An experimentation platform for MARIE.

master thesis

E. de Ruiter, 9742808
University of Amsterdam
edruiter@science.uva.nl

March, 2003

Contents

1 acknowledgements 4

2 introduction 5

3 theory 7
- 3.1 introduction . . . 7
- 3.2 parallelism . . . 7
- 3.3 Minsky's frame theory . . . 7
- 3.4 Arkin's scheme theory . . . 11
- 3.5 Frames and schemes compared . . . 14

4 autonomous systems 15
- 4.1 introduction . . . 15
- 4.2 functional architectures . . . 16
- 4.3 behavioural architectures . . . 18
- 4.4 hybrid architectures . . . 19

5 MARIE 21
- 5.1 hardware . . . 21
- 5.2 software . . . 21
- 5.3 some experiments . . . 24

6 the wall-sensor 25
- 6.1 introduction . . . 25
- 6.2 the model of the environment . . . 25
- 6.3 handling uncertainty in the sensor-values . . . 27
- 6.4 new line-segments . . . 27
- 6.5 implementation details . . . 28
- 6.6 wall-searching . . . 28
- 6.7 corridor-searching and following . . . 29

7 the task-planner and the action dispatcher 30
- 7.1 action dispatcher . . . 30
- 7.2 task-planner and scenarios . . . 30

8 other modules and their parameters. 32
- 8.1 wall-sensor and wall-follower. . . . 32
- 8.2 path-planning and trajectory controller . . . 33
- 8.3 collision avoidance . . . 33
- 8.4 sensor-based controller . . . 34
- 8.5 docking-spot detector . . . 34
- 8.6 simple operations . . . 34

9 experimentation **35**

10 experimentation platform **36**
10.1 introduction 36
10.2 design 37
10.2.1 parameter optimization 37
10.2.2 fitness-function; measure for success 38
10.2.3 selection 40
10.3 implementation architecture 41
10.3.1 explanation of terms 41
10.3.2 prints-statements 42
10.3.3 the database 44
10.3.4 implementation details 46
10.4 module as the most stable element 48
10.5 results 48

11 EP for other modules **50**
11.1 future work 52

12 the virtual laboratory; a grid based system for experimental science. **53**
12.1 the grid 53
12.2 virtual laboratory : VLAM-G 54
12.3 EP for MARIE and VLAM-G; differences and similarities. 55

13 conclusion **57**

A odl-schema **59**

B C-source for inserting an executable **62**

1 acknowledgements

Finally my thesis is finished. It took a little longer than we expected, mostly because of me and my, sometimes troublesome, relationship with MARIE. Without the help of some people that made this thesis possible I never would be able to write this work. Therefore I first of all want to thank Arnoud Visser for giving me the opportunity to do this project and for his help in sometimes frustrating matters. Without his insigths and comments this thesis would never be possible.

I want to thank my parents and girlfriend for supporting me financially and emotionally during some hard times. Also I like to thank Frank Terpstra for proof-reading and comments especially on the style and structure of the text. His help when I was first introduced to MARIE was also very useful.

Many thanks to Koen Boinck who worked on a different project involving MARIE. Discussions I had with him, especially on technical issues, were sometimes very enlightning. (It's good not always have to figure things out yourselves.)

Last but not least I would like to thank the Golden-Boys for simply being there during the course of my study. Special thanks (within the Golden-Boys-thanks) to Barry Koopen, Matthijs Spaan and Wouter Caarls for help with lateX, playing tabletennis and making the coffee-breaks a lot more pleasant.

2 introduction

In this thesis I present my work on the experimentation platform for the MARIE mobile robot. MARIE stands for Mobile Autonomous Robot in an Industrial Environment.
An autonomous robot is a special kind of computer system with a certain degree of autonomy. This makes it possible to operate in real-time environments that often contain a lot of uncertainty. Just as any other computer system an autonomous robot works with memory in which it can store useful information. Because the robot is mobile you can think of this information as being for instance the start, current and goal position of the robot according to some given coordinate frame. This sort of data can be viewed as a human-like short-term memory, information that is used and necessary at a certain point during operation.
As with humans short-term memory can be viewed as being 'intelligence', while long-term memory can be seen as being 'wisdom'. The experimentation platform described in this thesis extends this memory with a long-term memory based on old experiments. Just as in most computer systems some sort of long-term memory is already present in the form of logfiles. These are files that, during execution, are filled with information on what happens internally within the system.
MARIE is a vehicle based on a Vesa Trekka cart for disabled people. A lot of software is written for MARIE and a lot of experiments have taken place to verify this software. Because it is often difficult to interpret the large and hard to understand logfiles mentioned above, I tried to design and implement a platform that makes it possible for a user that has little or no experience with the robot, to easily determine a set of good parameters for the different software-routines. Also the platform should make it possible to select an good or even optimal set of module and executable-versions. In other words, the system is like an advanced logging-system that is able to extract useful information from experiments more easily.
For instance in the RoboCup project a team of soccer-robots was build, they could use a platform like this for finding the information on what aspects are essential for winning.

So there are actually two questions that I'm trying to answer in this thesis:

- is it really possible to gain more insight in the experiments done with MARIE using the described experimentation platform?

- has this sort of long-term memory effect on the success-rate of experiments with MARIE?

All this is based on theories by Arkin (schemes) and Minsky (frames). I will start with a description of these theories, next I will describe the field of autonomous systems, especially behaviour-based robotic systems. Then I will explain the current wall-sensor module, a virtual sensor for searching and following a wall. Also my own contributions to this software will be described. Also I will briefly go over some other modules.
The rest of this thesis is about the experimentation platform which I have implemented for the wall-sensor and therefore will describe it using the wall-sensor as an example. At he end I will briefly explain how I would make my software generic for the other modules that MARIE works with.
Because the work of CAPS group at the University of Amsterdam on the virtual laboratory inspired me to a certain degree to write this thesis, I will briefly describe their work at the end.

