MAAT - Multi Agent Authoring Tool for Programming Autonomous Mobile Robots

Diplomarbeit

bei Prof. Dr. Raúl Rojas

vorgelegt von
Anna Egorova
egorova@inf.fu-berlin.de
am Fachbereich Mathematik und Informatik
Freie Universitẗ Berlin
Takustr. 9, 14195 Berlin
Berlin, den 24. Juni 2004

18th July 2004

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig angefertigt habe, sämtliche Quellen, die verwendet wurden, angegeben habe und dass die Arbeit nicht schon an anderer Stelle zur Prüfung vorgelegt wurde.

Berlin, den 19. Juli 2004

Abstract

This work presents a dedicated programming framework for behavior-based autonomous mobile robots. It has been developed to manage and assist the developing process of the behavior control of the FU Fighters system. However, the same approach can be used for developing any other programming framework for autonomous mobile robots.

The behavior control of the FU Fighters small size team is a behavior-driven architecture. It is based on the Dual Dynamics scheme and extends many of its features in order to present an Extended Dual Dynamics architecture. The building blocks of the architecture are behaviors, organized into a hierarchy of time-dependant layers. They feature their own dynamics, divided into an activation, perceptual and target dynamics.

This hierarchy is being mirrored by the framework, in order to create a user-friendly graphical programming interface. It provides tools for displaying the hierarchy, managing its elements and the source code, and managing the various properties of the elements.

Contents

Chapter 1

Introduction and Related Works

The last several years brought great development to the field of robotics and artificial intelligence. This can be best seen in the development of the RoboCup championships and the RoboCup teams. The hardware and the communication are getting more complex and precise, the computer vision is getting more stable, exact, reliable and faster. The best achievements can be though observed in the field of the behavior control and software as a whole.

Several years ago, a complex game with 5 or more robots, which communicate dynamically among each other and perform real team work, passing the ball to each other through the half field of play, was nearly unthinkable. Nowadays, an exciting, complex and very fast play in the small size RoboCup league is normal. This complexity and speed have on the other side its price. The software is getting more and more complex and the whole system is very difficult to manage and to develop. The FU Fighters behavior module consists, for example, of about 86.000 lines of code and a total of about 2 millions characters (see also Table 1.1 for more statistics of the FU Fighters source code). This makes the system very vulnerable to programming and logical errors. The complete testing of the system gets merely impossible and the integration of improvements and developments a hopeless job.

This work concentrates on the programming of small-sized robots, although the same methods could also be applied to all other RoboCup leagues, as well as to other mobile autonomous robots. The goal of the presented

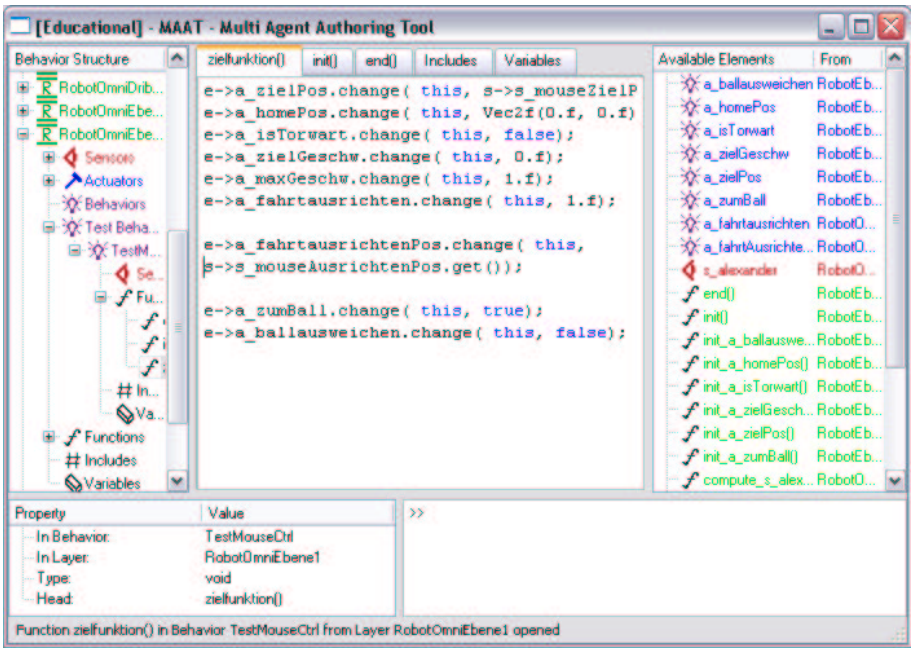


Figure 1.1: Overview of the MAAT graphical user interface.
The overview of all layers is on the left of the screen; all available elements for programming (sensors, functions, actuators) are on the right. The coding area is in the center; it mirrors the architecture of one particular, at this moment opened for programming, layer or behavior in a layer. An element viewer in the left bottom of the screen gives all information about a selected element (layer, behavior, sensor, actuator etc.). A small output and debug area is positioned on the right bottom.

work is to show that a good managed software system, developed especially for the needs of mobile robotics and more over, especially for one specific behavior architecture, is a big help to the developers.

MAAT is a programming framework for autonomous mobile robots, developed to manage the source code of the behavior architecture of the FU Fighters small size robot team. It provides a graphical interface for managing the source code and the behavior architecture itself. It provides also graphical tools for displaying and managing different architectural features. More information about the different tools and the graphical user interface can be found in Chapter 5.

Figure 1.2: Maat, the ancient eqypt goddess.
Maat was the goddess of truth, law and universal order. Maat or also named Ma'at is the kneeling one. The goddess on the throne is Isis.
Courtesy *www.kathsrealm.com*

The chief goal of the programming framework is to mirror the unvisible connections between all parts of the behavior architecture and to show them well ordered to the software developer. Thus, one doesn't have to know the building blocks of the system or how they work which each other, in order to start programming. All of the information one needs can be easily taken from the framework or looked up later. All of the connections between the many parts of the architecture can be managed under the assistance of graphical tools, thus making it unneccessary for the developer to keep in mind all of the thousands of details. Figure 1.1 shows a screen shot of the MAAT framework.

MAAT, or Multi Agent Authoring Tool, is also the name of an ancient egypt goddess. Maat was the goddess of wisdom and order, who kept the world of the ancient Egyptians in balance. (Fig. 1.2):

	Behavior Module	FU Fighters System
Number of files	516	1454
Number of lines	86.669	439.240
Number of significant lines	73.371	350.206
Percent of significant lines	84,7	79,7
Number of characters	2.573.202	14.748.302
Number of siginificant characters	2.268.743	11.924.909
Percent of significant characters	88,2	80,8
Size	2.750.136 Bytes	12.858.227 Bytes

Table 1.1: Statistics of the source code of the FU Fighters behavior module. The high complexity and size of the software can be clearly seen. The statistics were done with Source Code Statistics by LeptoSoft *http://users.otenet.gr/ nkourkou/*.

The presented work is divided into the following parts. First, an introduction of the Robocup environment and its different leagues is given. There you can also read something of the idea of this testbed for artificial intelligence and about its history.

An overview of different related or analogue works is presented. The most important ones are probably the works of Jäger [10] about Dual Dynamics, the works of Bredenfeld [3, 4, 5] about the DD Designer - a similar development environment of the german middle size RoboCup team AIS-GMD; and the works of Rojas and Behnke [2] about the behavior architecture of the FU Fighters.

Then, the behavior architecture of the FU Fighters is given in detail in Chapter 4, followed by the description of the MAAT, its graphical user interface, graphical tools and inner system layout in Chapter 5. At the end, an overview of the results of this work is given in Chapter 6.

1.1 Related Works

Efforts are made for long time to build an adequate programming interface for mobile robots. A very good "state of art" of general programming of robots give Geoffrey Briggs and Bruce MacDonald [6]. They make an overview of programming systems for all kinds of robots, however especially

for service oriented robots. They differenciate between automatic systems, where programming is based on interaction with the robot and learning and manual ones, which programm the behavior of the robot in a traditional way of coding it.

Much efforts are made to visualize and simulate robot's behavior before applying it to the real robot. For example, another work of the University of Auckland presents a graphical tool for visualization and simulation of robots after programming it's behavior [16]. Another similar effort is explained in the work of Gloye et all. from the Free University of Berlin, Germany [8], where a learned movement model of the real robot is used for simulation.

The DD-Designer is a robotic software development framework developed by GMD middle sized RoboCup team [3, 11]. It supports the Dual Dynamics robot control architecture [10]. It uses an abstract hyper-graph of typed data processing elements to generate HTML documentation, Java simulation code, C++ robot control programs and the parameters for their real-time monitoring tool beTee. An extensive description of the structure and modell of the DD Designer is given by [4]. It introduces the concept of integration classes, interface specification, derived interfaces and interface wrappers for effective software prototyping.

Another approach is building dedicated programming languages for robot behavior. The language *ConGolog* is a further development of Golog, described from the York University in Toronto [12]. It is a logic based robot/agent programming language, where high reactivity is achieved by ConGolog's concurrent processes and interrupt facilities.

Chapter 2

The Robocup Environment

Figure 2.1: The Robocup Logo.
Courtesy *www.robocup.org*

The research areas in artificial intelligence have always involved questions and tasks, which are fully natural for humans and which encounter great problems by implementing them on computers. Such tasks include face recognition, natural speech, learning, bipedal walking etc. There are on the other side problems, like chess, which are very difficult for humans to learn and master them, but are solvable for computers.

The progress in the AI research needs a benchmark, a testbed for evaluating and testing the newest efforts in these areas.

Robocup[1] is a world championship for soccer playing autonomous robots. RoboCup was established in order to give the needed testbed and to motivate research organisation to increase their efforts in robotic research. The world

[1]The RoboCup Organisation www.robocup.org

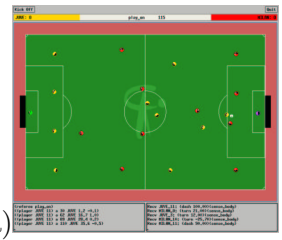
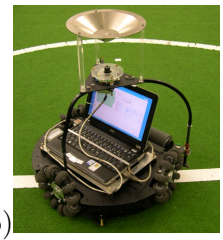
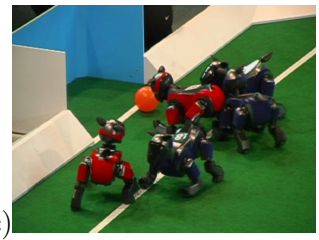
(a)  (b)  (c) 

Figure 2.2: Simulation, Middle and 4-Legged Robocup leagues.
(a) The PaSo team from the University of Padova in the simulation league. (b) The FU Fighters middle size team of the Free Uuniversity of Berlin. Please refer to [9] for more information. (c) The Dribbling Dackels team from the Technical University of Darmstadt in the Sony 4 Legged League.

