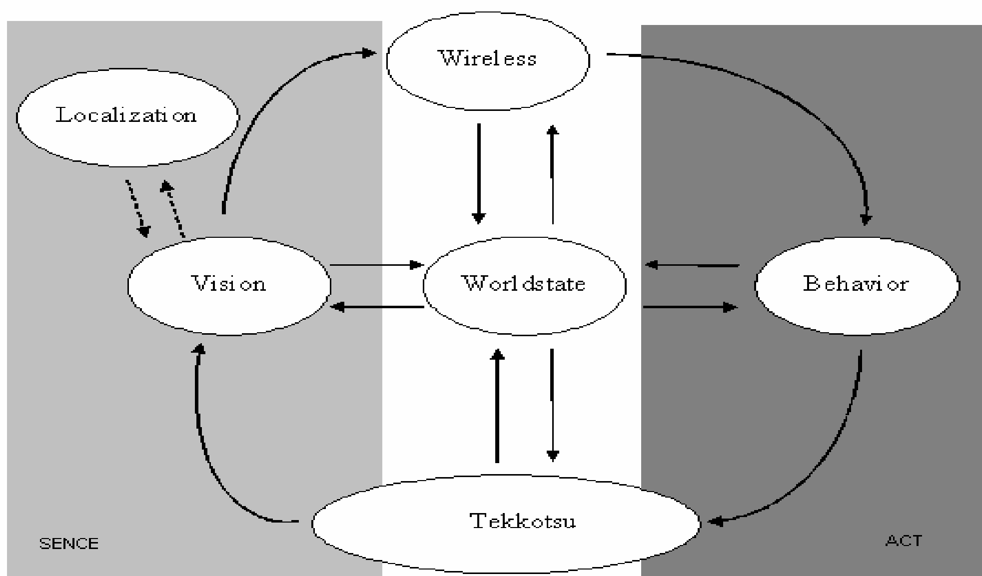


Figure 9 The Team Chaos framework illustrated with the reactive paradigm.

5.1 Decision Trees

A decision tree[9] is a hierarchal structure. The tree is traversed from the top and downwards until a leaf is reached. The decision tree takes as input an object or a situation and returns a decision – the predicted output value for the given input. A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached.

When a decision tree is used by robots for decision making, then a leaf corresponds to a simple behavior and the higher up in the tree that a node is the more complex is the behavior. An example of a decision tree for scoring a goal is illustrated in figure 10. The tree decides how to act when the robot wants to score a goal. This tree can be one of many behaviors used in a even more complex structure, where scoring is on the same level as for example passing, and the decision on what to do is made in the level just above. Passing would then have a similar structure, with different sub nodes and leafs.

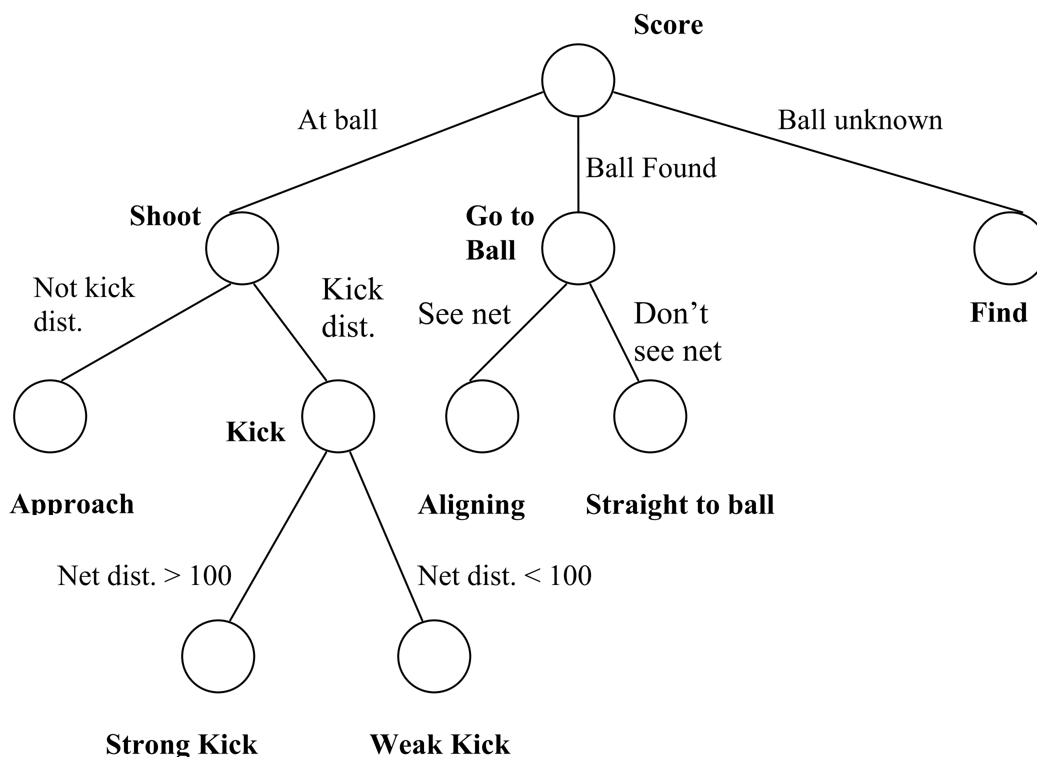


Figure 10. Example of a decision tree.

5.2 Planning

Planning is the task of coming up with a sequence of actions that will achieve a goal[7]. Different planners are used depending on the environment in which they are supposed to act in. One could roughly divide the environment into two types, static and dynamic. The environment in the challenges is static, since the robot is the only one that acts in it, and it is dynamic, e.g., in the soccer games.

A planning system consists of a language that describes the environment and the goal, and an algorithm that finds the solution to the task. Planning languages represent a planning problem using states, goals, and actions.

- A state is represented as a conjunction of positive literals.
- A goal is a partially specified state.
- An action is specified in terms of the preconditions that must hold before it can be executed and the effect that occurs when it is executed.

Examples of planning languages are the STRIPS and ADL language. The STRIPS language describes actions in terms of their preconditions and effects and describes the initial and goal states as conjunctions of positive literals. The ADL language is a variation of the STRIPS language and it relaxes some of theses constraints, allowing disjunction, negation and quantifiers.

Consider the problem of going to the ball and kicking it into the blue net. The ADL description of the problem would look like:

Init(At(ball, field) ∧ See(robot, blueNet))
GOAL(At(ball, blueNet))

Action(GoToBall(ball, field)),
  PRECOND: At(ball, field)
  EFFECT: ¬At(ball, field) ∧ At(ball, robot)
Action(Kick(ball, blueNet)),
  PRECOND: At(ball, robot) ∧ See(robot, blueNet)
  EFFECT: ¬ At(ball, robot) ∧ At(ball, blueNet)

There are basically two different algorithms to use when solving the planning problem; State-space search and Partial-order planning. Subgoals can be used to get more effective heuristics.

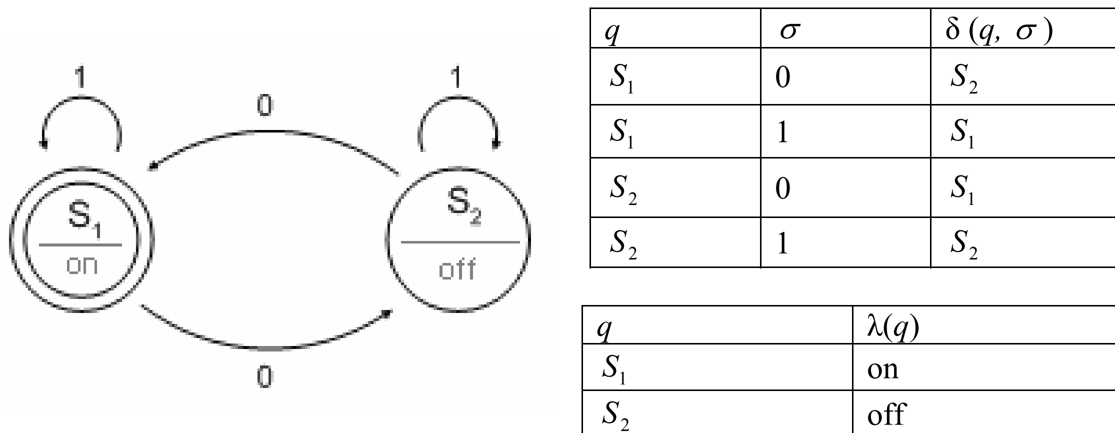
- State-space search can operate in the forward direction (progression) or the backward direction (regression).

- Partial-order planning (POP) explore the space of plans without committing to a totally ordered sequence of actions. They work back from the goal, adding actions to the plan to solve a subgoal.

5.3 State machines

A Finite State Machine (FSM)[10] is a model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state by reading an input string. It changes to new states depending on the transition function. There are many variants of FSMs, for instance, machines having actions (outputs) associated with transitions (Mealy machines) or with states (Moore machines), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (nondeterministic finite state machines), one or more states designated as accepting states (recognizer), etc.