3 theory

3.1 introduction

Researchers in robotics often use existing theories of cognition on their robots. Most of the time these theories are borrowed from psychology, but sometimes they adapt or extend them for the specific use within an autonomous system. Especially theories on memory have inspired much of the research in robotics. The way memory is used to handle current situations obviously the most important issue when designing a learning system. In the following sections I will describe two theories dealing with memory and control which are actually very similar but not the same. The first, frame-theory however is used for explaining human cognition, the second, scheme-theory, focuses on robot navigation.

3.2 parallelism

Within both theories described below the notion of parallelism plays an important role. With a parallel cognitive system I mean units of cognition that work together to gain intelligent behaviour of some kind (navigation, recognition, etc..).
As we will see in the section on behavioural architectures, parallelism can be used within robotics to not only speed up the computation, but also to keep the modules involved as simple as possible. There is evidence that parallelism also plays a role in natural intelligence. Research at MIT has shown that when a certain region on the spinal cord of a frog is stimulated by touch, its behaviour looks like it generates a force field directing the frog to a specific location. When two or more regions are stimulated the outcome is similar to the corresponding force fields combined by simple vector addition resulting in a new force field for control. Scheme-based robot navigation, which I will describe in section 2.4, is based on these results.

3.3 Minsky's frame theory

In the early 70's Marvin Minsky constructed a theory for representing knowledge, that could better handle things like common sense reasoning, perception and language than the existing ones could. According to Minsky all cognitive processes use agents that are as simple as possible. A coordinated process of placing them in a network of relations generates human behaviour.
He stated that when someone is presented with a new situation,

whether in language, perception or in other mental processes, he selects from memory a context-structure which Minsky calls a *frame*. A frame can be viewed as a network of relations. The top-level represents facts about the situation, things that are always true in this sort of situation. The lower-levels have many terminals that must be filled in with specific data, this is like filling in the details. Each of these terminals can have certain conditions that must be met by the data that is assigned to the terminal. Examples of conditions for terminals are for instance that the data must represent a person or a certain object. Data can also represent a pointer to a sub-frame or a relation between two or more terminal assignments. A frame-array is defined as a collection of frames that share the same terminals. Usually the terminals are already filled in with default data, these can be seen as representing what can be expected. When the terminal-values are not corresponding with the terminal-conditions they can be easily adapted to the current situation. Minsky distinguishes two types of memory structures; *polynemes* and *pronomes*. Polynemes represent long-term memory objects, such as names and places. Pronomes can be filled at the time they are necessary and can therefore be viewed as short-term memory objects. To model the dynamically changing real world Minsky introduces systems of frames. These can represent actions by transformations between the frames of a frame-system.
When we are looking at a cube from a certain angle (see figure 1), we see side "A" on the left and side "B" on the right. To represent this knowledge in a frame we simply attach "A" to a "left-side" terminal of a cube-frame and "B" to a "right-side" terminal of the same cube-frame. When we change view by moving to the right, "A" becomes invisible and side "B" becomes the new face on the left, while side "C" becomes the new face on the right. Using frames we don't have to do any perceptual computation to represent this new knowledge because we can simply attach "B" to a "left-side" terminal of a second cube-frame and store "C" at the "right-side" terminal of this second frame. To remember "A" we can simply attach it to a "invisible" terminal of the second cube-frame. When this method is repeated for different viewpoints a frame-system will be created consisting out of different frame representing different perspectives. The entire frame-array therefore represents movement-actions. As mentioned before the terminals carry conditions, constraints and/or expectations. These can be used in the matching process that selects appropriate frames for the current situation. The current goals determine whether or not the selected frame is suitable. Frames are usually initiated by their expectations.

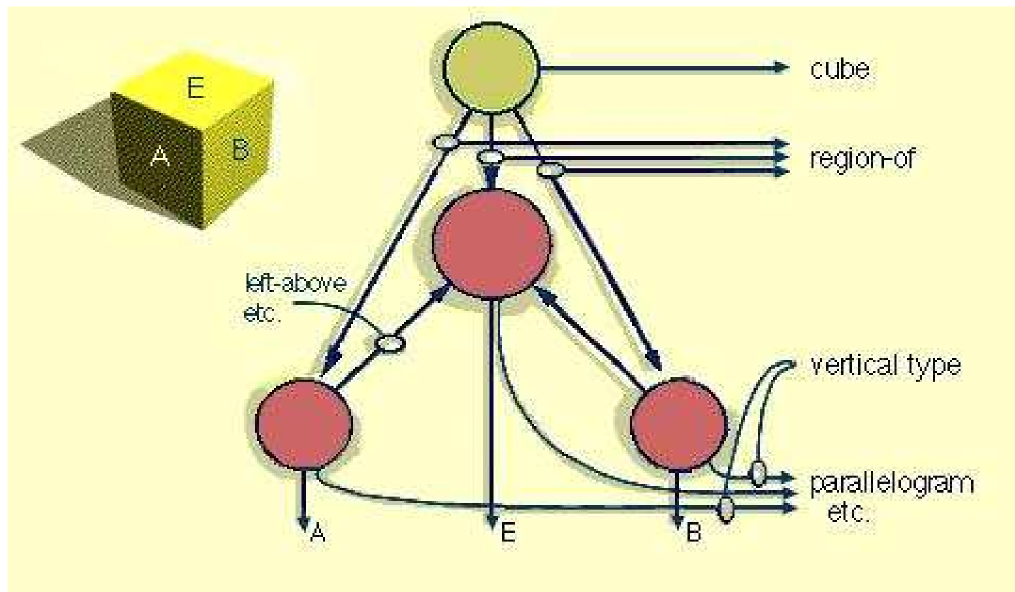

Figure 1: frame-representation of the cube-example.

In language the frame method can also be used for explaining the way we understand certain sentences. Consider the following sentence: "Jack drove from Boston to New York." To understand this sentence a "travel"-frame can be constructed with pronomes:

- origin.
- destination.
- actor.