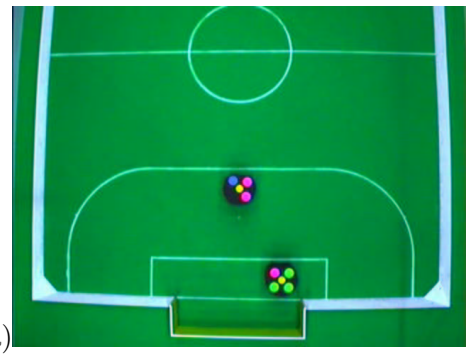
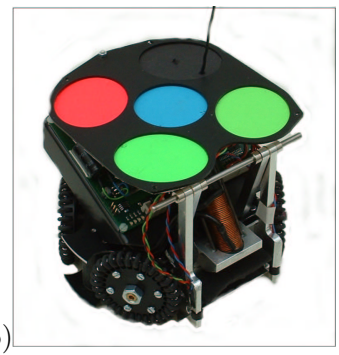
(a)  (b) 

Figure 2.3: Small size Robocup league.
(a)The field of play for the small-size league, as seen from one of two global cameras, positioned over the field. (b) The small-size player of the FU Fighters, as in 2004.

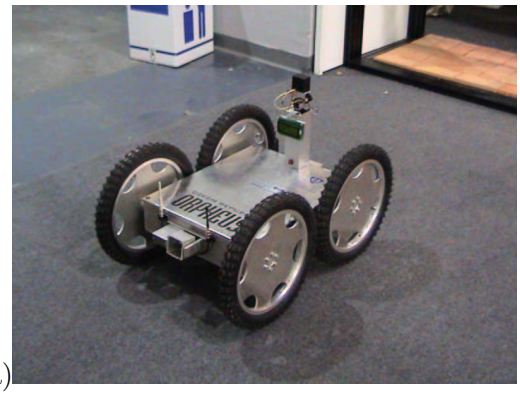
(a)  (b) 

Figure 2.4: Rescue and Humanoid RoboCup leagues.
(a) The Robrno team from the Brno University of Technology in the Rescue Robot League. (b) The humanoid Robo Erectus of the Singapore PolyTechnic University robot.

championships are held every year in different countries with participating teams from robotic research institutions from all over the world. Besides the world events there are also regional championships - for Europe and Germany it is the yearly held German Open championship in Paderborn, Germany.

The RoboCup teams have the opportunity to test their robots, their mechanics and behavior and to exchange expreciences with other teams.

RoboCup was first held in Japan in 1996 as a "preliminary RoboCup championship". This year (2004) Robocup will be held in Lisboa, Portugal for the 8th time. For more information about the project and its history see [14].

2.1 The Robocup Leagues

In 2004, there are 8 different RoboCup leagues, separated into two divisions - RoboCup Soccer and RoboCup Rescue. Beside these two categories, there is also a RoboCup Junior League (see Table 2.1).

The oldest league in RoboCup is the Simulation League, started 1996 as the Pre RoboCup. 11 virtual players are playing against a similar team

Year	Teams	Sim	Small Size	Mid-Size	Legged	Rescue	Humanoid
1997	41	32	4	5			
1998	62	34	12	16			
1999	73	35	18	20			
2000	84	40	16	16	12		
2001	105	44	20	18	16	7	
2002	133	46	20	16	19	20	12
2003	131	35	18	23	19	22	14
2004	159	32	20	24	24	40	14

Table 2.1: Attendance to the RoboCup World Championship, number of teams in each league, 1997 - 2004. Data from the RoboCup Organisation.

(see Fig. 2.2a). The small size and the middle size leagues joined the championship in 1997. The rules for the leagues change every year to carry the further development of thne robots.

The small size robots have a maximum diameter of 18 cm and a maximum height of 15 cm (see Fig. 2.3b). The current field of play is 5,5 m to 6,5 m. The most important sensor of the team are global cameras, positioned above the field, which catch an image of the whole field with the robots and the ball on it, and pass it to a central processing unit, usually a PC. Some teams, like the FU Fighters, use more than one camera in order to catch several views of the field and to use higher resolution of the images. An image of the small size field, as seen from above, is given in Fig. 2.3a. One specific characteristics of small size league is the high speed of the robots. They move at above 2 m/sec, crossing the whole field in about 3 sec. This fact demands a highly adaptive and reactional behavior of the robots.

The middle size players have a maximum diameter of 50 cm and play in teams of up to 6 players on a field of 10 m length to 5 m width. Each of the robots in this league has his own PC or laptop and a local camera (see Fig. 2.2b).

Starting in 2000, the 4-Legged League was formed. The league uses one standard hardware platform - the Sony Aibo Robot. In 2001 and 2002, the Rescue and the Humanoid Leagues were formed respectively (see Fig. 2.4). With the start of the research in the area of humanoids, the RoboCup Or-

ganisation has made its first step towards the end goal of the initiative:

- By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team. - [2].

[2] www.robocup.org

Chapter 3

Overall Structure of the FU Fighters System

One specific characteristics of the RoboCup small size league is the high speed of the robots. They move at above 2 m/sec, crossing the whole field in about 2.5 sec. This fact demands a highly adaptive and reactional behavior of the robots, with a minimized system delay. The FU Fighters small size team uses for example prediction of the plant's position and direction in order to nearly elliminate the effect of the system delay. The prediction control is explained in detail in [1].

Figure 3.1 gives a brief description of how the whole system of the FU Fighters small size robots is functioning. As explained earlier in Chapter 2, the small size architecture uses as single sensor one or more global cameras, positioned over the field of play. The dimensions of the robots, the field, the markers on the field and the height of the camera are defined in the small size league rules (visit *www.robocup.org* for more information and the current rules).

The FU Fighters small size team uses two global cameras at 60 frames/sec each to sensor the field of play. The images are sent to a central processing unit, usually a normal off-field PC, which undertakes the calculations. The first step is the processing and evaluating the images from the cameras. In both views (from each camera one) the robots and the ball are found, using the color information. The ball is, for example, always orange, and the robots have one mandatory team marker – blue of yellow in the middle of the robot – and other color markers on the cover, which help to identify each exact position and direction. Figure 3.2 shows the view from one camera

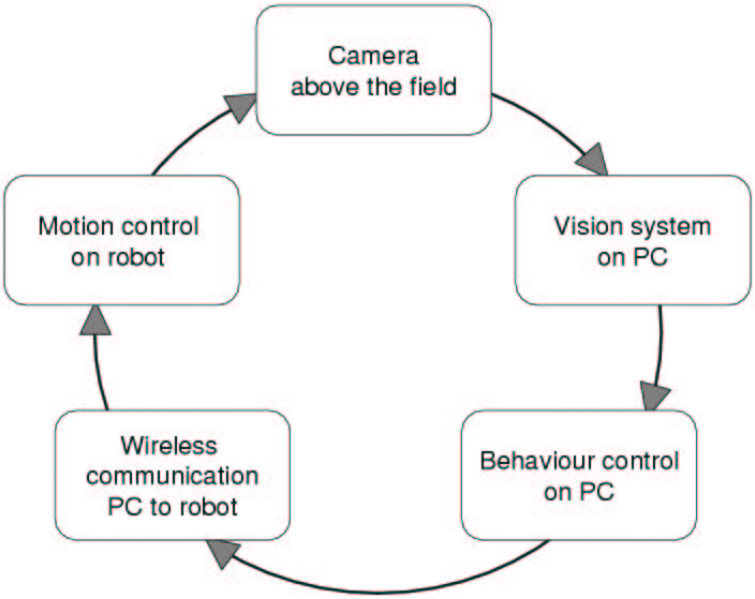


Figure 3.1: The overall control circle of the FU Fighter's small size system. The significant modules are the global sensor, the vision system, the behavior control, the wireless communication and the on-board controller on each robot.

and Figure 3.3 the color markers of the FU Fighters as in 2004.

The evaluated information from both views is being merged and the final positions and orientations of the robots and the ball are computed. This "world" is then being passed to the next module, the behavior control. More information about the vision of the FU Fighters can be found in [15].

The behavior control uses the positions and orientations of the robots together with the position of the ball to calculate the best behavior step for every robot. The behavior is planned in a top-down matter, deciding first what the team as a whole has to do and then precising these decisions down to the robots. The behavior control is explained in detail in Section 4 and in [2].

The next step is to pass the information to the robots. The commands for every direction (X and Y axis, rotation) is being calculated and passed to the robots via wireless communication. On the robot itself, only a minimal "intelligence" is provided by a PID controller. More information of the PID

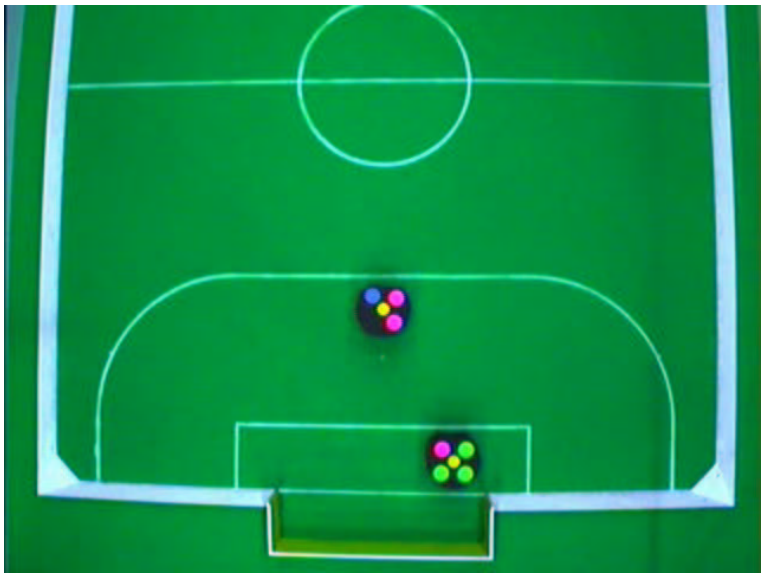


Figure 3.2: The field of play
as viewed from one of two global cameras with robots.

controller and the optimization of it is given in [8].

A more detailed description of the team as a whole is given in [7].

Figure 3.3: The color markers of the small size robot as in 2004.
In the middle is the mandatory team marker (blue or yellow), the other markers code the orientation of the robot and its identity number.

Chapter 4

Behavior Architecture

The behavior architecture of the FU Fighters small size team has been developed in its latest version for the RoboCup competition in 1999 by Rojas and Behnke [2]. It is based on the Dual Dynamics principle of Herbert Jäger [10], with extended multi-level time hierarchy with simple and fast behaviors on the bottom and slow and complex behaviors on the top of it. Deliberation is not explicitly designed, though the cooperation and interaction of the behaviors on all hierarchy levels immitate deliberation processes.