To visualize an FSM it can be written as table or drawn as a state diagram. Figure 11 is an example of a Moore Machine.



q	σ	$\delta(q, \sigma)$
S_1	0	S_2
S_1	1	S_1
S_2	0	S_1
S_2	1	S_2

q	$\lambda(q)$
S_1	on
S_2	off

Figure 11 Example of a Moore machine; state diagram and table.

A Moore machine M can be defined as a tuple M = { Q, Σ, Δ, δ, λ, q_0 } consisting of

- a finite set of states (Q), each state $q \in Q$;
- a finite set called the input alphabet (Σ), each input $\sigma \in \Sigma$;
- a finite set called the output alphabet (Δ), each output $\delta \in \Delta$;
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$) mapping a state and an input to the next state;
- an output function ($\lambda : Q \rightarrow \Delta$) mapping each state to the output alphabet;
- a start state (q_0).

For the example above the set of states $Q = \{S_1, S_2\}$, input alphabet Σ = {0,1}, output alphabet Δ = {on, off} and start state $q_0 = S_1$. Translation function and output mapping can be seen in the tables.

This simple machine switches something on or off when a 0 is received. For example if 1101100 is an input string the following would happen: Machine is turned on and the start state S_1 is entered.

time	q	σ	Δ
1	S_1	1	on
2	S_1	1	on

3	S_1	0	on
4	S_2	1	off
5	S_2	1	off
6	S_2	0	off
7	S_1	0	on

Then every time a 0 is received, the machine will change its state and output. As long as the input is 1 the machine will remain in the same state.

5.4 Hybrid automata

Hybrid automata are generalized finite-state machines for modeling hybrid systems [11]. A hybrid system consists of a discrete program within an analog environment. The discrete transitions of such a program are modeled by a change of the program counter, which ranges over a finite set of control locations. For each control location, the continuous activities of the environment are governed by a set of differential equations. Each location is also labeled with an invariant condition that must hold while the control resides at the location, and each transition is labeled with a guarded set of assignments.

A linear hybrid automaton is a hybrid automaton that has linear differential equations and a special case is a timed automaton. In a timed automaton each continuously changing variable is an accurate clock whose rate of change with time is always 1. Furthermore, in a timed automaton all terms involved in assignments are constants, and all invariants and guards only involve comparisons of clock values with constants.

The practical use of hybrid automata is that they can be used for proving that a control program will work correctly, and there is no possibility that the program will jump into an illegal state. There are two semidecision procedures for verifying safety properties of a linear hybrid automaton; fixpoint computation and minimization. The two procedures are based on generally infinite state spaces. If the procedures terminate, then they will give correct answers.

The state transitions must be guaranteed, either by the state machine or some underlying software. This is to avoid that for example one state does not leave a leg in the air, and the next state assumes that the leg is on the floor. Because if this was to happen then the system would probably not work since the robot would not be able to walk. In our case the state transitions are handled by the state machine and the Tekkotsu application framework. Tekkotsu handles the positions of the joints and makes sure that they always move smoothly and the state machine handles the objects with which the robot interacts, e.g., the ball.

The hybrid automata and the procedures for verifying their correctness that have currently been developed are too simple to apply on a complex machine such as the AIBO robot. To prove the correctness of a system like an AIBO robot and its enviroment, methods and algorithms has to be developed further before this is possible.

6 Behaviors in TCC

These are some important criteria that we have taken into account while choosing our approach.

- **Modularity**
  How the behaviors can be used at the same time to create a more sophisticated behavior. The ability to reuse and create new behaviors out of the existing ones.
- **Niche targetability**
  How well the solution works for the intended application.
- **Robustness**
  Smooth transitions between the modules and prevention of deadlocks.
- **Completeness**
  Is everything in the structure covered?

One of the ideas behind this project was to write reusable code that was general enough to be used in many different applications. For the Championship we identified some important behaviors. One of those was to score. The natural point of attack now was the divide-and-conquer method. As we analyzed the procedure of scoring we found some lower level behaviors that we considered to be usable in many approaches. Our solution was to use these simple behaviors to construct more complex ones. The question was which method to use to combine the simple behaviors.

One criterion with reusable code is that the interface is clear to later users. The code should also be simple so that users easily can overview it and understand it with no great effort. For this we needed a *simple structure*. If the structure was to complex no one would care about it. In a dynamic environment like a soccer game, planning a path, a shot or some other action is not easy. With this in mind we were afraid that a planner would make the structure far to complex. Also for the code to be *robust* it should not be too complex. Nevertheless it must guarantee to care about any type of condition and possibly make something meaningful out of it too. A deadlock is a disaster and must be prevented at any price.

This is a probable scenario: We would like to go to the yellow goal. We would also like to avoid bumping into things on the way there. Then we would like to look for the ball at the same time. This means we want to do several things at the same time, not always agreeing with each other. For example we can not focus on the goal and the ball at the same time. We can not approach the goal, at the same time trying to avoid it. We can not have two parallel behaviors, one telling the legs to move straight on, and another telling them to keep left. We have to *avoid conflicts* between behaviors. When we studied the decision tree approach we came to the conclusion that there was a risk for the behaviors starting oscillating between each other. In one moment one leaf would be reached and in another moment another leaf.

6.1 Using finite state machine approach.

We came to the conclusion that the state machine approach best meets our requirements. The state machine approach is part of the reactive paradigm and is commonly used for robotics and computer science. However it does not mean that there is nothing more to it.

In a finite state machine the states represent the present condition or the content of memory at a certain time. For a Moore machine the state produces an output, in our case the corresponding behavior. As we have chosen to use state machines, we call our lowest level behavior *states* and the higher level behaviors *machines*. In the machine we combine the states in certain ways. It is also possible to combine the machines into higher level machines, and then you can view the machine as kind of a state.

Thus the states represent behaviors, or give a behavior as output. Input to the states is the refined perception information, that is, the camera and sensor information. The transition function is a combination of pre and post conditions. Output is information about actions. These will be explained later.

6.1.1 States

How shall we know when to use a specific behavior? Of course we must have a certain goal to reach for, but then we have to know in what order to execute our possible behaviors. If we, for example, are about to score a goal and are placed a meter from the ball there is no idea using the KickBall behavior. We have to look at what the world looks like, or what is in our memory at the moment. As our memory is not very well developed, we can just remember what recently happened, which we call the world state. The world state is how the dog understands the world in this very moment. For the AIBO the world is limited to what the camera and sensors detect and it decides what to do depending on these factors. If the *world* is the set of *world states* that could possibly occur to the dog, then a *state* is a subset of the world. A set of world states corresponds to a certain state.

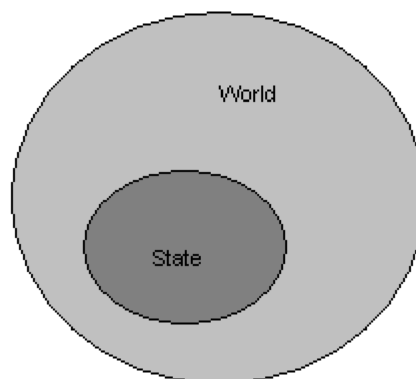


Figure 12. ***World State* $\in$ *World, State* $\subseteq$ *World.***

A problem is to determine how abstract a behavior should be for a state. The fact that a behavior can be almost anything makes it necessary to delimit it somehow. Since we use a Moore machine, a state output is a function of the input. The output that the states produce is reactive. This means that there is no possibility to change output function within the state. The

output function is only changed when a new state is activated. This leads to a natural delimitation among the lower level behaviors (that we call states). Walk to that point, turn around, go between, etc., can all be implemented as reactive behaviors while Score would be almost impossible. This can be seen as a limit “upwards”. The fact that we used Tekkotsu also contributed to the reason why we chose the limits the way we did, seeing that it already implemented the lowest level of actions, like walk, spin and move head, for example. This was our limit “downwards”.

As part of the reusable code idea, the intention was to let the states be exchangeable objects which could be combined as one liked and with which one could build any state machine one wished, the states would just be like building bricks. The aim with this was to take us one level higher, so that users could focus on other, more interesting issues, using our states.

Then the question was how the user interface should look like. As we reflected on the question of usability versus functionality, we came to the conclusion that usability was more important in this application since too complex code would make people uninterested. This means that it is rather easy to build new machines out of the states, but the states are unfortunately not very flexible. The interface is presented in the appendix.

When machines are built out of states one has to care about in which order they are put together. For one state to fit to another, the transition has to be smooth. For example there is no meaning putting together state findObject and state Kick. Between them one has to use the state goToObject followed by approachBall. In other case the dog would be stuck doing nothing after findObject.

What would happen if the goal for a state is not possible to fulfill? Unfortunately nothing. The dog will infinitely keep on trying to reach the goal. For example, the turnToObject state will never end if there is no such object and goToObject is never ending if the object is blocked.

A special behavior is the avoidance behavior. It serves as a reflex function for the dog and signals when it is moving too close to something. As this function is always active it has to be running at the same time as the other behaviors. As mentioned earlier, parallel behaviors can be a difficulty due to the problem of choosing between them in the end. The solution to this specific case was to let the avoidance behavior filter the outputs from the states, as illustrated in figure 13.