The terminals corresponding with these pronomes are filled with:

- Boston.
- New York.
- John.

A terminal of a frame can also contain a subframe. For instance in the following sentence: *The thief who took the jewels moved them to Amsterdam.* can be viewed as a 'trans'-frame consisting of an actor-terminal, an action-terminal, an object-terminal and a destination-terminal. The actor-terminal is filled in with yet another 'trans'-frame. This frame also has an actor-terminal, an action-terminal and a object-terminal. Because there is no destination mentioned

here it is left out. The notion of a trans-frame is based on ideas of Roger Schank in the early 1970's. He developed a theory of what he calls 'conceptual dependencies'. He stated that *transportation in space, transmission of thougths* and *transfer of ownership* can be represented by the same general structure, what Minsky calls a trans-frame. According to Minsky this has the same reason as why we use the word 'trans' in each of these situations, so language already views these 'trans'-constructs as similar. When John tells his address to Mary this process can be seen as a trans-frame with John as origin, Mary as destination, tells as action and John's address as object.

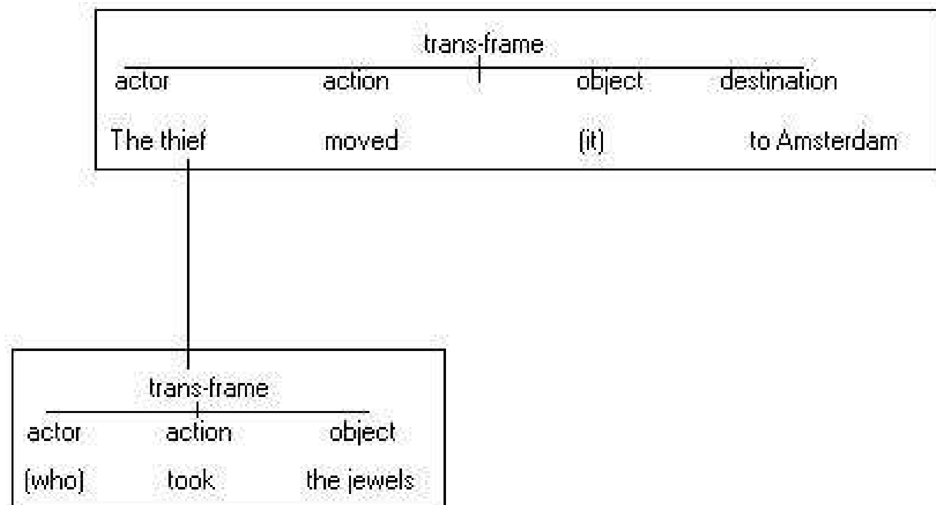

Figure 2: trans-frame system for the sentence: The thief who took the jewels moved them to Amsterdam.

This shows that the general structure of a frame-system can also be used for explaining language processing and understanding. In his book Minsky shows that his theory can be applied to many more cognitive processes, but most of them are explained in the same way as the above examples. The most important aspect of Minsky's theory for this thesis is the selection of a context with defaults that can be adapted to a new situation by filling in the details.

3.4 Arkin's scheme theory

Originally scheme-theory was developed by psychologists to explain certain workings of the human brain. It was first proposed for use within artificial intelligence by Arbib. In [7] Lyons gives a theoretical description of scheme-theory for robotic systems. Arkin describes his view on this theory and also his implementations on real robots in [8] and [9].
A scheme is defined by Arbib as a data-base for knowledge essential to the behaviour the scheme describes (as in Minsky's frames), but a scheme also includes procedural information how to use this knowledge. We speak of a scheme instance when a scheme is provided with the optimal parameters for the current task. Schemes get instantiated by perceptual information directly or by the output of other scheme instances.
Arbib distinguishes two types of schemes: perceptual- and motor-schemes. Perceptual schemes are always instantiated by sensor-information, they then provide access to the particular motor-schemes that can react to this sensor-information. Motor-schemes are descriptions of simple, basic behaviours that all serve a particular purpose. Examples of motor-schemes that Arkin uses in his AuRA, Autonomous Robot Architecture are:

- *Move-ahead* scheme for straight driving.
- *Move-to-goal* scheme defining a goal position and the procedure to drive towards it.
- *Avoid-static-obstacle* scheme for obstacle avoidance.
- *Stay-on-path* scheme defining procedure to stay on a given path defined by path-parameters.
- *Noise* A process that uses randomness to escape local minima (in simulation only, real robots have noise by default because of the noisy environment.

Each of these schemes are instantiated with the right parameters reflecting the current status of the environment. They output a single vector describing the velocity and direction that the single scheme wants the robot to move. The set of vectors that the set of schemes creates is combined by simple vector-additions and normalizations. This is basically a variation of the potential field method, the only difference is that in this case only the motion-vector is computed, not the entire field. Also because the seperate scheme-instances are relatively simple to compute the overall process is computationally