The ordered, multi-level, architecture constists to a big part of it of simple, similar elements with similar inner design. This gives the idea to automate some processes in the developing of the system, like adding new sensors, new actuators or new behaviors for example. Such processes are simple, yet time-consuming and error-pruned. They are part of the allday work of each developer and have nothing to do with the artistic process of designing and developing better or faster robot behavior. The best solution of this is therefore designing a dedicated graphical framework. It should manage the architecture and its elements, like layers, behaviors, sensors and actuators; set restrictions to the development process in order to achieve neaty, well-organized and readable code; provide graphical and non-graphical tools for displaying and managing the work of the architecture.

In order to better understand the need of such a framework and to understand the role of it in the existing system, a detailed description of the behavior architecture itself is first given.

In this chapter, first an overview of the classical Dual Dynamics architecture is given. Then, the extended and modified architecture of the FU Fighters, the extended Dual Dynamics architecture, is described in detail.

The communication and cooperation of all elements is discussed and some examples of robot behaviors are given. An overview of the challenges and difficulties of the system's implementation and management is given in Section 4.5.

4.1 Dual Dynamics

The Dual Dynamics scheme is was first presented by Herbert Jäger in 1998, see [10]. They described a formal scheme for robotic behavior control systems. In their work, they give a good formal theory for designing "behavior-based" robot control.

The behaviors are designed as dynamical systems, which are specified in ordinary differential equations. The key idea is that the architecture has two levels, which enables the agent, the robot, to work in different "modes". These modes are specified by the higher level of control, and the different modes activate different combinations of behaviors in the lower layer of control, which leads to qualitative different behavioral patterns of the agent.

Each behavior is a dynamical system, which interacts with the other behaviors through shared variables. There is no global control; the functionality of the whole system arises from the interaction and the activation dynamics of the behaviors themselves.

4.1.1 Organisation of the DD robot architecture

The building blocks of the DD scheme, as already stated above, are the behaviors. They are ordered in 2 levels. The bottom level consists of elementary behaviors, which have direct access to the environment and can therefore control it. These behaviors are simple sensomotoric behaviors, like moving forward, turning left etc. The environment itself can be stated to have two main elements: sensors, which gather information about the environment and give it to the agent, and actuators, which the agent can use to influence the environment, for example wheels for moving or a kicking device.

The higher level consists of complex behaviors, which have also access to sensoric information, but cannot directly influence the environment, e.g. they have no access to the actuators. They mirror the "moods" of the agent,

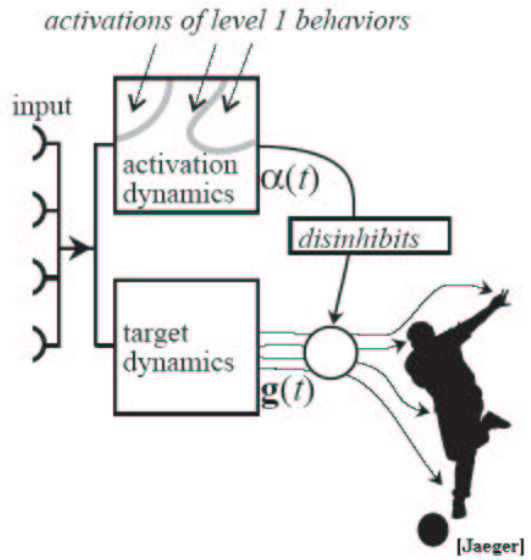


Figure 4.1: The bottom level of the Dual Dynamics Architecture of Jäger[10].
The overall architecture of the bottom level is given here. The dynamics constists of two parts: the activations dynamics on the left and the target dynamics on the right. The activation dynamics decides whether this behavior is allowed to influence actuators and how strong and the target dynamics decides which actuators to influence and how.

which makes it to activate different elementary behaviors in the bottom level and thus, behave different.

The dual dynamics scheme differenciates between two kinds of behavior dynamics. Simply explained, there is a difference between what is being done by some agent, and when; and how strong it is done. The first one is referred to as target dynamics and the second one as activation dynamics. Thus, some behaviors first have to decide whether and how strong to do something, and then what exactly to do. Only behaviors which are really active, that is, their activation status is not zero, are allowed to write target values to the actuators. The real value of the actuator, which is then performed, is computed from the target values, given by the behaviors, and the activation values of these behaviors. The idea of these dual dynamics principle is shown again in Fig. 4.1.

It is important, that the dual dynamics of the complex levels differs from those of the bottom level. The most important difference is that the bottom levels controls directly the actuators, making them do something. The higher level has no access to these behaviors. It also does not select actions from the bottom level. It is there to change the mode or the "mood" of the agent. For example, assume that two possible modes of a robot are "work" and "charge". The higher level can decide which mode to set as active. The bottom level with all its behaviors is completely independent from this decision. It provides basic behaviors, like turning left or moving forward, which can operate even if the higher level does nothing. In our example, if the active mode is "work", then the behavior "move forward" activates just when the job to be performed is in front of the robot. If the robot changes its mode to "charge", then it can still activate its behavior "move forward", but only if a charging station is in front of it. Thus, there is a difference what it does and why.

The Dual Dynamics scheme allows in general any number of levels. In this case, the bottom level (or the zero-level) is referred as the elementary level and all other higher levels as complex levels.

4.1.2 The formal model

Activation Dynamics

As stated above, the activation dynamics is one of the two main parts in the Dual Dynamics architecture and controls whether a particular behavior is allowed to access the actuators and how strong. The activation value is just a time-dependant parameter, and all activation values of all higher behaviors give the "mode" of the agent. Thus, the activation dynamics can be expressed as:

$$\alpha' = \mu_1 T_1(\alpha_1, \alpha_2, \ldots) + \ldots + \mu_m T_m(\alpha_1, \alpha_2, \ldots) \tag{4.1}$$

Where α_i are the activation values of the behaviors, the T_i are the m possible different modes of the agent and the μ_i are time-dependant factors for the modes. When the agent is in a "pure" mode, exactly one of the m modes is active and exactly its factor μ_k is nearly 1, all other are nearly zero. Mixed modes can also occur; for example if more than one factor μ_i is different from zero.

Target Dynamics

The target dynamics can be expressed as an ordinary differential equation:

$$g'_j = G(g_j, \alpha_j, I_j) \tag{4.2}$$

Where g_j is the target trajectory, time-dependant; α_j is the activation value of the behavior and I_j is the time-dependant input to this behavior, the sensory input for example.

4.2 The FU Fighters Architecture

The FU Fighters architecture is based mainly on the Dual Dynamics scheme of Herbert Jäger. However, it is been extended and slightly modified, in order to meet better the challenges and problems of real systems. The scheme of Dual Dynamics is in fact a good formal method of how to organize the behavior control, but it gives no or very little information of how to design and code the activation and target dynamics.

In this section, the main idea of how to organize and design an example behavior control based on the Dual Dynamics scheme is presented. It is the current version of FU Fighters small size behavior control.

4.2.1 Differences to Dual Dynamics

There are three main differences to the original proposal of Herbert Jäger [10]. The original proposal included a two-level architecture, where the bottom level consisted of target and activation dynamics and the complex level constisted of a restricted dynamics, namely normal activation dynamics and zero target dynamics. This means in particular, that the target dynamics exists, but merely mirrors the activation status of the behavior.

The two level architecture in the original proposal is in fact not mandatory. It can consist of more than two levels, but this provides no advantages to the whole architecture.

Thus, the first difference is that the FU Fighters', or here also referred as Extended Dual Dynamics architecture, constists of multi-level time hierarchy. The robots are controlled in closed loops that use many different

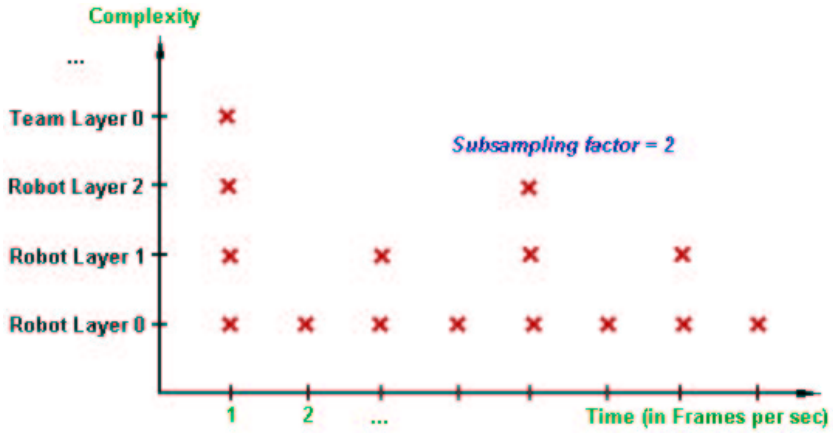


Figure 4.2: Hierarchical time management of the behavior control. The different subsampling factors and the invokings of the different levels of the hierarchy are given over the time.

time scales and correspond to behaviors on different levels of the hierarchy. Fig. 4.2 gives an idea about the time management of the different levels.

Behaviors on the bottom of the hierarchy are fast and simple. While the speed of the behaviors up the levels descreases, their number and complexity, the number of sensors and actuators increases. This allows to model very complex systems. We use temporal subsampling for the different levels of the hierarchy, thus providing room for more and more complex behaviors, sensors and actuators.

The sensors were already mentioned above. This is the second difference to the original proposal. We extended the activation and target dynamics by one more dynamics, the perceptual dynamics. It includes the sensor information, which we model as time-dependant information. Different sensors are aggregated into dynamic processes across a hierarchical model, which represents very fast information on the bottom (like ball or robot position) and more time-lasting information (like the team color or the number of robots in the team) in the highest levels.