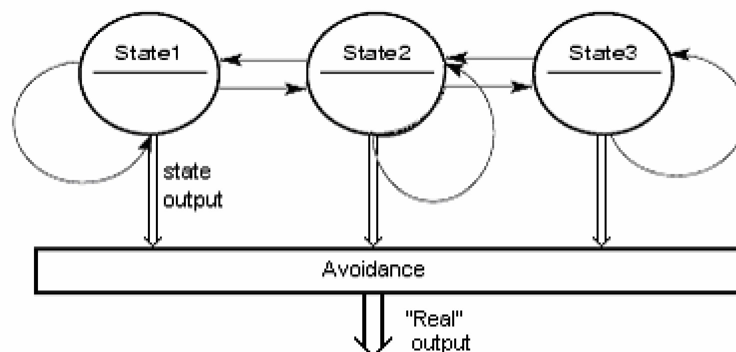


Figure 13. Filtering of the outputs through the Avoidance behavior.

If a certain state is active and the sensed world state lies within the boundaries of this state, the output behavior will continue.

6.1.2 Input alphabet

World State is not just a state in time but also our physical memory. It is a global shared memory area containing variables updated by the sensing mechanism in the TCC framework. This is also where we get our input from. Actually our input alphabet could be defined as anything from our world state, but for our needs we have used the following variables:

.

- **Theta**
  The horizontal angle to an object in radians.
- **Epsilon**
  The vertical angle to an object in radians.
- **Distance**
  Distance to an object in mm.
- **Confidence**
  How certain is it that the theta, epsilon and distance inputs is correct
- **Timer**
  A timer in ms telling us for how long we have been doing certain things.

6.1.3 Output alphabet

The AIBO moves by updating the angle for a set of joints for every cycle of the framework. The angles are handled by Tekkotsu which also takes care about the movement of the head. For the states to access these joints, variables have to be set in the World State. Following variables can be set in the World State to operate the dog:

- **Alpha -** The direction of the dog.
- **Spin -** How much it turns.
- **Speed -** The speed.
- **Tilt -** To bend the neck up and down.
- **Pan -** To turn the head left and right.
- **Nod -** To bend the head up and down.

6.1.4 Output function

To transform input into output, each state has its own output function. As we are enabled to move both head and legs at the same time (due to coordination done in Tekkotsu), there might be two functions in each state, but since the Vision object is in charge of the head most of the time, there will often be one. Figure 14. Example of an output function. is an example of an output function from behavior to Tekkotsu.

$$\text{Speed} = \sqrt{\left|\frac{theta}{MAX_THETA}\right|} * \sqrt[2]{\left|\frac{distance - APPROACH_DISTANCE}{MAX_DISTANCE}\right|}$$

Alpha = 0 distance > APPROACH_DISTANCE
1 distance < APPROACH_DISTANCE

$$\text{Spin} = \frac{theta}{MAX_THETA}$$

Figure 14. Example of an output function.

6.1.5 Transition function

The transition function switches states from one to another. For that to happen, two conditions have to be fulfilled. A *precondition* has to be fulfilled for the state to make sense. For example it's no meaning kicking the ball if the ball is on the other side of the field. *Pos condition* is fulfilled when the goal for the behavior, represented by the state, is reached. As figure 15 shows postcondition can be viewed as a subset of precondition. If postcondition is fulfilled, then precondition is also always fulfilled.

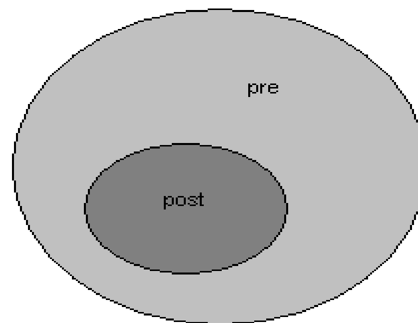


Figure 15. ***Post condition*** $\subset$ ***Pre condition.***

To enter a state, its precondition has to be fulfilled. If pre-, but not postcondition is fulfilled, no transition will be performed, that is the state will remain the same. When both pre- and postcondition are fulfilled, transition takes place and a new state is entered, as figure 16 shows.

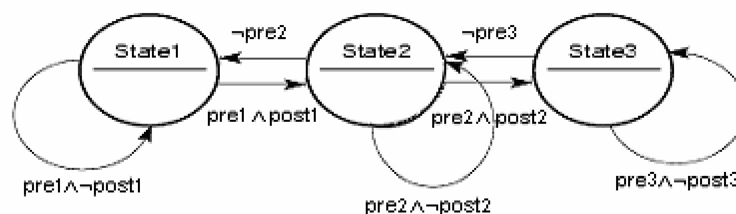


Figure 16. State machine with pre- and postconditions as transitions.

The fact that the goal for a behavior can vary depending on the circumstances is a problem and we have reflected over the possibility of letting pre- and postcondition be an input while creating the states. This is to make the structure more flexible at the expense of cleanness.

However we came to the conclusion that the user interface would be far too complicated in that case. As it is implemented now pre- and postconditions are private constants for each state.

The input controls which state that is active. To change to a new state pre- and postcondition must be fulfilled. Here is a transition function from the code that illustrates how the input affects transition.

- **Precondition:**
    1. `CONFIDENCE_INIT_VALUE` <= *object.Confidence* <= `MAX_CONFIDENCE`.
    2. `MIN_PAN - CAM_HANGLE/2` <= *object.Theta* <= `THETA_INIT_VALUE`.
    3. `MIN_NOD + MIN_TILT - CAM_VANGLE/2` <= *object.Epsilon* <= `EPSILON_INIT_VALUE`.
    4. `DISTANCE_INIT_VALUE` <= *object.Distance* <= `MAX_DISTANCE`.

- **Postcondition:**
    1. `MAX_CONFIDENCE - CONFIDENCE_TOLERANCE` <= *object.Confidence* <= `MAX_CONFIDENCE`.
    2. `MIN_PAN - CAM_HANGLE/2` <= *object.Theta* <= `MAX_PAN + CAM_HANGLE/2`.
    3. `MIN_NOD + MIN_TILT - CAM_VANGLE/2` <= *object.Epsilon* <= `MAX_NOD + MAX_TILT + CAM_VANGLE/2`.
    4. `APPROACH_DISTANCE - DISTANCE_TOLERANCE` <= *object.Distance* <= `MAX_DISTANCE`.

Pre- and postcondition will then be calculated as $(1 \wedge 2 \wedge 3 \wedge 4)$.
To make sure that one state is smoothly transformed into another, we have introduced a mechanism called overlap, see figure 17. One reason for this is to secure that no "gap" between two states exists. Second reason is to minimize the risk of oscillation between two states. Overlap means that, if there are two states mutually connected to each other by transitions, there is a difference between input when entering a new state and input when returning to the one just left. In other words the transition function exhibits hysteresis.

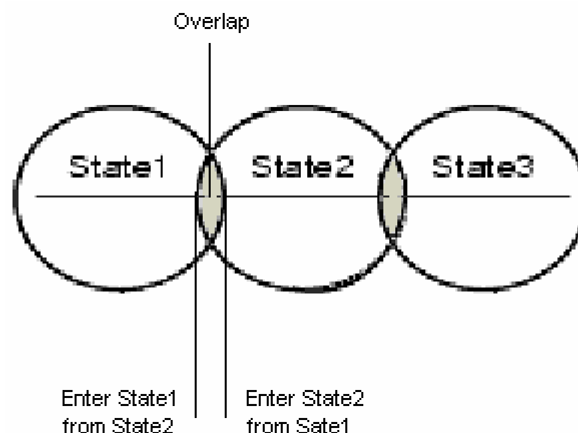


Figure 17. State overlapping.

On a lower level, Tekkotsu takes the responsibility for that joints are smoothly redirected from the angles when one state ends to new angles when the next state takes over.

6.1.6 Start state

The start state is the state entered when the program start executing on the AIBO. It is often illustrated with a double circle in the state diagram. Different state machines can have different start states. However, they must cover all possible world states, i.e., situations the dog can be put into, otherwise the dog will get stuck doing nothing. For example, the KickBall state is a really bad start state because the precondition for KickBall does not cover every situation the dog can be started in. Alternatively, there might be a default state doing something irrational if the start state would fail, but that would be rather meaningless.