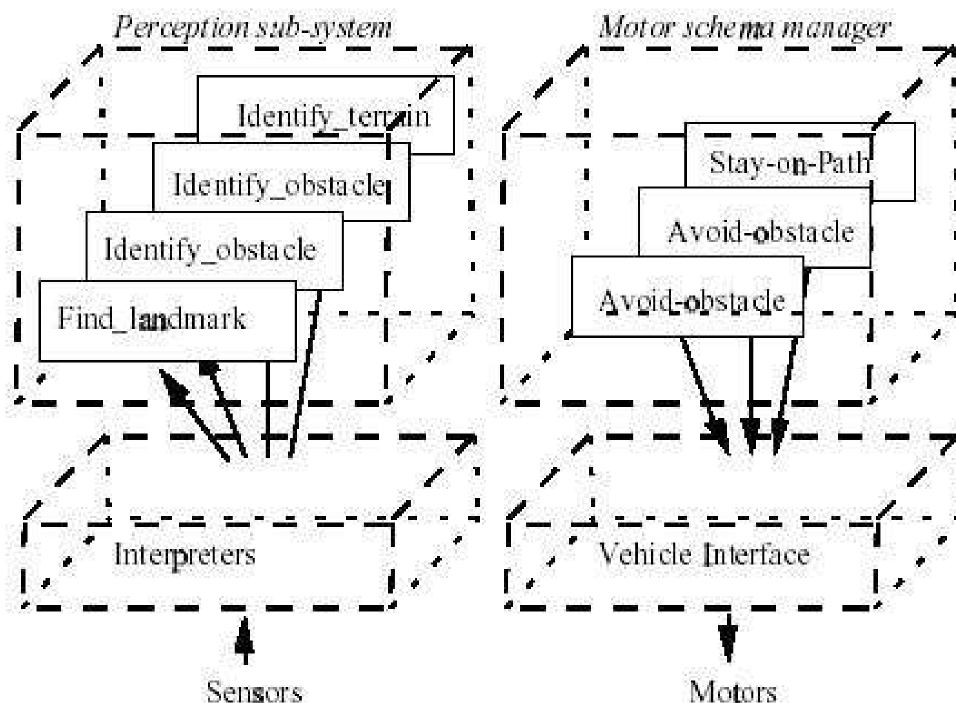

Figure 3: robot architecture using schemes.

fast and accurate. Perceptual schemes are (in the case of AuRA) based on the information particular motor-schemes need. For instance the *Avoid-static-obstacle* scheme needs some sensor-scheme to provide it with information about the static obstacle, therefore a corresponding perceptual scheme could be *ultra-sonic sensing* or *find-landmark*. A *Follow-person* scheme should need visual information and therefore something like a *Identify-person* perceptual scheme. More than one instance of a scheme can be active at the same time, for example when there are two obstacles present at the same time at different locations, two instances of the *Avoid-static-obstacle* are concurrently active, though with different parameter-values. Schemes that cooperate increase, while competing ones decrease each other's influence on the whole process. In such a system it is very rare that one scheme is controlling the complete motion. Lots of different scheme-instances representing different behaviours are combined to output the single motion-vector. An example of how to compute a vector for a given scheme is the following equa-

tion for the *avoid-obstacle* scheme.

$$O_{magnitude} = \begin{cases} 0 & \text{if } d > S \\ \frac{S-d}{S-R} * G & \text{if } R < d <= S \\ \infty & \text{if } d <= R \end{cases} \quad (1)$$

$O_{direction}$ = along a line from robot to center of obstacle moving away from obstacle. (2)

with:

- S: Sphere of influence of the obstacle.
- R: Radius of the obstacle.
- G: Gain.
- d: distance from the robot to the center of the obstacle.

When such a scheme is instantiated S or G can be variated in different situations, finding optimal instantions is one of the main issues in this thesis.
The instances of schemes representing obstacles are all kept in memory, this is the most important similarity between my platform an Arkin's work.
Arkin's AuRA is contains five basic subsystems:

- 1.perception: this is where the sensor-data is passed to the motor-schemes and to the cartographic subsystem.
- 2.cartographic: this is where both a priori and percieved world-knowledge is stored in long term memory and short term memory respectively. Also the uncertainty in the spatial knowledge is managed here.
- 3.planning: this is both the high-level planning system (hierarchical) and the distributed plan execution system. This planning system is what makes AuRA a hybrid architecture.
- 4.motor: this is the interface to the robot-control, this should be the only part that is really robot-type dependent. (Arkin himself used Denning mobile robots, first HARV and then George).
- 5. homeostatic control: this is where the robot monitors its own internal conditions, like fuel and temperature.

Arkin has also extended his system to 3-dimensional navigation by increasing the degrees of freedom for the robot to six (3 translational and 3 rotational). This has been tested on rough terrain

with the same Denning robot. Another system that uses schemes is VISIONS for interpretation of natural scenes, this involves only perceptual schemes. In Arkin's systems, not only the optimal set of parameters for the different schemes and scheme instances must be found, but also, when more versions of scheme-implementations are available, the optimal set of schemes must be found. Because these configurations differ from situation to situation, experience, i.e. old experiments with somewhat the same circumstances, is very useful.

3.5 Frames and schemes compared

Frame-theory and scheme-theory are very similar though not equivalent. What are the similarities and what are the differences?
One major similarity is that they use *input-matching routines* to determine the current state of the system (in the case of frames you can view 'system' as 'human'). This means that both have a way of matching the its input to a default value within the frame or scheme, then, based on this value, accept or reject it. However in the case of frames, when a frame is rejected another (similar) frame is selected from memory. In the case of schemes the parameter-value responsible for the failure is adapted. A scheme therefore can be a single entity, while a frame only works within a system. Arbib[13] states that a scheme is actually equivalent to a frame-system and a transformation within a frame-system is equivalent to updating the parameters of a scheme.
Another difference is the fact that frame-selection is a sequential process while the scheme-system of cooperation and competition is essentially parallel. "Is the language of frame-systems of schemes appropriate to the study 'representations of representations'? I suspect that the answer is 'Yes'".[13].

4 autonomous systems

4.1 introduction

Autonomous systems are defined as computational devices that need to operate in an environment that is similar to that in which humans must operate. Therefore they need to be able to react to unforeseen circumstances, they must be able to handle exceptions. By exceptions I mean input that is not expected,i.e. information errors (of course also hardware errors can occur but this is a different subject).
For instance a robot is meant to drive from a point A to a point B. Its memory (short term) contains descriptions of path-segments which the robot needs to drive to reach B from A, it has done it many times. But unfortunately this time an object is standing in its way. The robot detects the object using sensors (ultra-sonic, laser, camera, etc..), and plans a path around the object such that when the object is passed the normal execution can take over again. To do this it must generate new path-segments and replace these with part of the old set of segments. These new segments are of course determined by the sensor-values.