The last main difference to the original model is the target dynamics. It is not limited to the bottom level, but has absolutely the same role in the higher levels. The complex level (all over the bottom one) have access to actuators of the lower levels, which don't represent real physical actuators, but

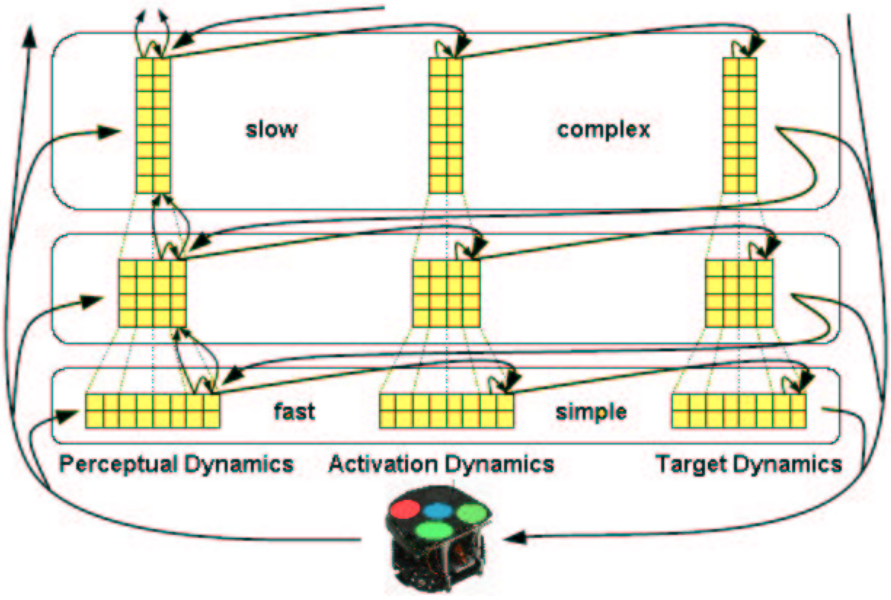


Figure 4.3: The overall architecture of the FU Fighters behavior control. The introduced perceptual dynamics is on the right, the activation dynamics (the behaviors) in the middle and the target dynamics (the actuators) are on the left. Each cell represents a constant for a given time value of the particular element, time increases from left to right. The number of elements increases goin up the hierarchy, its speed decreases. The different subsampling factors can also be seen.

simulated, virtual ones. This approach has two obvious advantages. First, all levels are designed identical. This simplifies the design and modelling process of a programming framework, which has to manage such a behavior control. This design simplifies also the programming work itself fo the system, in that it provides one common architecture. Second, the dynamics of the levels becomes more flexible and powerfull, in that it can do much more than just setting "moods" or combinations of them.

In the next sections, a detailed description of each element of the Extended Dual Dynamics architecture is given.

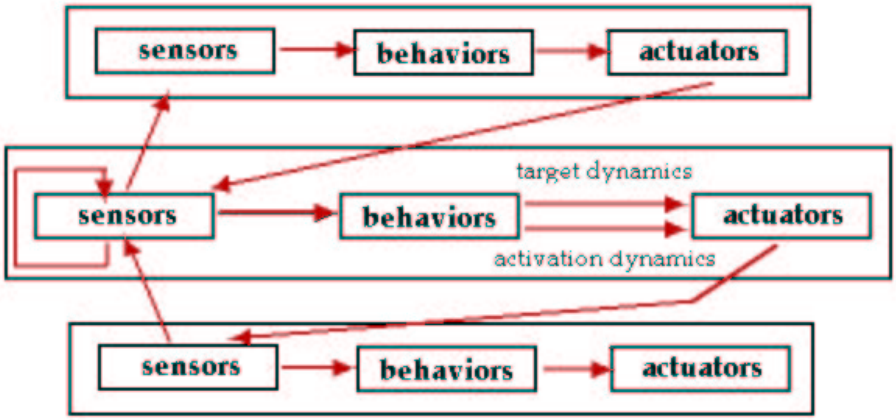


Figure 4.4: Communication between the layers.

4.2.2 Architecture of a Layer

As stated above, the main building blocks of the behavior control architecture, from design point of view, are the levels of layers. Each one consists of three dynamics blocks: target dynamics, activation dynamics and perceptual dynamics. See Fig. 4.3.

4.2.3 Communication between Layers

The different levels have to be done to cooperate and to communicate among each other. In Figure 4.4 the communication channels between the levels are shown. As one can see, there are two channels: up the hierarchy and down the hierarchy. The communication down the hierarchy is modelled with the actuators. Every behavior, according to its activation value, is allowed to change the actuators of its layer. Every actuator is connected to a sensor of the lower level and, after updating its value according to the activation values of the behaviors, which are allowed to change him, writes its new value to this sensor. The sensor can now be used as direct sensory input of the lower layer.

Besides these sensors, there are also sensors from the perceptual dynamics of each layer. They are computed in the layer itself, based on other sensors, connected with higher levels, and sensors, connected to lower layers. The last ones build the second communication channel in the architecture

– up the hierarchy. Sensors, computed in particular levels, are copied to sensors of the higher layers in order to give information up the hierarchy. For example, the position of some robot is a sensory input of the bottom layer. This information is however relevant not only to the bottom level, but to all. Thatswhy the sensor value is being propagated up the levels.

Let summarize the use of the sensors in the levels. There are three kinds of them: sensors, connected to actuators from higher levels; sensors, copied from sensors from lower levels; and sensors, computed "in place" in the particular level. The first kind, which are connected to actuators, build the top-down communication. The second kind, which are copied from sensor from lower levels, build the bottom-up communication. The third kind, which are simply computed in place, represent information relevant to levels up the current one, including it.

The updating of all three kinds of sensors has to be done in the correct order to avoid the use of old data in computation, where newer data is already available. Thus, first the sensors which come from the lower layers are updated. Then, the actuators of the higher levels hand their values down; at last, the in place computed sensors in the current layer are updated.

4.2.4 Behaviors

Above was stated, that the building blocks of the behavior control are the layers. This is so only from the design point of view. Moreover, the layers are the organising blocks of the system. The driving parts of the system are the behaviors, which define the dynamics and the complexity of the system.

Each behavior is a an autonomous, independant entity, which endues its own dynamics. The behavior uses its activation dynamics in order to decide whether it should activate itself; and its target dynamics in order to influence the provided actuators. The layer where it is placed provides the behavior its environment; that is, the speed and the complexity of it, the sensory input and the actuators.

Every behavior is thus assigned to one level. The choice of the level, where the behavior is positioned, defines how fast the behavior is and how complex. The complexity of the behaviors, together with their abstractness, increases up the hierarchy and their speed descreases. For the design of the behavior control this will mean that fast and simple behaviors, like moving to predefined point, have to be placed in the bottom layer in order to be highly reactive. More complex behavior, like planning a path, has to be

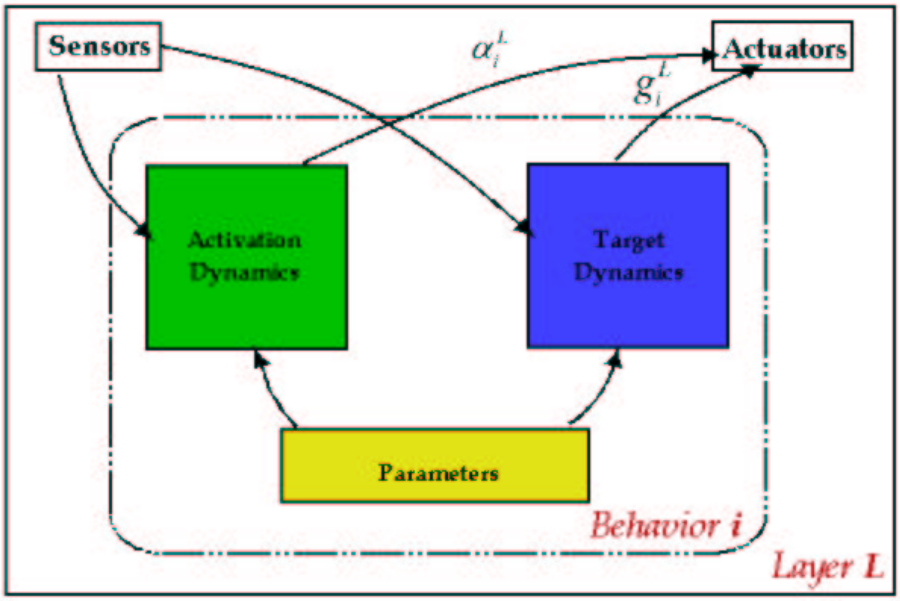


Figure 4.5: Structure of one behavior.

placed in higher levels. And, at last, very complex and abstract behaviors, like assigning roles to robots or planning the strategy of the game, have to be placed in the highest layers.

Fig. 4.5 shows the design of one behavior and how it interacts with its environment. Every behavior has its own dynamics, consisting of several parts. The target dynamics decides how to affect the provided actuators, that is, it decides *what* to do. The activation dynamics decides *whether* to do something. That is, it computes an activation factor of the current behavior, which is used by every actuator to compute how strong this behavior may influence its value. The perceptual dynamics is in this case very simple, as it just provides parameters for the behavior, in order to simplify designing, debugging and fitting the behavior.

4.2.5 The Team Layers

One interesting layer with a slightly different structure is the bottom team layer. The team layers are layers, which refers to behaviors common to all agents; that is the whole team. They decide about strategy of the team as

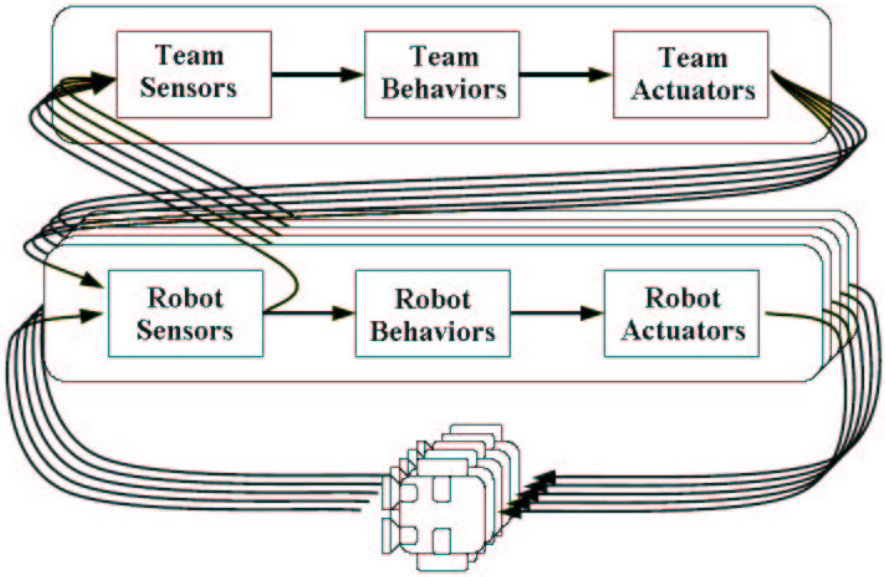


Figure 4.6: The team layers.
Every actuator of the bottom team level is cloned to provide a copy of itself to every top-level robot layer.

one agent, the roles of each robot and the coordination between the robots. Such layers are very usefull and even important in order to prevent all robots driving to the ball, for example.

The team layers have the same structure as the robot layers, except for the bottom one. Fig. 4.6 gives an idea how the bottom (or zero) team layer is different from the other ones. The bottom team layer provides, just as all other layers, actuators for the lower level. However, there is no lower layer, but many lower layers. Actually, all top-levels of each robot are connected to the bottom team layer. So, there are copies of every actuator in the zero team layer for every top robot layer.

4.2.6 Dynamics Computation

In this section, the exact computation of the dynamics of one level is presented. The dynamic system fo the sensors, actuators and behaviors can

be specified as a set of differential equations. Of course, for simplicity and performance reasons, the actual computations are difference equations.