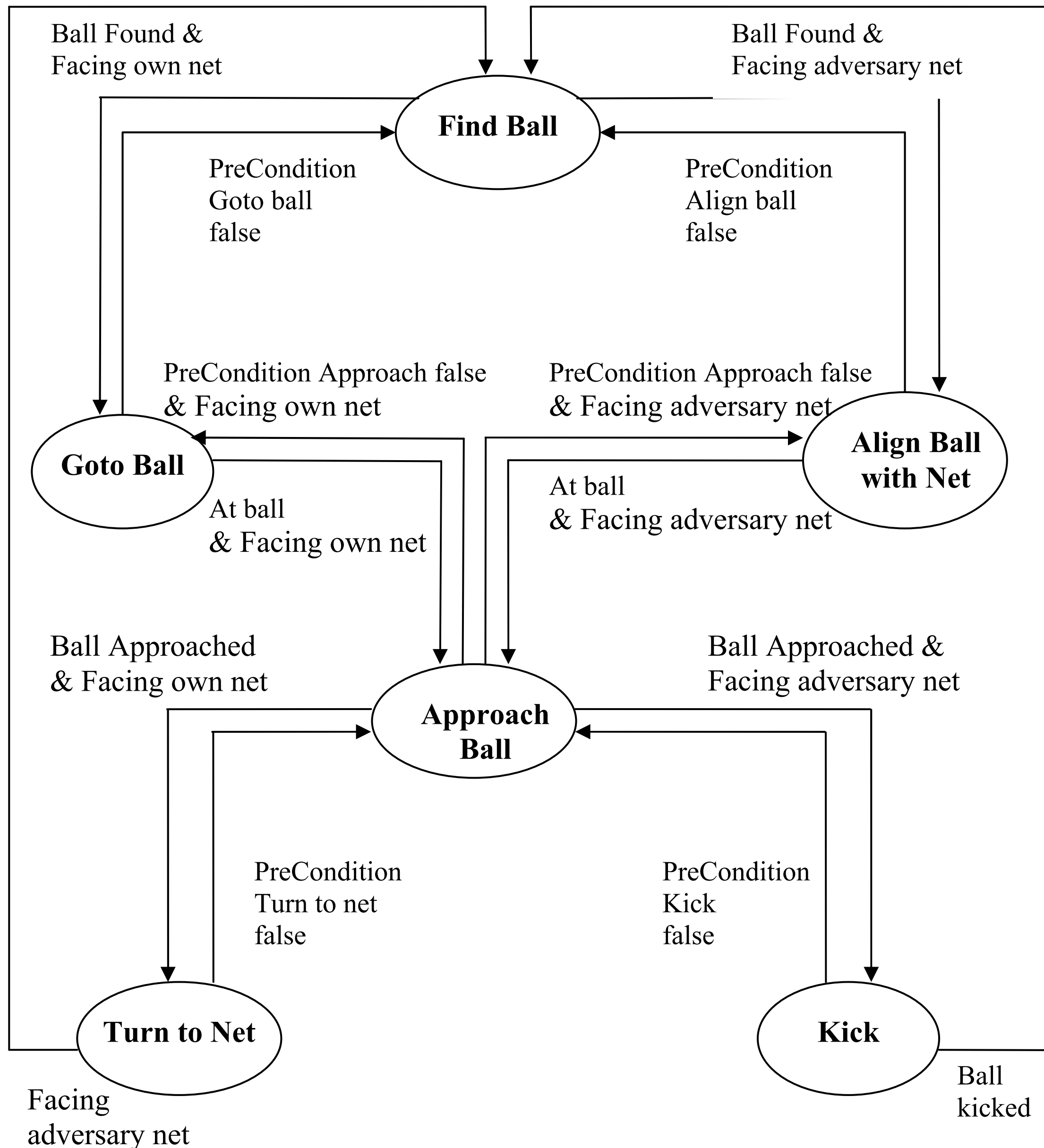


Figure 18. The state machine that is used in the Variable lighting challenge.

6.2 Solving Variable lighting challenge Behavior problem

The state machine that was constructed for this challenge is basically a machine that scores a penalty kick and returns to the start position and tries to score again. After analyzing we have concluded that the state machine should include the following states: Find, Walk, Align, AlignWithBall, TurnToObject, ApproachBall and Kick. The machine was designed by analyzing where the different states could fit in the penalty kick behavior. The resulting machine is displayed in figure 18.

The state transitions that could cause problems in this machine are the *Find Ball* to *Goto Ball* or *Align Ball with Net*, and *Approach Ball* to *Turn to Net* or *Kick*. The difficulty with the *Find Ball* state is that in some cases the dog should walk to the ball and in other it should perform the align behavior. The difficulty with the Approach Ball state is that in some cases the dog should kick and in other cases it should turn to the net. Both these problems were solved by looking at the direction in which the AIBO was facing. If the direction was away from the net in which it would score, the robot should walk to the ball and turn. Otherwise it should align and perform a kick. There are also two opponent robots in the field, one in front of the net, and one somewhere placed in the third of the field near the goal. If the robot was to just shoot at the goal, it would be a big risk of striking one of the other robots, and getting into a problematic situation, e.g., the ball stops just next to an adversary robot and thus makes it hard for the dog to perform the next kick. Our solution to this was to try to shoot around the closest robot and then try to score. This was done by first aiming, with the first kick, towards one of the corner landmarks. Which one depends on where the closest opponent robot is located. Then it approaches the ball and tries to score from that distance. This has not been tested because when this was implemented the AIBO could not detect other robots, but it has been tested by coding the location of the opponent robot into the program.

This is how the state machine works:

1. The dog is placed in the center of the field a bit closer to its own net.
2. The dog starts searching for the ball and adversary robots.
3. The dog aims at the landmark, which is on the opposite side to the closest adversary robot.
4. The dog approaches the ball.
5. The dog kicks the ball with a medium-hard kick. This will move the ball about one meter.
6. The dog then locates the ball again.
7. The dog walks towards the ball and aligns the ball with the net during the same time.
8. The dog approaches the ball
9. The dog kicks the ball with a hard kick. This will move the ball at most two meter
10. If the dog doesn't score, then are 7 – 9 performed again.
11. If the dog scores, then it will then turn 180°.
12. The dog walks to the ball.
13. The dog approaches the ball.
14. The dog turns 180° with the ball.
15. Repeat 2 – 15 again until the time is out.

6.3 Solving SLAM challenge behavior problem

This challenge was divided into two parts. The first part was to recognize landmarks and estimate the position for these. The second part was to, on the basis of the landmarks, localize and navigate on the field.

Part 1

For the first part the problem was how to, most precisely and efficiently, estimate the position of the landmarks. This was made, assuming that we knew the dog's position and direction on the field, by measuring the distance to a landmark and then calculating the position for it. Then we could mark them in our global map, whish is a representation of the field in the World State. Then the problem was for the dog to locate itself, i.e., measure its own position, and to estimate the distance to the landmark in the most accurate way. Clearly the distance became more accurate the closer we could get to the object. Also the dog's position was best estimated when it was positioned in front of one of the goals, directed at it. For the best landmark estimation we needed to take care of these things:

- Get as close as possible to the landmarks.
- Get to some positions where the dog can easily estimate its own position.
- Make it on time.

Since there was no way to know were the landmarks would be positioned around the field, we couldn't in advance decide where to go to get close to them. Also there was a risk for the dog not being able to locate itself, or at least very badly, if they were placed somewhere along the borders. The fact that the first point wasn't possible to foresee made us focus more on the second point and then try to get as close as possible to the landmarks by scanning as much as we could of the field surroundings. By moving between different points in the field, scanning in all directions, we tried to cover all the surrounding area. Unfortunately the dog was not able to walk, at the same time locating both itself and the landmarks. The best way was to let it visit a couple of predetermined points, let it turn around there and scan the area there. Now we had the problem to find useful points. Our first idea was to visit each corner of the field, as illustrated in figure 19, but this wasn't possible due to the time limit.

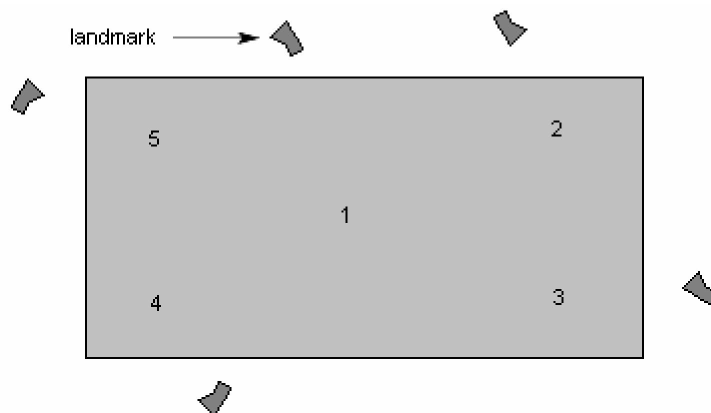


Figure 19. The field with new landmarks placed in the surroundings.

Then we decided just to use three scanning points, wish was enough to cover the entire field, as seen in figure 20.

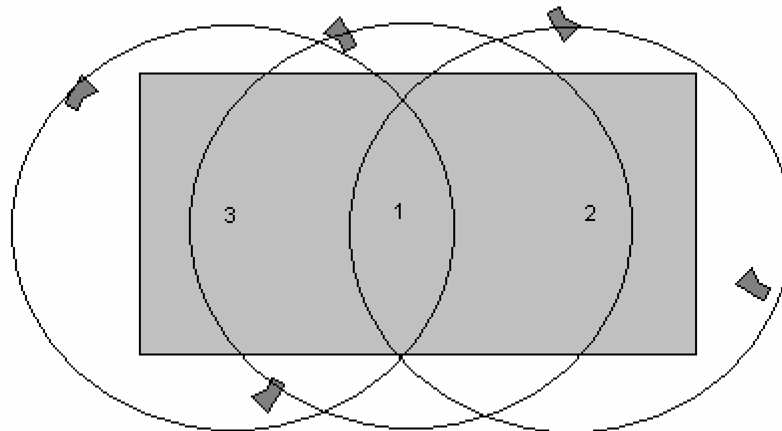


Figure 20. The field and the dog's view from tree points.