Roughly two types of autonomous robots exist:

- manipulators.
- mobile robots.

Because MARIE is a mobile robot we will be discussing the latter one. Of course there are other distinctions between robots, a mobile robot, for instance, can have legs or wheels, but it could also be that the robot is able to fly. Most autonomous robot systems are divided into three main parts:

- perception; the robot needs to be able to percieve its surroundings.
- reasoning; the robot needs to reason about what it percieves to make smart choices about what to do.
- control; the robot needs to actually undertake the actions provided by the reasoning part.

Because the actuators change the environment relative to the robot these changes need to be percieved again, thus the whole process is in fact a loop: the perception-reasoning-control loop (also often called sensor-reasoning-actuator-loop. This loop is in fact one the most

important aspects to keep in mind when designing a robot. In most mobile autonomous systems the sensor-data is used to construct a map which can then be used for reasoning, in this case probably path-planning. When a (partial) map is constructed control takes over and changes the environment, i.e. the map needs to be updated.

In the literature on autonomous systems a number of different proposals for *robot-architectures* can be found. In the next sections I will describe the three most important ones:

- functional architecture.
- behavioural architecture.
- hybrid architecture; the one that was used to design MARIE .

The last one is actually a combination of the first two as we shall see in section 3.4.

4.2 functional architectures

Within this type of architecture the system is divided into modules that each describe a basic functionality. For instance there could be a module for the perception part, one for the reasoning part and one for the control part. These architectures have as advantage that the entire system consists out of smaller modules which therefore can be kept reasonably simple. The problem with this type of architecture is that the different modules cannot be tested separately because they depend on each other for input, when an interface between two modules is changed at least these two modules need to be updated. In the worst case this need to change modules propagates through the entire system. Furthermore when a module fails the entire system will fail because the data is sequentially propagated through the modules. Thus a system build up in this way is a sequential system. Also because all modules are necessarily involved in the execution process these systems tend to be slow, which is a major drawback for a real-time system. This type of architecture is also known as *hierarchical.* An example of a functional architecture is the twin-hierarchy designed by Crowley for his surveillance robot. See figure 4.
The term *functional architecture* is also often used to describe one step in the design process of an autonomous system. The design of such a system is divided into three steps:

- functional architecture design.

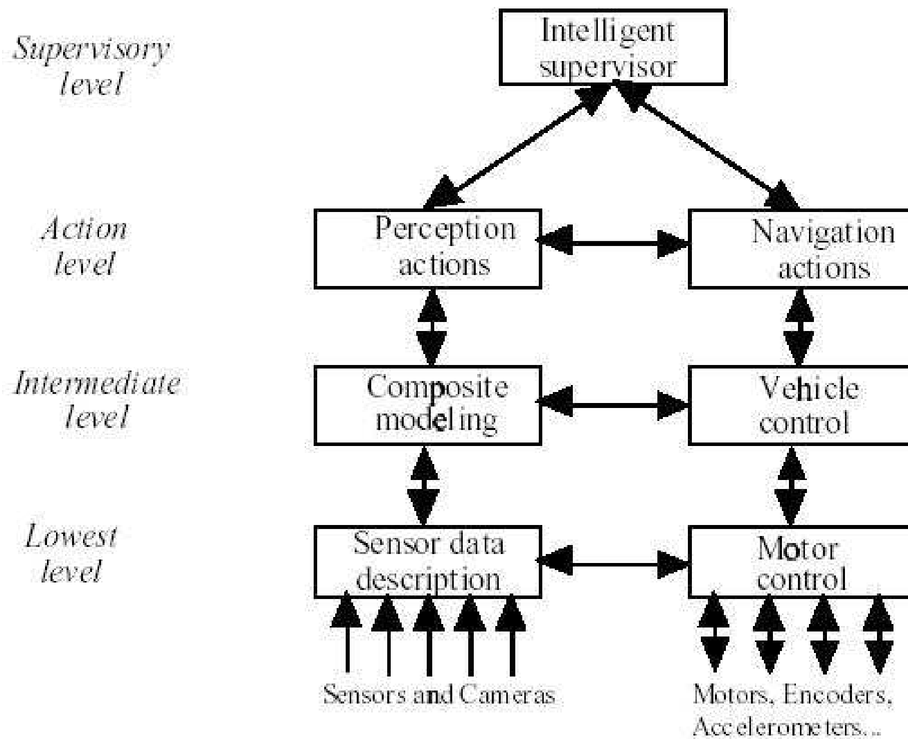

Figure 4: twin-hierarchy by Crowley, an example of a functional architecture.

- operational architecture design.
- implementation architecture design.

In the first stage decisions have to be made about the functionalities the system needs and how these functions have to be interconnected. The second stage is mainly concerned with operations and the environmental constraints to these operations. It deals with the ordering in time of the operations and the data-flow between the different operational levels.
In the last step the choices of programming language, operating system, etc.. are made. Also at this stage the designer decides where and how data is stored (central storage, each module its own datamanager, etc..).
This thesis is mainly about the implementational level, the module plays a central role and its design is part of implementational design step.
Because of these two ways to use the term *functional architecture*, the hierarchical architecture is often called *functional decomposition*.

4.3 behavioural architectures

An other way of designing a robot is to define different behaviours that can run in parallel. A behaviour contains both perception and control, a reasoning part is omitted. Examples of behaviours are:

- follow a wall
- collision avoidance.
- move towards goal.