The update of the dynamics presented below refers only to the method how the dynamics is updated. It states nothing about the real computation of these values and how they cooperate to enable a smooth and robust behavior control.

Time steps

The time runs in discrete steps, normally the computer vision frame rate (see Eq. 4.3). These time steps refer to the bottom level, the zero layer. Every higher level runs at another time steps, less frequently (see Eq. 4.4). Useful choices for the parameter f are 2,4,8, ... The FU Fighters system was tested for $f = 2$ and for $f = 1$; the last one is as updating every level every time step.

$$\Delta t^0 = t_i^0 - t_{i-1}^0 \tag{4.3}$$

$$\Delta t^z = t_i^z - t_{i-1}^z = f \Delta t^{z-1} \tag{4.4}$$

Updating the perceptual dynamics

The sensor values S_i^L of a layer L in time step t_i^L (the time step i in layer L) are updated as shown in Eq. 4.5. All sensor values are dependant on the real sensor values R_i^L connected to this layer, the previous sensor values S_{i-1}^L of this layer from time step t_{i-1}^L and all previous sensor values of the lower layer $S_i^{L-1}, S_{i-1}^{L-1}, \ldots$ at time steps $t_i^{L-1}, t_{i-1}^{L-1}, \ldots$.

$$S_i^L = update_function(R_i^L, S_{i-1}^L, S_i^{L-1}, S_{i-1}^{L-1}, \ldots) \tag{4.5}$$

By analyzing old sensor values from the layers, one can build a kind of prediction for this sensor value in the future, for example the trajectory of the ball or the future positions of the robots. This prediction can be used to cancel a delay in the system. One can also build more complex models, using the motor values of the robots and the actuator values. More about the prediction and the system delay can be found in [1].

Updating the activation dynamics

The activation values α_i^L of each behavior B in a layer L are also updated every time step t_i^L for given layer L. They depend on the previous activation values of the behaviors in the current layer α_{i-1}^L and all sensor values of the current layer S_i^L (see Eq. 4.6).

$$\alpha_i^L = activation_function(S_i^L, \alpha_{i-1}^L) \tag{4.6}$$

The choice of the elements used for updating is very simple and natural. One can use the last activation values of the current behavior or even a set of last activation values in order to make a behavior more continuous and long-lasting. Rapid fluctuations of the activation values or of the target dynamics can be thus prevented.

The sensor values are the most reasonable elements to use for updating the activation values. They provide the sensory information for the current behavior, as real sensor information from lower layers, computed sensors and values from the actuators from the higher layer.

Inhibition between layers

Another very important aspect is the inhibition between layers. An inhibition manager in every layer is presented in order to manage additionally the activation values of the behaviors. Every behavior may register itself in the inhibition manager as an inhibitor of another behaviors. The main function of the manager is to prevent circles in the inhibition structure.

Inhibition means that activation of some behavior, referred to as inhibitor, is priviliged against all of the behaviors, registered as his inhibitants. The allover activation factor (the accumulated activation factor of all behaviors in the layer) has to be normally equal to one. This quarantees in the normal case robust and well-defined behavior. Activation values of more than one are in fact allowed, but not wished.

There are two cases of inhibition: when the inhibitor is fully activated (activation factor equal to one) and when it is just partially activated (activation factor more than zero). In the first case, all behaviors, registered as its inhibitants, have the activation values of zero, regardless of their own activation dynamics. In the second case, the rest of the overall activation value of the layer is to be distributed among the inhibitants. Fig. 4.8 presents a case with one inhibitor and one inhibitant with their activation values

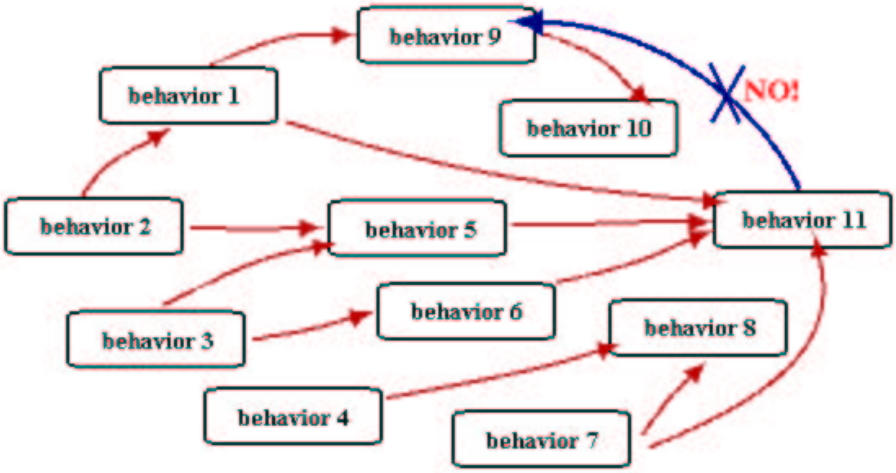


Figure 4.7: Inhibition structure between behaviors in one sample layer. A very important aspect is the order of the inhibitions. In the figure above all red arrows indicate legal inhibitions between behaviors. Th blue one is not a legal one - it builds an inhibition circle.

over the time. It is very important to understand that the inhibition is an additinal dynamics in the layer, independant from the activation dynamics of the behaviors. This special management of behaviors allows for defining "standard" behaviors, which run all the time, except they are inhibited by someone else. This guarantees that each layer definitely does something and doesn't just block.

Updating the target dynamics

Every behavior is allowed to "book" a value for a given actuator. This means that the behavior doesn't really writes to that actuator, but rather gives him a desired value. Thus, every actuator gathers the desired values from all behaviors and combines them to a single value.

Every behavior *beh* is then allowed to give for each actuator *act* a desired value g(see Eq. refeq:actor). Eq. 4.8 presents the computation of the end values of the actuators. The values of the activation factors and of the desired values for each behavior are combined; all of the values are added together.

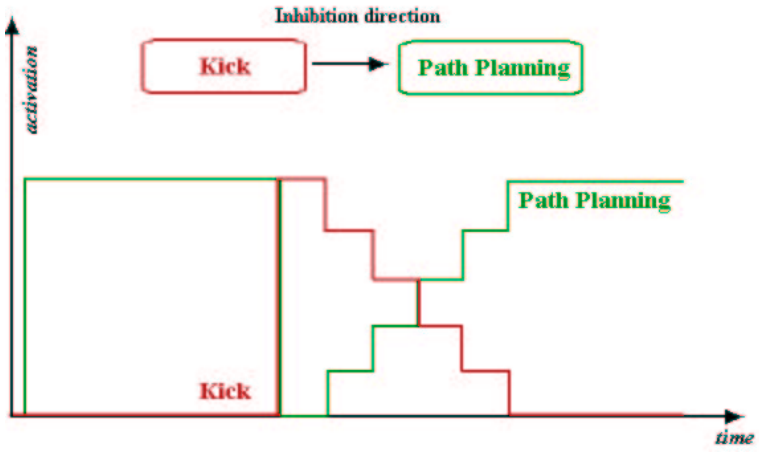


Figure 4.8: Inhibition between two behaviors.

$$g^L_{i,beh,act} = target_function(S^L_i) \tag{4.7}$$

$$act^L_i = \sum_{beh=0}^{N_{beh}} \alpha^L_{i,beh} . g^L_{i,beh,act} \tag{4.8}$$

4.3 Design and Implementation

The design and implementation of the particular behaviors is very simplified by the fact that there just small modules like the behaviors have to be designed and coded. The developers can restrict so their effrots to small amounts of source code, without keeping in mind the design of the other behaviors. Nevertheless, the structure and the communication between all elements have to be kept in mind in order to guarantee the proper functioning of the system.

The behaviors are constructed in a bottom-up fashion: first, the behaviors of the lowest layers are designed and tested with constant parameters. For example, you first need behaviors for just driving the robot from one position to another, with given orientation, speed, end speed etc. When these basic behaviors work reliable and properly, the design can be extended to the

next levels of the architecture. The second level is often used for planning specific tasks and the thrid level is used for game strategy.

It should be emphasized again that the design of the individual behaviors is independent from all others. Developments and improvements to any behavior, in a lower or higher level, should not have any effect on any other behavior. This characteristic is very important for the design of a programming framework to be used by many developers.

4.4 Examples

In this section, some examples how the behavior controls works, are given. They are taken from the actual (as to 2004) FU Fighters small size system.

Figure 4.9 presents a typical situation in a simple game. Two attackers are playing against a golie. The situation is pretty clear to a human observer: the yellow robot has to drive first to the ball, pass it to the other yellow robot, which has to kich the ball.

Below the field views the activation values of all interacting behaviors are given. One can see, how they activate themself and change the whole behavior of the robots.

4.5 Challenges

In this section, I try to describe the main challenges of designing a programming framework for behavior control. In the sections above, the behavior control and its architecture in their current state were presented. It is now time for a little discussion of this architecture.

The Extended Dual Dynamics scheme is a very powerful and highly organized scheme for programming behavior of mobile robots. However, the scheme is just a formal method how to organize the architecture and gives no idea how the behaviors work, how they influence the actuators and the target dynamics as a whole. Making it short, the system is very vulnerable to errors and very unfriendly for debugging and tuning. This unfriendiness comes mainly with its complexity and particularly with its size (refer to Chapter 1 for some statistics of the system). A developer working on it has to keep in mind the whole element architecture with all layers, sensors, behaviors and actuators; the communication among them and the cooperation among them. Typical examples are the inhibition structure of the behaviors or the computation order of the sensors in the layers.