An advantage with using the two points in front of the goals was that we could use the goToObject behavior instead of the goToPosition behavior. It is faster to walk towards an object than to find a coordinate in the field since no localization is needed.

This is the solution for the first part of the almost SLAM challenge, and the resulting machine is displayed in figure 21.

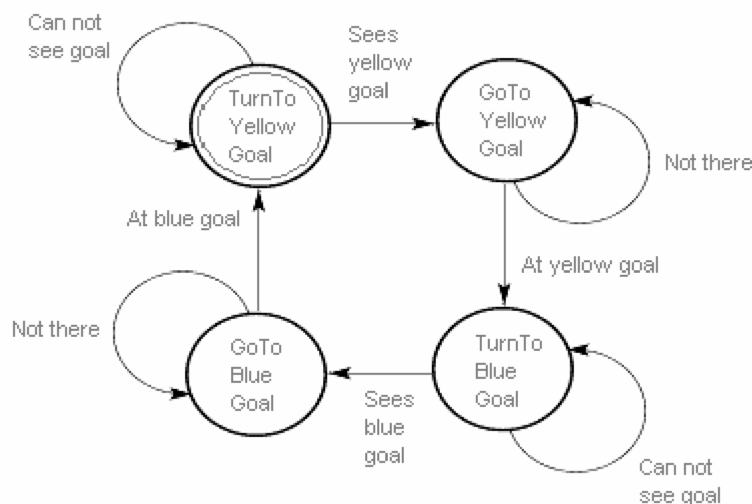


Figure 21. State machine for SLAM challenge part 1.

1. The dog is placed in the center of the field. On the starting signal the dog turns >360 degrees until it sees the yellow goal.
2. The dog walks with maximum speed towards the yellow net until it reaches a point placed one meter in front of the goal.
3. The dog turns 540 degrees and stops when it sees the blue goal.

4. The dog walks with maximum speed towards the blue net until it reaches a point placed one meter in front of the goal.
5. The dog turns 540 degrees and stops when it sees the yellow goal.
6. At this point the dog should have walked to a place, efficient to start part two at, but we run out of time. Instead it just start walking towards the yellow goal again.
7. One minute after the starting signal the dog stops. That's just when it reaches the yellow goal.

Part 2

For the second part of the challenge five points in the field were to be marked by the dog. To mark a spot the dog was to wag its tail, standing on the coordinates for the spot. Five points were to be visited. The only behaviors needed for this part was goToCoordinate and Mark. The resulting machine is illustrated in figure 22. A question was in which order to visit the points. The solution to this became to calculate the shortest path through all the points, starting from the point the dog was standing at when the second part started. Since only six points were to be taken into account there was no problem calculating all possible paths and use the shortest from them. Afterwards it would have been wiser to calculate a point to start from when the second part started.

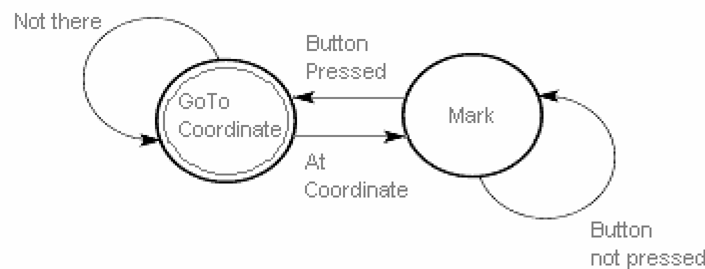


Figure 22 State machine for SLAM challenge part 2.

This is the solution for the **second** part of the almost SLAM challenge:

1. Go To the first point in the shortest path list.
2. Mark the spot by wagging the tail and wait for a push on the head to continue.
3. Go to next point in the shortest path list.
4. Repeat 2 to 3 until two minutes have pasted or five points have been visited.
5. Stop.

7 Results

Since this program competed in the robocup challenges in Lisbon this year, there were two results; how the implementation behaved in the robot lab and how it did in Lisbon. But actually there is one significant difference: the result from Lisbon tell how the whole project performed, not just the behavior part. This section will therefore only refer the result obtained in the laboratory.

The implemented machines have both the functions that are needed for the challenges. That is, for the Variable light challenge: repeatedly score as many goals as possible in three minutes and, for the SLAM Challenge, first look for alternative landmarks and then go to five positions in the field. By utilizing the finite state machine approach we were able to construct an implementation which was modular, stable and had the behaviors that were needed for the challenges in Robocup 2004.

This is the best result that we got from our machines:

- The robot managed to score 4 goals in three minutes.
- The robot managed to mark four spots within an average error margin of 50 cm in the time that it had according to the challenge rules.

7.1 Tests

We did two types of testing, practical and software simulations. The practical testing was conducted in the soccer field when the program was run on the robot, and the software simulations were conducted on the computers in the robotic laboratory. When we conducted the practical testing we used the divide and conquer approach. That means in this case that we first tested the small simple behaviors and then combined them into more and more complex behaviors which were also tested. The simplest behavior could for example be walking forward. A bit more complex behavior could for example be walking towards a ball and one of the most complicated behaviors could for example be walk to a ball and stop 10 cm in front of it while aligning with a net. The behaviors that interacted with other objects, for example go to ball, we first tested with the robot acting totally reactive and then as the robot became "smarter" we tested it by using the information that was stored in Worldstate. When we performed the most advanced tests, we prepared the field as similar as possible to the set up for the challenges.

Basically we were only able to test our implementation in the robotics laboratory under ideal conditions and without some features of which some were quite relevant for how the robot would behave in the real challenge conditions. Features which we had to work without were for example robot detection and alternative landmark detection. The robot was also very sensitive to the lighting of the room. This made it difficult to construct behaviors that would work under the conditions that were in Lisbon, but we tried to make the behaviors as good as they could be under these circumstances.

We tested some parts of our implementation by software simulations. These were very task specific simulations and no specific tool was developed. It was mostly specific algorithms that were tested, for example the shortest path algorithm used in the SLAM challenge. Software

simulations consisted of the algorithm that should be tested and a C++ program that was written specifically for testing that algorithm.

The implementation was continuously tested throughout the entire development, however there were some problems during the practical testing. Previous modules in the framework were not fully developed and therefore we could not be sure that the inputs to our module were correct. This made it hard to pinpoint the location of the error since we couldn't be sure whether our module had failed or if the error propagated from an earlier stage of the pipeline..

7.2 Analysis of the result

The result in the robotics laboratory and the result in Lisbon differed a lot. This could be due to a lot of reasons, some of them are more obvious than others. The most obvious one is perhaps that the environment in the robotics laboratory and the one in Lisbon differed too much, since the robot performed well in Lund and poorly in Lisbon. As mentioned earlier we do only consider the result in Lund to be relevant for this thesis.

The general states and state machine approach that we decided to use has, as any solution to a problem, both advantages and disadvantages.

The advantages are that state machines provide good control over the behavior that is activated. It is easy to construct new machines from the states that we have prepared.

The disadvantages are that our FSM provide a general solution to a specific problem. Speed could be improved if the states were more specific to the tasks and the states could be constructed more general if they were not constructed for any particular task.

8 Conclusions and Future Work

During this master thesis we have learned a lot. When we started it, we just had some vague ideas of what we wanted to do and how to do it. As the project evolved, so did our knowledge of the problem and our ideas of how to solve it. At the end we successfully implemented finite state machines to control the behavior of an AIBO ERS-7 in the Robocup challenges 2004.

When we summarized our work we came to some conclusions about what we should have done different if we were to do this again. We have roughly divided the conclusions into two types: our work and the whole project.

- Our work
    - We should have tested our software more with other test software instead of by practical tests in the field.
    - Our user interface became too messy, especially the pre- and postconditions. We should have considered another way of representing the state transitions.

- The project in its entirety
    - The framework should have been more complete before we started. There should have been working Vision and Localization modules that computed correct information, so that our implementation had something concrete to start from. We spent a lot of time locating erroneous code in other modules.
    - The deadlines should have been kept better.
    - The framework and modules should have been better specified from the beginning – we had to rewrite our code far too many times.

This may seem a bit negative, but unfortunately it is the problems and mistakes that are best remembered. It has been interesting and fun working on a robot, which acts in the "real" world where nothing is black and white but everything is gray. We encountered problems which we solved, and we made mistakes which we learned from.

Possible extensions

Computer programs in general can always be improved and this is no exception. The behavior module is dependent on the modules which precede it in the control loop. It is difficult to implement functions and features that are not supported by the other modules. For example, if the Vision module can not detect an object, then Behavior can not avoid it. Apart from this there are a couple of improvements which could be made in the Behavior module.

- More states - The states that we made are the ones that were needed for the challenges, but to make the solution more general, more states could be defined and programmed.
- Refine the states - make them better and more "intelligent".
- Improve the adversary-robot-avoidance - when trying to score in the Variable light challenge.
- Develop an algorithm that describes the robot and which can be used to prove that the control system works by using hybrid automata theories.

References

[1] Web site of AIBO:
http://www.aibo.com
(Verified 14th October 2004)

[2] Web site of RoboCup:
http://www.robocup.org
(Verified 14th October 2004)

[3] K. LeBlanc, S. Johansson, J. Malec, H. Martínez and A. Saffotti, Team Chaos 2004
Proc. Robocup 2004 Symposium, Lisbon, Portugal, 2004.