In the early 1960's ai-researchers began to understand the benefits of distributed artificial intelligence (DAI). The fact that multiple, simple, agents can, when interconnected in some purposeful way, generate intelligent behaviour.
The advantage of this architecture as opposed to the functional one is that you can easily add and remove modules (behaviours) and each module can be tested seperately. Because the behaviours are independent of each other one or more behaviours can fail without making the entire system fail, this makes the system more robust. We call this *gracefull degradation*. A drawback is however that it is difficult to assign new tasks to the system without adding new behaviours that are needed for achieving the new goal. Adding new behaviours can result in the need to completely rewrite the procedure that selects behaviours needed for a certain task.
Because there is no reasoning part, the intelligent behaviour emerges from the cooperation between the modules. When for instance the *move towards goal* behaviour is controlling the system at a certain moment and an object is percieved the *collision avoidance* behaviour starts influencing the control by modifying the path. It is still moving towards the goal but avoids obstacles on the way: the *move towards goal* and the *collision avoidance* behaviours are cooperating. This type of architecture is essentially parallel in nature and therefore can respond more reactively in real-time. Systems like these are closely related to DAI, behaviours can be seen as individual agents.

An example of a behavioural decomposition is the subsumption architecture by Brooks. In this system each high-level behaviour is decomposed out of more lower-level behaviours. The behaviours all have a certain priority at a certain moment in time, the behaviour that has the highest priority has the highest influence on the control of the actuators. Also Arkin's AuRA, described in section 2.4, is an example of a behaviour based system.

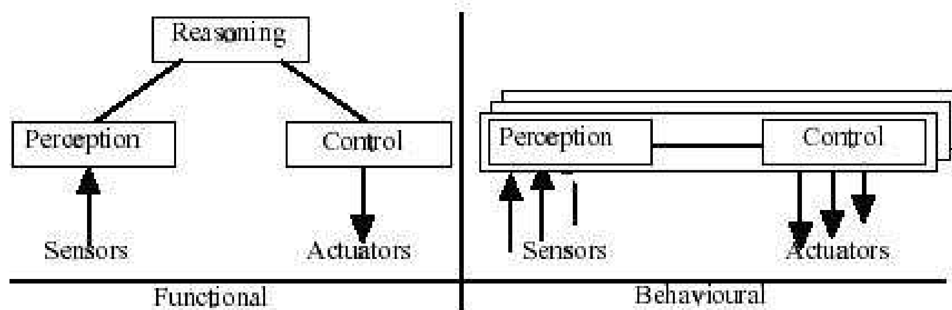

Figure 5: functional vs behavioural architecture.

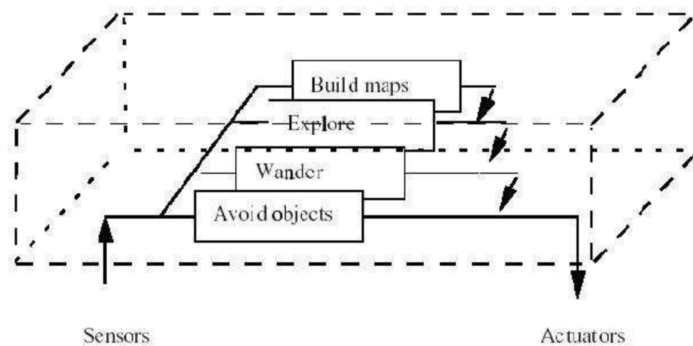

Figure 6: subsumption architecture by Brooks.

4.4 hybrid architectures

As I said earlier hybrid architectures are actually a combination of the ones described above. There still are behaviours that run in parallel but there also is a reasoning part that decides which behaviours to run and maybe which parameter-values to use. This type of system has the advantages of both the types of architectures mentioned above. When the high-level planner is generic enough it can handle new behaviours more easily than behavioural systems can. In most systems using a hybrid architecture the lower levels are more behavioural while the higher ones are more functional.
MARIE is designed according to a hybrid architecture. MARIE has as its highest level the online task-planner. Then it has a series of high-level modules such as the pathplanner that use symbolic representations. At the lowest level there are behaviours like *collision avoidance*. MARIE uses a central database for storing information, this database is used by all the modules. This makes the parts

of the system less dependent on each other. The connection between the high-level symbolic part and the low-level numerical part is established by the execution control level which uses the *action dispatcher* to create a number of virtual robot instructions that can be passed to the lower levels for execution. The action dispatcher needs a task-tree as input which will be described later on. The execution control level still is sequential but provides an interface to the distributed part.

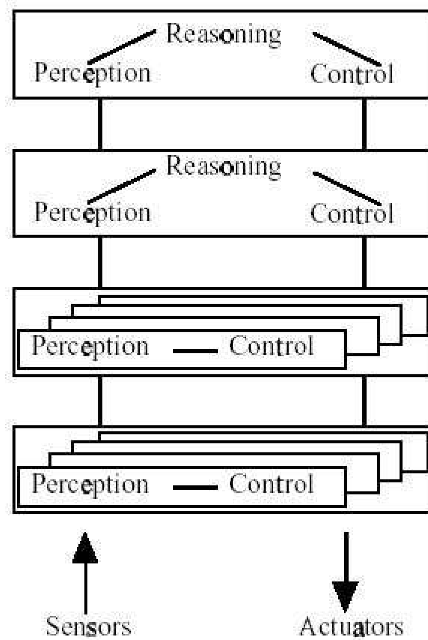

Figure 7: a general overview of a hybrid architecture.

Other robots-systems based on hybrid arhitectures are implemented by Payton.

5 MARIE

In this section I will describe MARIE. First I will briefly go over the hardware the robot consists of. Then I describe in more detail the implementation-architecture of MARIE. At the end of this section a few experiments done with MARIE are described.