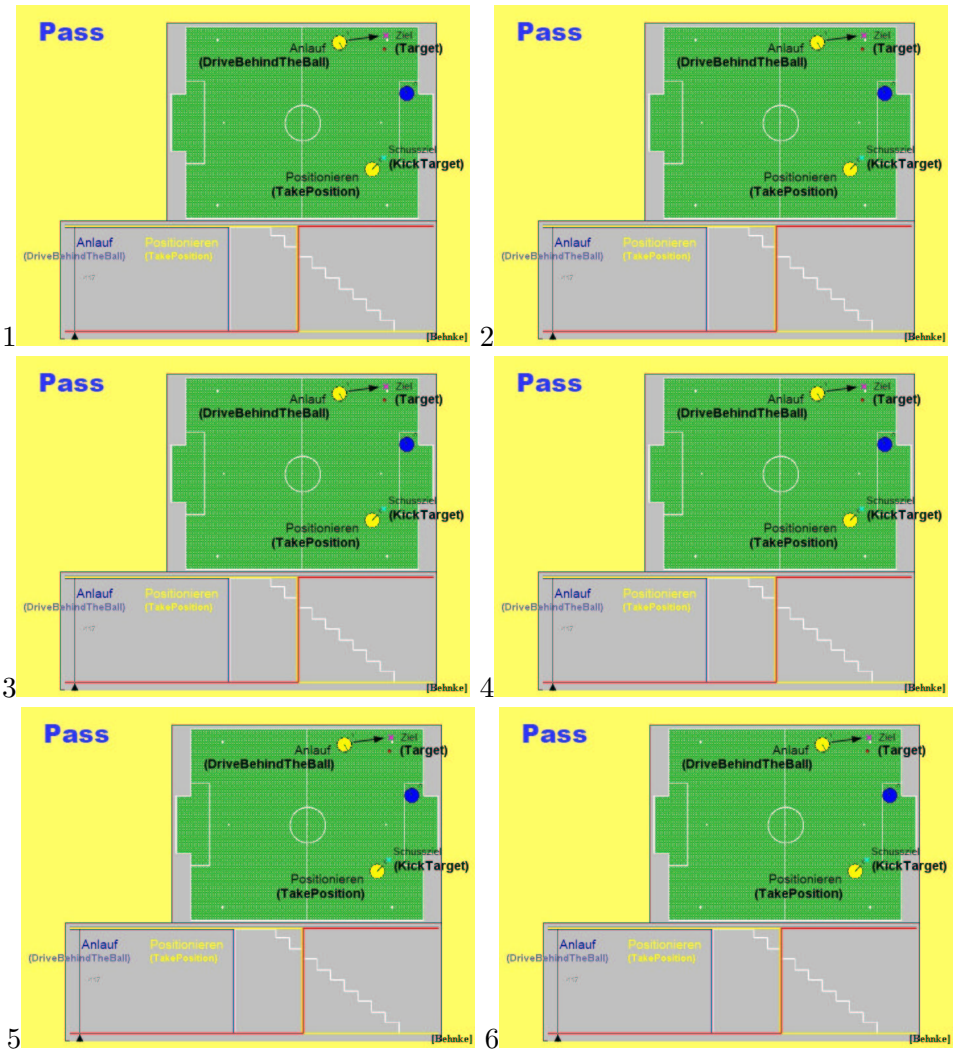


Figure 4.9: Behavior interaction example.
Six scenes from the behavior control of robots are given here. The activation values are given below the field view.

Thus, a programming framework has to fullfil several main conditions and provide enough room for designing various tools and adding them to the framework. Main criteria include among other things the following:

- *Meeting the requirements of the particular FU Fighters architecture.* The behavior programming framework has to be designed especially for the needs of the FU Fighters Extended Dual Dynamics scheme. A more general framework would have the advantage of managing more systems and better meeting changes in the allover architecture. However, this will also restrict the power of the tools and the automatization of the processes, because they will be no more clearly defined. It should be also noted, that the behavior architecture has experienced no significant changes since its introduction. Thatwhy, a dedicated programming framework will be the better choice.

- *Graphical overview of the architecture.* The whole architecture, for example the layer hierarchy and the organisation of the behaviors in the layers, should be clearly seen. A proper naming of the layers (for example, RobotLayer0 is the bottom layer, RobotLayer1 the middle, and RobotLayer2 the highest layer of the robots) does already partially the job.

- *Graphical tools for editing hierarchical properties.* Some of the properties of layers, such as the computation order of the sensors or the inhibition between the behaviors, could be perfectly managed in a graphical tool. Other tools, for example for debugging or gathering statistical information, would also be nice.

- *Providing a graphical user programming interface.* Like many other programming interfaces, the behavior programming framework should provide the user with some useful functionality, which will simplify the tough software developer's life.

- *Automatic management of standard processes.* There are many standard processes in the programming of the behavior control. For example, adding a new sensor or a new behavior is connected with much work, which is almost routine. Such processes should be automated in order to increase the developer's productivity and to prevent errors.

- *Giving access only to relevant code.* For safety reasons, some parts of the behavior control code should be invisible to the developers. For

example, the architecture and the hierarchy itself should not be altered so easily. This will prevent "dirty code" and errors.

- *Good code readability.* The automatic management of routine processes and all of the tools requires source code management. It should be considered that the generated code should be very good readable to human programmers. This should quarantees, that the framework could be changed or abandoned at any time. Compatability of the generated source code with older versions of the software should be provided.

Chapter 5

MAAT - System Overview

The MAAT graphical programming framework is a dedicated programming interface which manages the behavior control architecture of the FU Fighters small size team. The behavior control is an Extended Dual Dynamics behavior-based scheme, as described in Chapter 4.

The MAAT software was developed in C++ in Microsoft Visual Studio 6.0. The Qt Library 3.2.2 from Trolltech[1] was used for designing the graphical user interface.

In this chapter, the structure of the framework, its tools and graphical user interface is presented. First, an overview of the inner organisation of the framework is given, as also an overview of the connection between the behavior control architecture and the MAAT architecture. Next, the graphical user interface and the single tools are presented.

5.1 System Overview

The MAAT architecture consists of two main parts: a display manager and a maat manager (see Fig. 5.1). Both units are visible for all elements in the framework and work together to assure smooth managing and displaying of the elements.

The display manager represents on the one side the main window of the application and, on the other side, holds all displaying tools of it. The displaying tools themselves are autonomous units, which "know what to do" and communicate via messages with all other units in the architecture. For

[1] www.trolltech.com

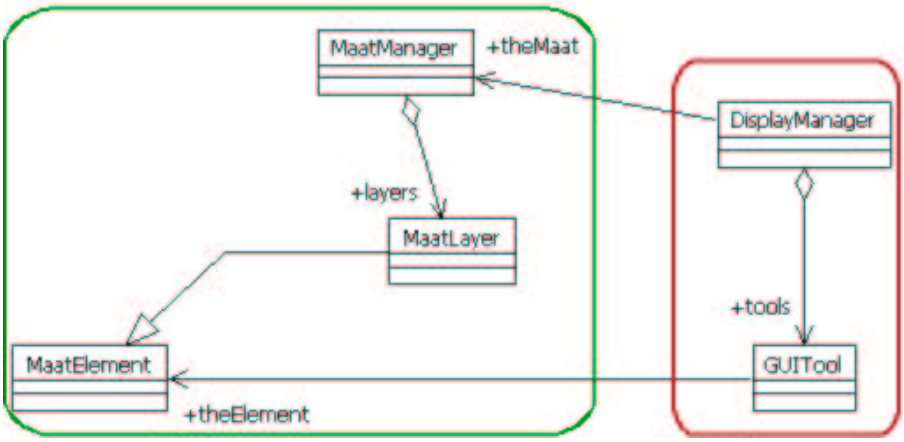


Figure 5.1: The overall structure of the MAAT.
There are two main units: the display manager and the MAAT manager. The display manager is responsible for the correct and up-to-date displaying of the elements. The function of the MAAT manager is to administer the architecture of the behavior control. Both elements are independent from one another and visible for all elements in the whole MAAT structure.

example, the ElementViewer displays information about the current selected item - sensor, layer, behavior etc. Selecting new item sends a message, which is received by the ElementViewer. The ElementViewer processes the message and updates the displayed information. The whole process is very simple and well organized. The messages contain only the necessary information - in this case that a new item is selected. Access to additional information is gained via the MaatManager for all elements, which are interested in this information.

The MaatManager is the one responsible for managing the behavior control hierarchy. It holds the information about the layers, the behaviors, sensors and actuators in them; manages these elements and take care of the translating and saving of them. The manager communicates also via messages with all other elements in the framework. The most important communication here is between the MaatManager and the displaying tools, which edit the hierarchy's information - for example, adding a new sensor to one layer.

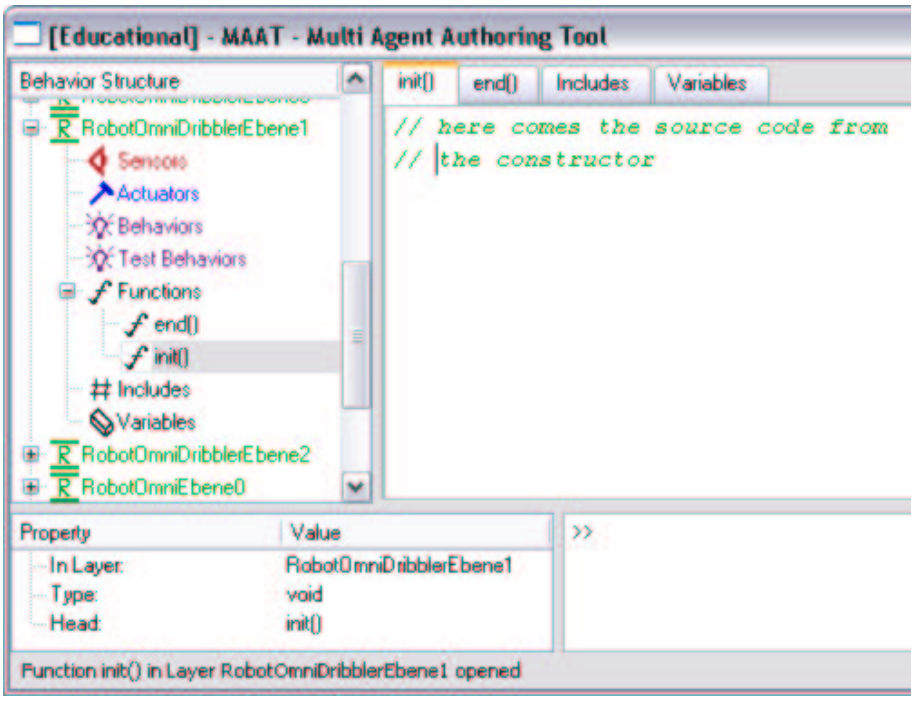


Figure 5.2: MAAT code management.
The developer is gained access to a special function *init()*, invoked by the layer's constructor. Thus the automatically generated code in the constructor remains hidden and the developer can still use the functionality of the constructor.

5.2 Code Management

The most important goal of the framework is to manage the code of the behavior control. This requires a comfortable user interface and a smart code management.

The MAAT framework gives access for the developer only to selected parts of the code. For example, user-defined computations like functions, activation dynamics and target dynamics computations and other similar code fragments are freely accessed by the developer. They are not part of some source code file, but managed as autonomous units, which are then translated into "normal" C++ code, organized in header and implementation files.

The advantage of this code organisation is first the safety of the code. The developer cannot change parts of the source code, which could eventually damage the architecture or the cooperation between the single elements. The behavior architecture itself could not be altered in the framework. As stated above, the programming framework is a dedicated interface for managing the current behavior control scheme of the FU Fighters small size team. Therefore, all changes made in the architecture itself cause also changes in the framework. However, some small alterations, like extending the number of involved layers, could be made easily.