[4] Y. Yokote, The Apertos Reflective Operating System, The Concept and Its Implementation.
Sony Computer Science Laboratory Inc., 1992.

[5] Web site of OPEN-R
http://openr.aibo.com
(Verified 14th October 2004)

[6] Web site of Tekkotsu
http://www.tekkotsu.org
(Verified 14th October 2004)

[7] R. R. Murphy, Introduction to AI Robotics,
MIT Press, 2000.

[8] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot"
IEEE Journal of Robotics and Automation vol. 1, 1986.

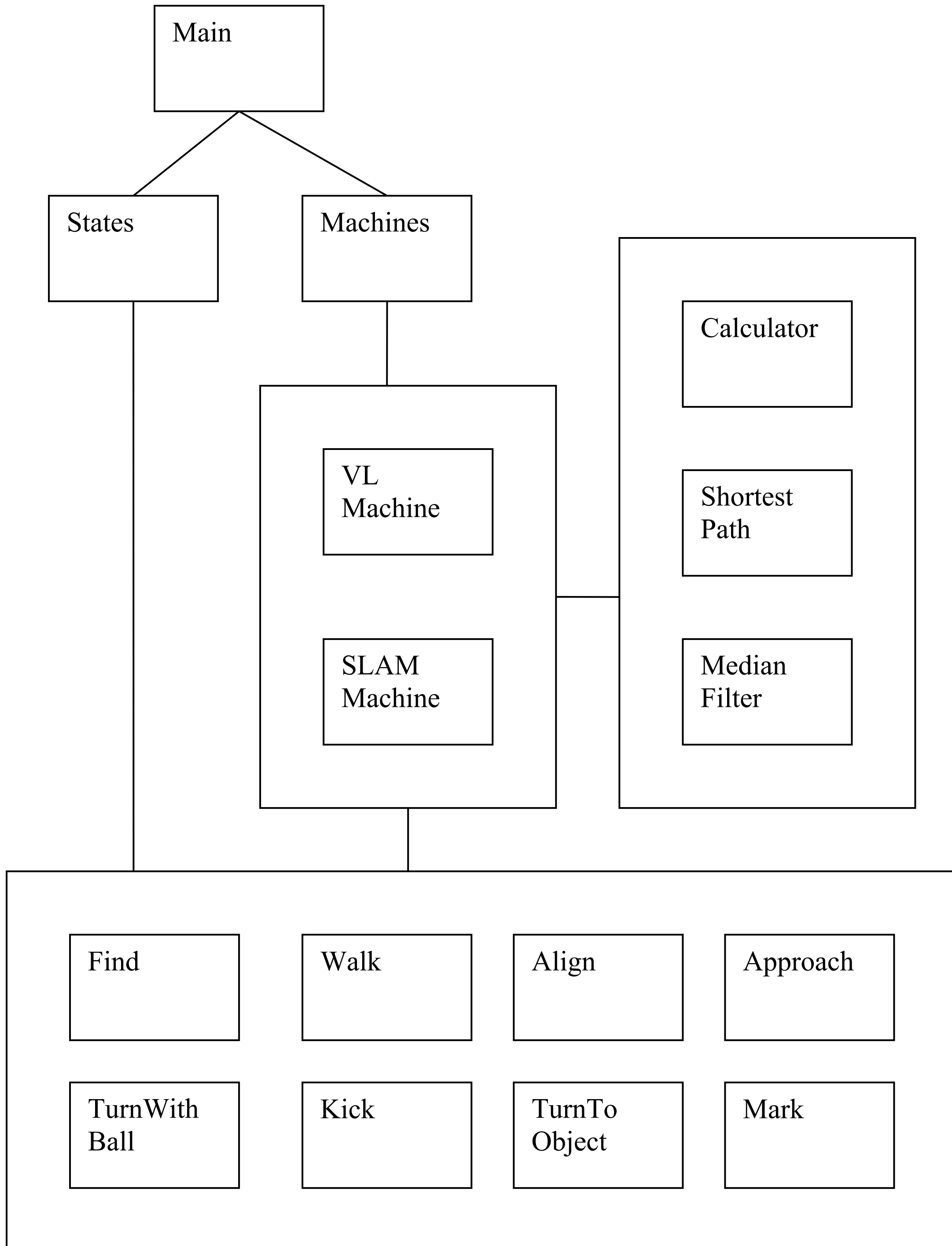
[9] S. Russel, P. Norvig, Artificial Intelligence A Modern Approach
Prentice Hall, 2003.

[10] Web site of wikipedia
http://www.wikipedia.org
(Verified 29th October 2004)

[11] R. Alur, C. Courcoubetis, T. A. Henzinger, P. Ho,
Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. Hybrid Systems, Lecture Notes in Computer Science 736, 1993.

Appendix

A. Class structure



B. States

Walk

1. GotoObject(object)

   - **Precondition:**
     MAX_CONFIDENCE - (CONFIDENCE_TOLERANCE + CONFIDENCE_OVERLAP) <= *object.Confidence* <= MAX_CONFIDENCE.
     MIN_PAN - CAM_HANGLE/2 <= *object.Theta* <= MAX_PAN + CAM_HANGLE/2.
     MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object.Epsilon* <= MAX_NOD + MAX_TILT + CAM_VANGLE/2.
     APPROACH_DISTANCE - (DISTANCE_TOLERANCE + DISTANCE_OVERLAP) <= *object.Distance* <= MAX_DISTANCE.

   - **Postcondition:**
     MAX_CONFIDENCE - CONFIDENCE_TOLERANCE <= *object.Confidence* <= MAX_CONFIDENCE.
     -THETA_TOLERANCE <= *object.Theta* <= THETA_TOLERANCE.
     MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object.Epsilon* <= MAX_NOD + MAX_TILT + CAM_VANGLE/2.
     APPROACH_DISTANCE - DISTANCE_TOLERANCE <= *object.Distance* <= APPROACH_DISTANCE + DISTANCE_TOLERANCE.

2. GotoCoordinate(x, y)

   - **Precondition:**
     MIN_CONFIDENCE <= *self.Confidence* <= MAX_CONFIDENCE.
     -Π <= *coordinate.Theta* <= Π.
     -Π <= *coordinate.Epsilon* <= Π.
     MIN_DISTANCE <= *coordinate.Distance* <= MAX_DISTANCE.

   - **Postcondition:**
     MIN_CONFIDENCE < *self.Confidence* <= MAX_CONFIDENCE.
     -Π <= *coordinate.Theta* <= Π.
     -Π <= *coordinate.Epsilon* <= Π.
     -GLOBAL_DISTANCE_ACCURACY <= *coordinate.Distance* <= GLOBAL_DISTANCE_ACCURACY.

Affects:

$$\text{Speed} = \sqrt{\left|\frac{theta}{MAX_THETA}\right|} * \sqrt[2]{\left|\frac{distnce - APPROACH_DISTANCE}{MAX_DISTANCE}\right|}$$

$$\text{Alpha} = \begin{cases} 0 & \text{distance} > \text{APPROACH_DISTANCE} \\ 1 & \text{distance} < \text{APPROACH_DISTANCE} \end{cases}$$

Spin = $\frac{theta}{MAX_THETA}$
Description: The dog will reactively walk towards the target. It will then stop at a specified distance from the object or position. The behavior has two options to choose from; fast or accurate.

Find

1. FindObject(object)

- **Precondition:**
  CONFIDENCE_INIT_VALUE <= *object.Confidence* <= MAX_CONFIDENCE.
  MIN_PAN - CAM_HANGLE/2 <= *object.Theta* <= THETA_INIT_VALUE.
  MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object.Epsilon* <=
  EPSILON_INIT_VALUE.
  DISTANCE_INIT_VALUE <= *object.Distance* <= MAX_DISTANCE.

- **Postcondition:**
  MAX_CONFIDENCE - CONFIDENCE_TOLERANCE <= *object.Confidence* <=
  MAX_CONFIDENCE.
  MIN_PAN - CAM_HANGLE/2 <= *object.Theta* <= MAX_PAN + CAM_HANGLE/2.
  MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object.Epsilon* <= MAX_NOD +
  MAX_TILT + CAM_VANGLE/2.
  APPROACH_DISTANCE - DISTANCE_TOLERANCE <= *object.Distance* <=
  MAX_DISTANCE.

Affects:

Speed = constant when backing, else 0.
Spin = constant when turning, else 0.

Description:

The dog backs for two seconds, then turns approx. 240 degrees, then backs again and so on. It starts by backing because the most common way of loosing the ball is under its view sight.

Align

1. AlignObjectWithObject(object1, object2)

- **Precondition:**
MAX_CONFIDENCE - (CONFIDENCE_TOLERANCE + CONFIDENCE_OVERLAP) <= *object1.Confidence* <= MAX_CONFIDENCE.
MIN_PAN - CAM_HANGLE/2 <= *object1.Theta* <= MAX_PAN + CAM_HANGLE/2.
MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object1.Epsilon* <= MAX_NOD + MAX_TILT + CAM_VANGLE/2.
APPROACH_DISTANCE - (DISTANCE_TOLERANCE + DISTANCE_OVERLAP) <= *object1.Distance* <= MAX_DISTANCE.
MIN_CONFIDENCE <= *object2.Confidence* <= MAX_CONFIDENCE.
$-\Pi$ <= *object2.Theta* <= Π.
$-\Pi$ <= *object2.Epsilon* <= Π.
MIN_DISTANCE <= *object2.Distance* <= MAX_DISTANCE.

- **Postcondition:**
MAX_CONFIDENCE - CONFIDENCE_TOLERANCE <= *object1.Confidence* <= MAX_CONFIDENCE.
-THETA_TOLERANCE <= *object1.Theta* <= THETA_TOLERANCE.
MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object1.Epsilon* <= MAX_NOD + MAX_TILT + CAM_VANGLE/2.
APPROACH_DISTANCE - DISTANCE_TOLERANCE <= *object1.Distance* <= APPROACH_DISTANCE + DISTANCE_TOLERANCE.
MAX_CONFIDENCE - CONFIDENCE_TOLERANCE <= *object2.Confidence* <= MAX_CONFIDENCE.
-THETA_TOLERANCE <= *object2.Theta* <= THETA_TOLERANCE.
MIN_NOD + MIN_TILT - CAM_VANGLE/2 < *object2.Epsilon* <= MAX_NOD + MAX_TILT + CAM_VANGLE/2.
MIN_DISTANCE <= *object2.Distance* <= MAX_DISTANCE.

2. AlignObjectWithCoordinate(object, x, y)

- **Precondition:**
MAX_CONFIDENCE - (CONFIDENCE_TOLERANCE + CONFIDENCE_OVERLAP) <= *object.Confidence* <= MAX_CONFIDENCE.
MIN_PAN - CAM_HANGLE/2 <= *object.Theta* <= MAX_PAN + CAM_HANGLE/2.
MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object.Epsilon* <= MAX_NOD + MAX_TILT + CAM_VANGLE/2.
APPROACH_DISTANCE - (DISTANCE_TOLERANCE + DISTANCE_OVERLAP) <= *object.Distance* <= MAX_DISTANCE.
MAX_CONFIDENCE - SELF_CONFIDENCE_TOLERANCE < *self.Confidence* <= MAX_CONFIDENCE.
$-\Pi$ <= *object2.Theta* <= Π.
$-\Pi$ <= *object2.Epsilon* <= Π.
MIN_DISTANCE <= *object2.Distance* <= MAX_DISTANCE.

- **Postcondition:**
`MAX_CONFIDENCE - CONFIDENCE_TOLERANCE` <= *object1.Confidence* <= `MAX_CONFIDENCE`.
`-THETA_TOLERANCE` <= *object1.Theta* <= `THETA_TOLERANCE`.
`MIN_NOD + MIN_TILT - CAM_VANGLE/2` <= *object1.Epsilon* <= `MAX_NOD + MAX_TILT + CAM_VANGLE/2`.
`APPROACH_DISTANCE - DISTANCE_TOLERANCE` <= *object1.Distance* <= `APPROACH_DISTANCE + DISTANCE_TOLERANCE`.

`MAX_CONFIDENCE - SELF_CONFIDENCE_TOLERANCE` < *self.Confidence* <= `MAX_CONFIDENCE`.
`-THETA_TOLERANCE` <= *coordinate.Theta* <= `THETA_TOLERANCE`.
-Π <= *coordinate.Epsilon* <= Π.
`MIN_DISTANCE` <= *coordinate.Distance* <= `MAX_DISTANCE`.

Affects:

Speed = $\sqrt[2]{\frac{|theta2 / MAX_THETA| + |DistanceToCircle / MaxDistanceToCircle|}{2}}$

$$DistanceToCircle = distance1 - APPROACH_DISTANCE$$
$$MAXDistanceToCircle = MAX_DISTANCE - APPROACH_DISTANCE$$

Alpha $= (\Pi / 2) * (APPROACH_DISTANCE / distance1)$ theta2<0
$-(\Pi / 2) * (APPROACH_DISTANCE / distance1)$ theta2>0

Spin = $\frac{theta1}{MAX_THETA}$

Description:

The goal for this state is to get an object in line between the dog and another object. Moreover the dog is to stay close to the object. It reactively makes its way towards the target, starting circulating around it as it gets closer. The dog will take into account which is the best direction to circulate. It will slow down at a predefined distance to object1, facing object2 or the point with coordinates x, y.

TurnWithBall

1. AlignBallWithObject(object)

- **Precondition:**
  MIN_CONFIDENCE <= *ball.Confidence* <= MAX_CONFIDENCE.
  MIN_PAN - CAM_HANGLE/2 <= *ball.Theta* <= MAX_PAN + CAM_HANGLE/2.
  MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *ball.Epsilon* <= 0.
  MIN_DISTANCE <= *ball.Distance* <= MIN_DISTANCE + DISTANCE_TOLERANCE.

- **Postcondition:**
  MIN_CONFIDENCE <= *ball.Confidence* <= MAX_CONFIDENCE.
  $-\Pi$ <= *ball.Theta* <= Π.
  $-\Pi$ <= *ball.Epsilon* <= Π.
  MIN_DISTANCE <= *ball.Distance* <= MAX_DISTANCE.
  .

Affects:
NECK-TILT, NECK-PAN, NECK-NOD, spin.

Description:
The dog grabs the ball under its chin and rapidly turns a predefined angle, then releases the ball again. It will take into account which is the best direction to turn. This might be a faster way of aligning than Align.

ApproachBall

1. Approach()

- **Precondition:**
  `MIN_CONFIDENCE` <= *ball.Confidence* <= `MAX_CONFIDENCE`.
  `MIN_PAN - CAM_HANGLE/2` <= *ball.Theta* <= `MAX_PAN + CAM_HANGLE/2`.
  `MIN_NOD + MIN_TILT - CAM_VANGLE/2` <= *ball.Epsilon* <=
  `MAX_NOD + MAX_TILT + CAM_VANGLE/2`.
  `APPROACH_DISTANCE - (DISTANCE_TOLERANCE + DISTANCE_OVERLAP)` <=
  ball.Distance <= `APPROACH_DISTANCE + (DISTANCE_TOLERANCE + DISTANCE_OVERLAP)`.

- **Postcondition:**
  `MIN_CONFIDENCE` <= *ball.Confidence* <= `MAX_CONFIDENCE`.
  `MIN_PAN - CAM_HANGLE/2` <= *ball.Theta* <= `MAX_PAN + CAM_HANGLE/2`.
  `MIN_NOD + MIN_TILT - CAM_VANGLE/2` <= *ball.Epsilon* <= 0.
  `MIN_DISTANCE` <= *ball.Distance* <= `MIN_DISTANCE + DISTANCE_TOLERANCE`.

Affects:
Speed = constant.
Alpha = *approachFactor* * *theta* .
TILT, PAN, NOD.

Description:
The goal for this state is to position the dog with its chin over the ball, which is its kicking position. The dog is approaching the ball slowly until it believes the ball is under its chin.

TurnToObject

1. Turn(object)

- **Precondition:**
  CONFIDENCE_INIT_VALUE <= *object.Confidence* <= MAX_CONFIDENCE.
  MIN_PAN - CAM_HANGLE/2 <= *object.Theta* <= THETA_INIT_VALUE.
  MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object.Epsilon* <=
  EPSILON_INIT_VALUE.
  DISTANCE_INIT_VALUE <= *object.Distance* <= MAX_DISTANCE

- **Postcondition:**
  MAX_CONFIDENCE - CONFIDENCE_TOLERANCE <= *object.Confidence* <=
  MAX_CONFIDENCE.
  -THETA_TURN_TOLERANCE <= *object.Theta* <= THETA_TURN_TOLERANCE.
  MIN_NOD + MIN_TILT - CAM_VANGLE/2 <= *object.Epsilon* <=
  MAX_NOD + MAX_TILT + CAM_VANGLE/2.
  MIN_DISTANCE <= *object.Distance* <= MAX_DISTANCE.

Affects:
Spin = constant.
TILT, PAN, NOD.

Description:
The dog turns around until it sees the object.

Kick

<table>
<tr><td>

1. Kick(kicktype)

- **Precondition:**
  `MIN_CONFIDENCE` <= *ball.Confidence* <= `MAX_CONFIDENCE`.
  `MIN_PAN - CAM_HANGLE/2` <= *ball.Theta* <= `MAX_PAN + CAM_HANGLE/2`.
  `MIN_NOD + MIN_TILT - CAM_VANGLE/2` <= *ball.Epsilon* <= 0.
  `MIN_DISTANCE` <= *ball.Distance* <= `MIN_DISTANCE + DISTANCE_TOLERANCE + DISTANCE_OVERLAP`.

- **Postcondition:**
  `KickType = 0`.

</td></tr>
<tr><td>

Affects:
Takes control over the whole dog.

</td></tr>
<tr><td>

Description:
Depending on the input, a certain kick is performed.

</td></tr>
</table>

MarkPoint

<table>
<tr><td>

1. Mark()

- **Precondition:**
  true
- **Postcondition:**
  Time is up. Wagtime = 3000 ms.

</td></tr>
<tr><td>

Affects:
Tail waging.
Speed, alpha, spin = 0.