Another advantage of hiding parts of the source code is the comfort of the developers. The framework is designed especially for programming the behavior control of the small size robots. The developers are interested thatswhy only in parts of the code, which are relevant to this control. All other things, like initialising and registering the sensors or actuators, like managing the class hierarchy etc., are irrelevant and bothering. Hiding them makes the developers concentrate on their work and on the behavior programming only.

Figure 5.2 gives an example of the code management. Assume that the developer needs access to the class' constructor. The constructor of a layer itself is hidden, because of the automatic registration and management of sensors, actuators and behaviors in the layer. However, the developer is gained access to a function *init()*, which is invoked by the layer's constructor. The advantage in this case is obvious: the parts of the constructor which are irrelevant for the developer are still hidden, but the access to the functionality of the constructor is ensured.

5.3 MAAT Class Hierarchy

As stated above in this document, the MAAT architecture simply mirrors the whole hierarchy of the behavior control architecture in order to manage the elements in it. The MAAT elements are derived partially from the behavior control elements, such as layers or behaviors, but have also some other properties.

The main element in the MAAT architecture is the MaatElement. It is a common class for representing an element for displaying and managing. All other elements, such as layers, sensors etc., are derived from MaatElement. Figure 5.3 shows all MAAT elements and their class hierarchy. Besides MaatElement, MaatEbene is a very important class.

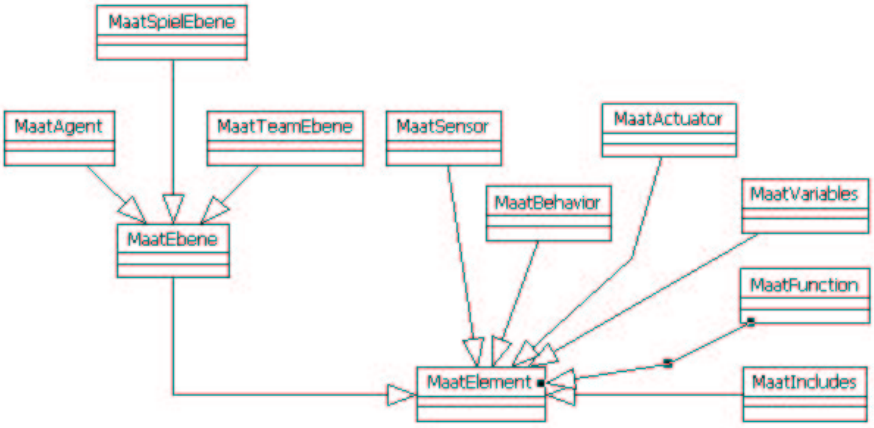


Figure 5.3: MAAT class hierarchy.
The common parent of all elements is the MaatElement. The other very important class is the MaatEbene (a MAAT layer).

5.3.1 MaatEbene

MaatEbene represents a container class, which holds other elements. There are two types of elements, which are managed by a layer. The first ones are the typical and well-known elements from the behavior control - sensors, actuators and behaviors. The other type of elements are C++ typical ones, like functions, initialisation and ending functions (invoked respectively from the constructor and the destructor of a layer), container for included external libraries etc. These elements have nothing to do with the rest of the behavior architecture, but are important for the proper translation of the code to C++ code.

Besides holding and managing elements, the MaatEbene's functionality include saving and loading of the properties of the layer into and from XML-files and translating the layer into normal C++ code. Figure 5.4 presents the class diagram of the MaatEbene class.

There are a set of elements, derived from the MaatEbene class. They represent either special layers, such as team layers, or agents. An agent is an important unit in the behavior control architecture. The agent represents an autonomous part of the system, such as one robot or the whole team, and holds information about specific properties of it. Every agent has its behavior layers, which are invoked time-dependant.

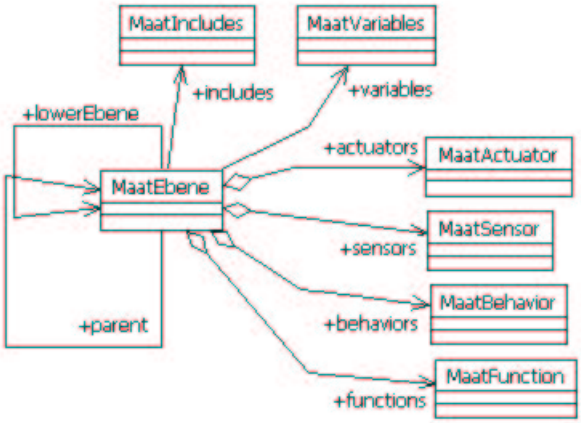


Figure 5.4: MaatEbene class diagram.
MaatEbene holds all of the information about sensors, actuators and behaviors in one layer, manages them via the elements' management methods and takes care of saving, loading and translating the layer via the layer's management methods.

From the point of view of the programming framework, a team layer or an agent are very similar to a normal layer. For the first one, the team layer, the difference is only in the zero team layer, where the actuators are not connected to one robot layer, but to a set of them (one for each robot of the team). Thus, the MaatTeamEbene is inherited from MaatEbene and its functionality is extended in the case that it presents the zero team layer.

An agent can be modelled also very easily by a layer. Please note, that this modelling is based on properties relevant to the programming framework and not to properties of the agent itself. In the behavior architecture, the agent has a very different role from that of a layer. So, in the programming framework, the agent contains sensors, which can be accessed from other layers in the behavior architecture, but cannot be changed. Thus, the contained information in an agent is not modelled or managed by the MAAT framework, but just mirrored in order to gain access to it.

MaatAgent is inherited from the MaatEbene and its functionality is restricted to displaying the information of the agent.

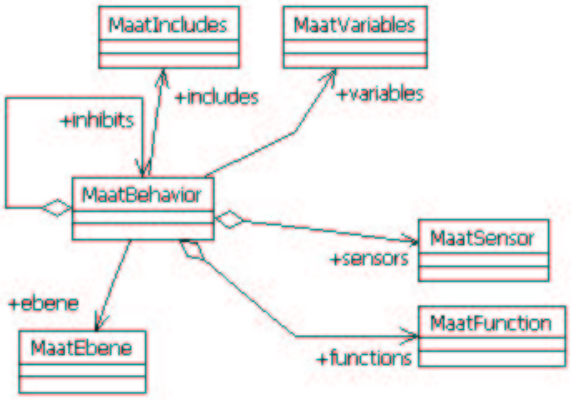


Figure 5.5: MaatBehavior class hierarchy.
A behavior holds information about sensors and functions, they represent the parameters of the behavior and its dynamics, activation and target.

5.3.2 MaatBehavior

The other very important unit in the behavior architecture is the MaatBehavior. It is inherited directly from the MaatElement. It is very similar to a layer in its inner design (see Fig. 5.5). It belongs to some layer and holds information about sensors and functions. The sensors, which are part of a behavior, are parameters of the behavior, which can be changed from the FU Fighters graphical system. The functions of a behavior can be separated into two groups: the dynamics computation methods and supporting functions. The first ones are the well known activation and target dynamics computation's methods, which are fixed part of each behavior. The other ones are normal class methods, used for additional computations.

Thus, the design of one behavior is similar to the design of a layer. It holds sensors, functions and other additional elements. It also manages the source code of the behavior by saving, loading and translating it into C++ code.

Beside the normal behavior there is also a so called test behavior. It is a special type of behavior, which activation function is not computed, but set manually to one or zero. Activating such a behavior causes manual deactivation of all elements in the same layer. These special behaviors can be used for testing the team or an individual robot, letting it for example drive

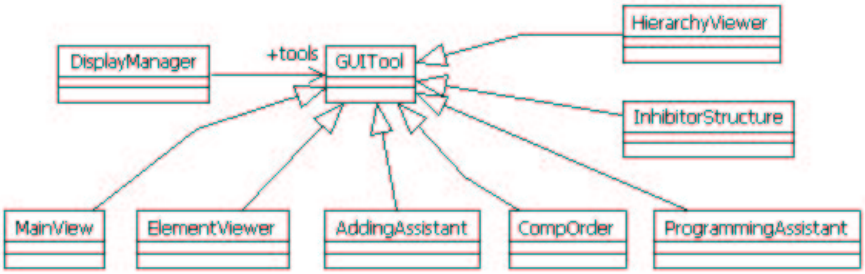


Figure 5.6: Display Manager.
The overall structure of the display manager and its associated GUI tools.

in a circle. There is no special test behavior class, because the differences between a normal behavior and a test behavior are negligible. The only difference by the implementation of a test behavior is the lack of an activation function. This property is implemented as a class field of the MaatBehavior.

5.4 The Display Manager

As already stated above, the Display Manager of the programming framework manages all of the GUI elements in it. It consists of a main window (the main window of the application at the same time) and a set of GUI tools. These are autonomous units, which have access to the MaatManager. They communicate with other parts of the framework via messages (the signals and slots scheme of Trolltech's Qt). Fig. 5.6 shows the overall structure of the Display Manager and its individual parts (the GUI tools).

5.4.1 GUI tools

There are several tools in the MAAT programming framework. Here is a list of the most important ones:

- *Hierarchy Viewer.* This is the most important tool, which gives the programmer the opportunity to select various elements and to display them in the main view. The layers are listed here, with all of their elements like behaviors, test behaviors, sensors and actuators. Clicking on a layer for example, has as result the displaying of all functions of

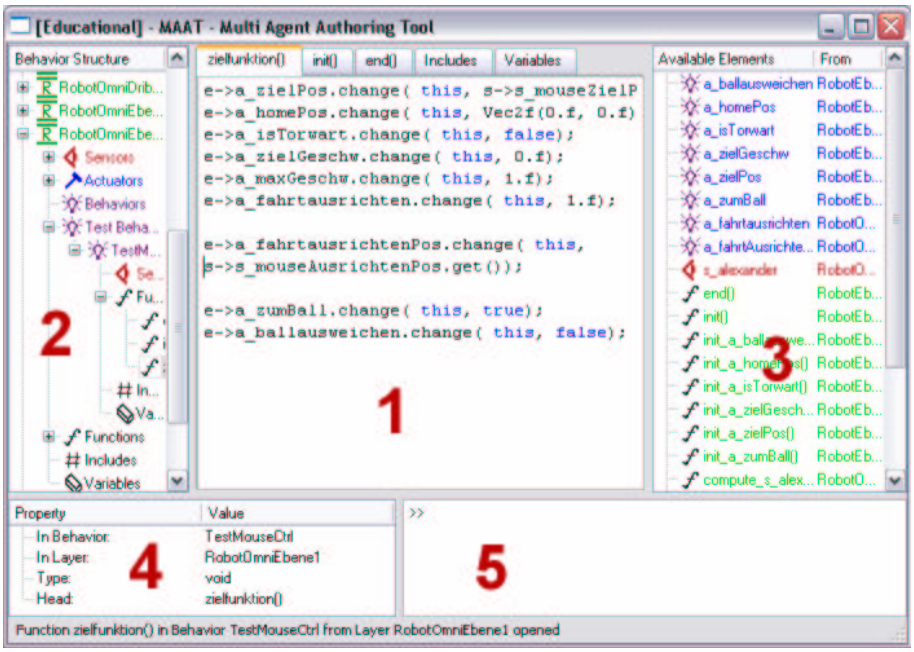


Figure 5.7: View of the MAAT framework.
All of the main GUI elements are given in this figure. (1) the main view, (2) the hierarchy viewer, (3) the programming assistant, (4) the element viewer, and (5) the debugging output.

this layer in the main view. (See Fig. 5.7 for an example). One can select either layer for displaying or a behavior (or a test behavior).