</td></tr>
<tr><td>

Description:
The dog is standing still, wagging its tail until the predefined Wagtime has passed.

</td></tr>
</table>

C. User’s Manual

In this manual we explain how to use our interface for building state machines in order to create behaviors for an AIBO ERS-7. The implementation is part of the Team Chaos Challenges framework (see chapter 4 in the report) which is based on the Tekkotsu[6] framework and is dependent on these.
The manual contains three parts:

- Overview
- How to crate state machines
- Example

Overview

The Team Chaos Challenge (TCC) framework consists of five modules, where this manual concerns only the behavior module. The behavior module runs in series with Vision, and Wireless, thus it is only activated after information has been updated by these modules. Localization runs in parallel and updates information less frequently.

All modules, including Tekkotsu, get and set all information in the shared memory area Worldstate. Tekkotsu is the interface to the AIBO and manages the hardware.

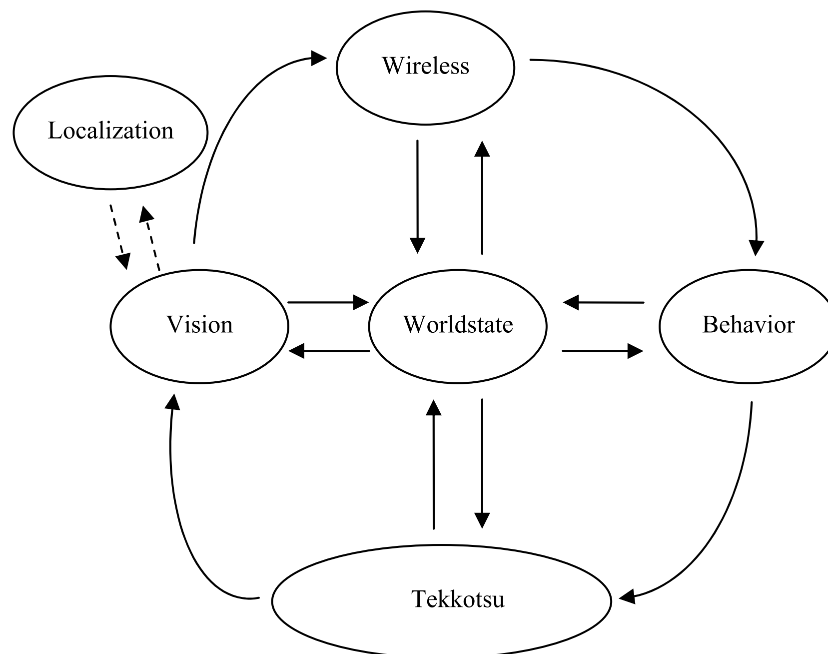


Figure 1. The TCC framework.

Behavior can be seen as a black box with a number of inputs and outputs.

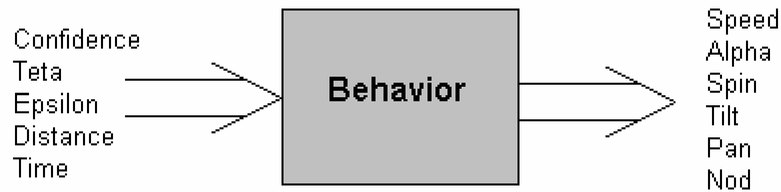


Figure 2. Behavior black box.

The input variables are set by Vision, Wireless, Localization and Tekkotsu and are used by Behavior to calculate the output. The black box contains state machines.

How to create state machines

In the Behavior module there are two central kinds of objects, namely States and Machines. A State object corresponds to a circle in the state diagram while a Machine object contains and connects State objects, that is, corresponds to the whole diagram. To output a behavior, states are combined in certain ways in a machine. At the moment there are ten states that can be combined. More information about the existing states can be found in appendix B. The present states are:

- GoToObject
- GoToCoordinate
- FindObject
- AlignObjectWithObject
- AlignObjectWithCoordinate
- Approach
- Kick
- Mark
- TurnToObject
- TurnWithBall

When building the machine, states are used as building blocks. To make the states fit with each other they have to be combined in certain ways. Some of the states can be anywhere in the machine, but most of them have to be in a specific order. For example kick has to be used after approach, since it is impossible to kick the ball if the dog is not at the ball.

	Sees object	Near object	At object
FindObject TurnToObject GoToCoordinate	AlignObjectWithObject AlignObjectWithCoordinate GoToObject	Approach	Kick TurnWithBall

Table 1

The states in the first column of Table 1 can always be used. The states in the second column are dependent on that one of the states in the first column is accomplished and so on. For example Kick or TurnWithBall must be preceded by Approach and can not be preceded by GoToObject, since the dog has to be at the ball in order to kick or turn with it. Approach, in turn, must be preceded by one of the states in column two.

Wait Mark	FindObject TurnToObject GoToCoordinate

Table 2

The states in the first column of Table 2 differs from the ones in Table 1 by the fact that they can be used at any time, but the states in the first column should be followed by the ones in the second column. This is because the Wait and Mark states use time as the condition for state change, instead of object preferences as the other states do.

Each state has a precondition and a postcondition. The precondition is fulfilled when it is considered meaningful to run the state and the postcondition is fulfilled when the "goal" of the state is reached. The pre and postcondition are used to connect the states to each other as illustrated in figure 3. For more information about the pre- and postconditions see chapter 6.1 in the thesis.

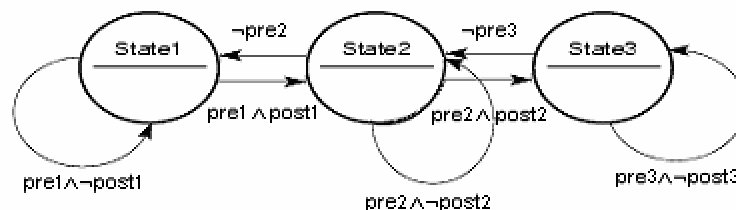
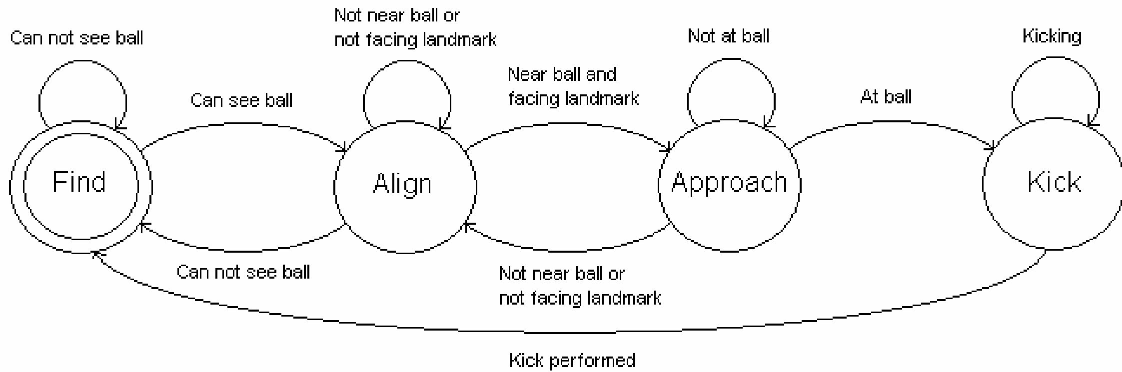


Figure 3. Pre- and postconditions.

Example

This example shows how to create a state machine for the Behavior module in the TCC framework. The implementation is written in C++ code, as everything in the OPEN-R environment.

Let's say that you want the dog to shoot the ball towards one of the landmarks. A state diagram could then look like this:



First the dog has to find the ball, which is the start state (marked with double circles). When the dog knows where the ball is, it starts aligning itself with the landmark. This means that it tries to get in a position where the ball is in line between itself and the landmark. It also walks towards the ball until it is standing in front of it. However, to perform the kick the ball has to be under the dogs chin, where it can not see it. Therefore, after aligning, the dog has to approach the ball, which means that it walks forward until the ball is under its chin. When the dog believes that it is in right position it kicks the ball.

If we call this machine BallKicker, the code would look like this:

CODE

```
void BallKicker::Machine()
{
  switch(State){

  case FindingBall:
    {
      if(FindObject.preConditionLocateObject(WS->GetBall())){
        FindObject.locateObject();
        if(FindObject.postConditionLocateObject())
          State = Aligning;
      }
    }
    break;
  case Aligning:
    {
      if(Align.preConditionAlignObjectWithObject(&WS->GetBall(),
&WS->GetLandmarkAR())){
        Align.alignObjectWithObject();
        if(Align.postConditionAlignObjectWithObject())
          State = Approaching;
      }
      else
        State = FindingBall;
    }
    break;
  case Approaching:
    {
      if(Approach.preConditionApproach()){
        Approach.approachBall();
        if(Approach.postConditionApproach())
```

```
          State = vl_Kicking;
      }
      else
        State = Aligning;
    }

    break;
  case Kicking:
    {
      if(Kick.preConditionKick()){
        Kick.doKick(netA);
        if(Kick.postConditionKick())
          State = FindingBall;
      }
    else
      State = FindingBall;
    break;
  default:
    cout << "This should never happen" << endl;
  }
}
```