- The *Element Viewer* shows always all of the properties of the current element, selected in the hierarchy viewer (Fig. 5.7).

- *Programming Assistant.* This assistant displays all the time an on-time list of all elements, which can be accessed from the current layer/behavior. This a great help to the programmer, as it shows all class elements of the current element, as sensors, behaviors, actuators etc. the whole class hierarchy down (see Fig. 5.7).

- *Main View.* This is the area, where the programmer really works. The main view consists of tabs, which each represent a function from the displayed layer/behavior. It is being updated every time the user double clicks an element from the Hierarchy Viewer. If the element is

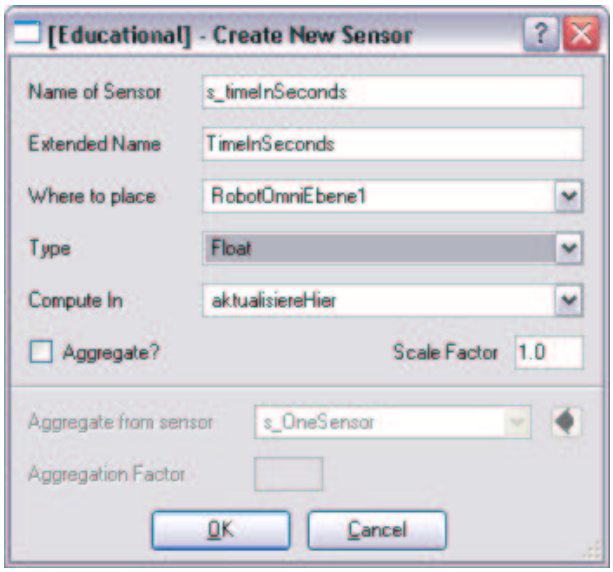


Figure 5.8: Sensor adding assistant.
The assistant can not only create a new sensor, but set all of its properties, propagate the sensor down the hierarchy etc. Similar assistants exist for creating actuators, behaviors, test behaviors, functions and layers.

a layer or a behavior itself, the main view doesn't change, but only if the clicked element is an element of a layer or behavior.

- *Element Adding Assistants.* There are also a number of assistants for adding new elements - layers, behaviors, sensors, actuators etc. See Fig. 5.8 for an example.

- *Inhibitor's Structure.* The architecture of the inhibitors and inhabitants in a layer was discussed earlier and found to be very important for the smooth functioning of the whole behavior control. The programmer can see in this tool which behaviors are inhibited or inhibit themselves others.

- *Computation Order.* The sensors in a particular layer have to be calculated in a particular order to guarantee actual values for all sensors. This tool shows the computation place of each sensor.

Chapter 6

Results

The most important goal of the MAAT Programming Framework is the automatization and simplification of the programming process for the autonomous mobile robots of the FU Fighters' team. The programming framework was implemented in order to accept this challenge and in order to give the oppurtunity to continuely change and improve the framework.

In the next section, an overview of the most important properties of the programming interface is given again (see also Chapter 4). Then, some examples of the achieved results is given. However, one should keep in mind, that a fully objective and countable results are impossible to be given. Some results, as for example the automatization of some processes or improving the speed of others are given. Others, like the simplicity of use, user friendiness etc. are fully subjective and the reader is kindly invited to review them.

In order to meet the challenge to design and build a user friendly, simple and intuitial interface for programming behavior control for autonomous mobile robots, some properties were defined to be crucial or very important ones. The properties can be divided into two groups: user interface and source management.

6.1 User Interface

The most important graphical property to be implemented is a user friendly view of the behavior control architecture. This requeirement was met with the *Hierarchy Viewer* (Fig. 6.1. It contains a list of all layers and each layer

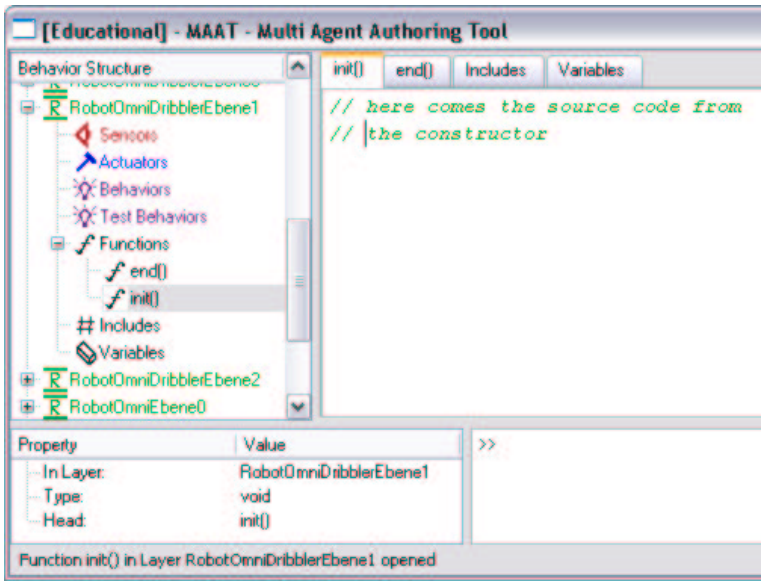


Figure 6.1: The Hierarchy Viewer.
Every layer contains groups with its behaviors, test behaviors, sensors, actuators and functions. Clicking on them activates the layer or behavior in the main view.

has groups of its behaviors, test behaviors, actuators, sensors, funcations etc.

The Hierarchy Viewer implements also another important feature of the graphical user interface - *editing hierarchical properties*, like adding and removing elements, changing the computation order ot sensors, managing inhibitors etc. (see Fig. 6.1).

There are many standard processes in the programming of the behavior control. For example, adding a new sensor or a new behavior is connected with much work, which is almost routine. Such processes should be automated in order to increase the developer's productivity and to prevent errors. The processes are not only faster, but also more reliable and safe.

Another aspect of safety is the access only to relevant code. Some parts of the source are irrelevant for the programmer, but very sensitive to errors and changes, such as the behavior structure itself, for example. Such changes should be not permitted in the framework, in order to provide the needed safety.

6.2 Source Management

As stated already, the behavior programming framework should be a dedicated one, which manages the source code of the current FU Fighters behavior control. The behavior control itself do not experience great changes, so these can excluded from the programming interface control.

On the other hand, a dedicated system gives more opportunities for automatization of processes and better source handling.

The MAAT behavior control is built especially for the FU Fighters' current architecture. However, it supports any number of layers and levels, making room for some changes and tunings of the architecture. The building of additional elements is very simple (they could be all inherited from MaatElement, see also Chapter 5) and makes the system more flexible and adaptive.

Another important aspect is good code readability. Although source code is managed fully automatically, the possibility to change every time to the old programming interface should be provided. The source code is therefore to be readable and self-explainable. This property is provided by the MAAT framework, as the translation of the source code makes sure the code is commented and ordered.

List of Figures

List of Tables

Bibliography

[1] Behnke, S.; Egorova, A.; Gloye, A.; Rojas, R.; and Simon, M.: Predicting away the Delay, in N.N. (editors): *RoboCup-2003: Robot Soccer World Cup VII*, Springer, 2004

[2] Behnke, S. and Rojas, R.: A hierarchy of reactive behaviors handles complexity, in: Proceedings of: Balancing Reactivity and Social Deliberation in Multi-Agent Systems, a Workshop at ECAI 2000, the *14th European Conference on Artificial Intelligence*, Berlin, 2000.

[3] Bredenfeld, A.; Indiveri, G.: Robot Behavior Engineering using DD-Designer. *IEEE International Conference on Robotics and Automation (ICRA 2001)*, Seoul, Korea, May 23-26, 2001.

[4] Bredenfeld, A.: Intergration and Evolution of Model-Based Tool Prototypes. *Proceedings of 11th IEEE International Workshop on Rapid System Prototyping (RSP 2000)*, Paris, France, June 21-23, 2000.

[5] Bredenfeld, A.: Behavior Engineering for Robot Teams.

[6] Briggs, G. and MacDonald, B.: A Survey of Robot Programming Systems, *University of Auckland*.

[7] Egorova, A.; Gloye, A.; Liers, A.; Rojas, R.; Schreiber, M.; Simon, M.; Tenchio, O.; and Wiesel, F.: FU Fighters 2003 (Global Vision), in N.N. (editors): *RoboCup-2003: Robot Soccer World Cup VII*, Springer, 2004

[8] Gloye, A.; Göktekin, C.; Egorova, A.; Rojas, R.: Learning to Drive and Simulate Autonomous Mobile Robots, submitted for *Robocup International Symposium 2004*, Lissabon, Portugal.

[9] von Hundelshausen, F.; Rojas, R.; Wiesel, F.; Cuevas, E.; Zaldivar, D.; and Gunarsson, K.: FU-Fighters Team Description 2003, in *D. Polani,*

B. Browning, A. Bonarini, K. Yoshida (Co-chairs): RoboCup-2003 - Proceedings of the International Symposium.

[10] Jaeger, H.; Christaller, T: Dual Dynamics: Designing behavior systems for autonomous robots. *Artificial Life and Robotics*, 2:108-112, 1998.

[11] Kobialka, H.-U.; Jaeger, H.: Experiences Using the Dynamical System Paradigm for Programming RoboCup Robots.

[12] Lespérance, Y.; Tam, K.; and Jenkin, M.: Reactivity in a Logic-Based Robot Programming Framework, *York University, Toronto.*

[13] MacDonald, B.; Yuen, D.; Wong, S.; Woo, E.; Gronlund, R.; Collett, T.; Trépanier, F.-E.; Biggs, G.: Robot Programming Environments, *Department of Electrical and Electronic Engineering, University of Auckland.*

[14] Rojas, R. : The Challenge of Robotic Soccer. *www.fu-fighters.de*

[15] Simon, M.; Behnke, S.; and Rojas, R.: Robust Real Time Color Tracking, in: *Proceedings of: The Fourth International Workshop on RoboCup*, pp. 62-71, Melbourne, Australia, 2000.

[16] Trèpanier, F.-E.; and MacDonald, B.: Graphical Simulation and Visualisation Tool for a Distributed Robot Programming Environment, *University of Auckland.*