5.1 hardware

MARIE is equipped with three batteries, two for the motors and one for the computer and other electronics. There are two types of ultrasonic sensors that the system uses, Robosoft and Polariod. The robosoft sensors are wide-beam sensors used for collision avoidance, the polariods are narrow-beam sensors for recognizing features in more detail. The wall-sensor uses these for constructing walls for example. MARIE has a VME-crate computer with a Motorola 68020 processor and 4 MB of local memory. MARIE uses a Breezenet wireless network to communicate with the UNIX-workstations of our network. The wheels are equipped with relative encoders for localization by *dead reckoning*. Bumper-sensors are used to prevent the system from bumping into objects, they immediately turn off the system when that happens. This can also be done manually by pressing a stop-button on the robot

5.2 software

The operational architecture for MARIE, besides the user and the robot itself, consists out of four levels. The plan-generator uses user-input to generate a task-tree that serves as input to the action dispatcher. The output of the action dispatcher are virtual robot instructions that at the virtual robot level are translated into virtual hardware instructions, which, through an interface, are translated into electronic signals. See figure 7. This is similar to the levels of abstraction a normal computer uses in the sense that data understandable by humans is translated through several layers into data the computer-hardware can understand.

The implentation architecture for MARIE is most relevant for our discussion because at this stage the modules play an important role. In figure 8 the modules are visualized by a rectangle with rounded corners.

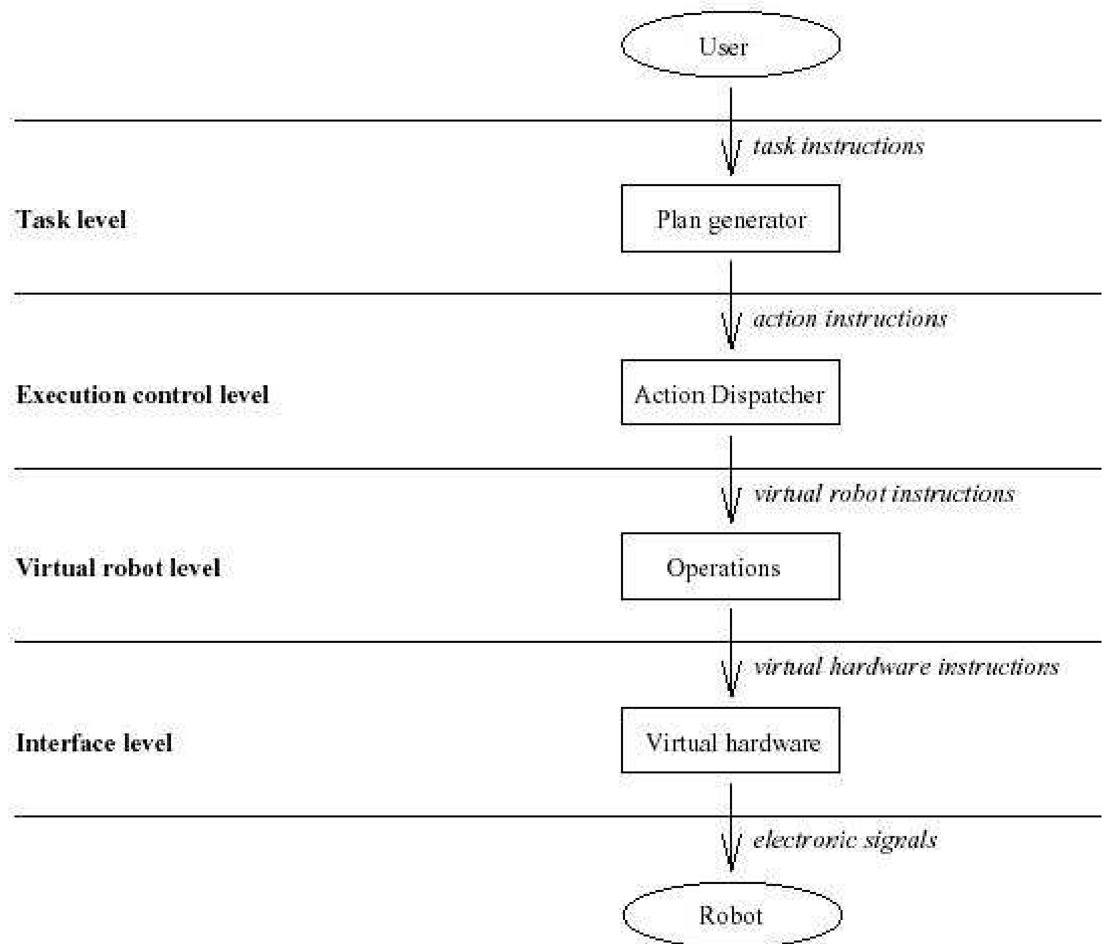

Figure 8: operational architecture for MARIE.

As I said earlier, the architecture of MARIE is hybrid. The task-planner is a sequential module which can be viewed as the reasoning part of the architecture. At the lowest level there are the operations of which the elementary operations run in parallel. Examples of these elementary operations are the logical sensors like the wall-sensor.

Data-handling is done through so called datamanagers. These are interfaces for storing and managing data of different kinds.
Similar data must be easily accesible even if they are the result of different sources. There are actually three different data-managers MARIE currently uses:

- World data-manager : data-manager for storing and accessing

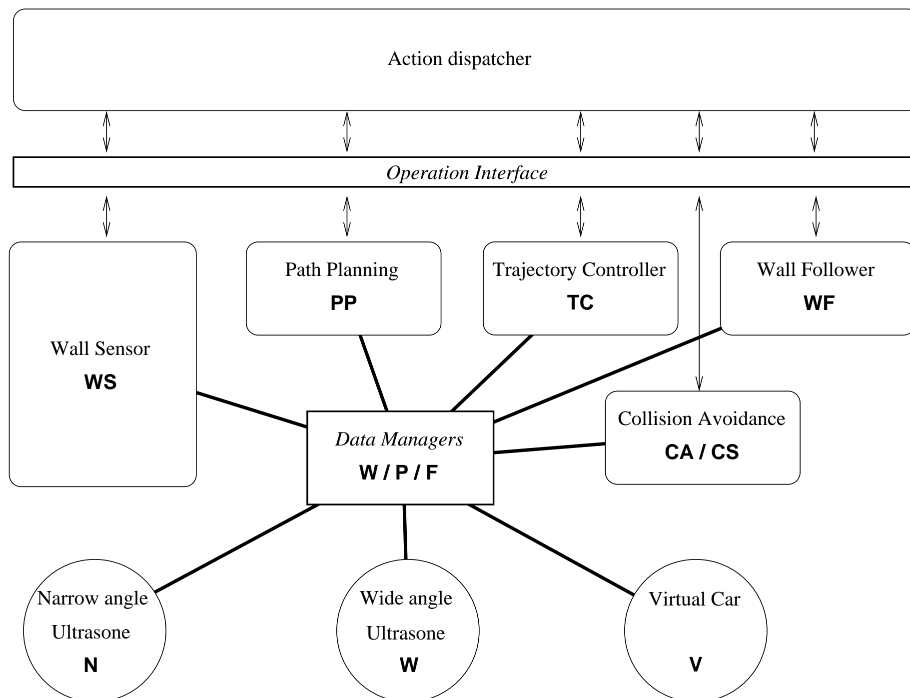

Figure 9: implementational architecture for MARIE, shows the module and their interconnection.

world-features provided by the human-operator at the start of the mission or by processed sensor-data (i.e. the output of a logical sensor).

- Feature data-manager : data-manager for storing and accessing features constructed by raw or processed sensor-output.
- Parameter data-manager : data-manager for storing and accessing parameter values that are set by the operator or that are the result of parameter planning algorithms (path-planning).

So for instance the path-planning module gets its input from the world-datamanager and outputs its results to the parameter-datamanager which in fact serves as the input-datamanager for the trajectory controller.

In the next section I describe the wall-sensor in some detail. In section 6 I describe the other elementary operations although in less detail than the wall-sensor. But first some experiments done with MARIE.

5.3 some experiments

In the past a lot of experiments have been done. For testing the online-planner [6] the robot had to drive a trajectory in which at one moment MARIE had to choose between two options. The wall-sensor was used here to find a wall which the wall-follower can follow. The trajectory controller was used to take corners. For my experiments it was enough to just let the robot follow a wall for a while each time with different parameters for the wall-sensor (actually just one as we shall see later). I did the same experiment for testing the corridor-sensor (see section 5.7). Also a docking-demo was done [3] in which the robot needed to find a docking spot and park itself. For this experiment a specific module *docking-spot detector* was designed. I will get back to experiments in section

6 the wall-sensor

6.1 introduction

The wall-sensor within MARIE is a virtual sensor that is used to search for or follow a wall. MARIE uses polariod ultra-sonic sensors for this purpose. These sensors have a narrow beam width that makes them more accurate in finding wall-like obstacles. The wall-sensor has one control parameter *the search-angle* that determines the direction in which the sensors will search for a wall. It constructs a map of the environment and keeps updating it, deleting old data.

6.2 the model of the environment

The environment for the robot is represented by a 2-dimensional map in which straight line segments represent obstacles, in our case only walls are interesting so we will stick to walls. Measurements are represented by points, when more points lie more or less on a straight line this is considered a wall (or wall-like obstacle). A line is defined by the following parameters:

- x_l, y_l :coordinates of the point on the line of which the uncertainty is minimal; the average of all measurement-points belonging to that line.
- θ_l :orientation of the line normal.
- $\sigma^2_{xy_l}$:the variance of x_l and y_l.
- $\sigma^2_{\theta_l}$:the variance of θ_l.
- l_r, l_l :the distance between (x_l, y_l) and the ultimate right end and left end of the line respectively.

Points have the following ingredients:

- x_p and y_p : the coordinates of the point(before bundle-correction is applied).
- θ_s :the orientation of the sensor on MARIE and therefore?? the angle between line through origin and point and the x-axis.
- ψ : half the beamwidth of the sensor.
- d :the range measurement.
- σ_{xy} : the variance in the coordinates.
- id : id-number (int) of the point.

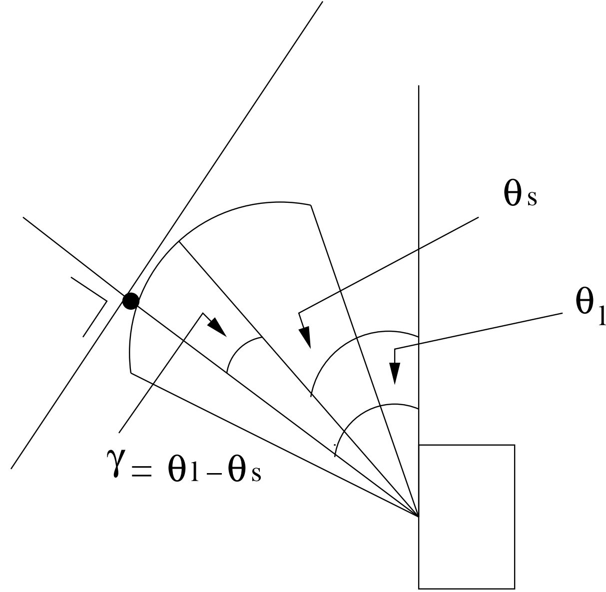

Figure 10: ultra-sonic cone with defined angles.

A robot-centered as well as a world-coordinate system is used. The first is used for the detection of objects (such as walls), the latter for self-localization by dead reckoning. Two things need to be done during execution of the wallsensor:

1. when the robot drives the new parameters of the wall within the robot-centered coordinate system must be determined. $\Delta x, \Delta y, \Delta\alpha$ are the x-translation, y-translation and rotation of MARIE respectively. The new parameters now become:

$$\begin{aligned} x_l &= x_l cos(-\Delta\alpha) - y_l sin(-\Delta\alpha) - \Delta x, &(3)\\ y_l &= y_l sin(-\Delta\alpha) + y_l cos(-\Delta\alpha) - \Delta y, &(4)\\ \theta_l &= \theta_l - \Delta\alpha. &(5) \end{aligned}$$

2. new measurements must be used to update the model.If (x_m, y_m) is the position of the measurement-point, then the new surface-normal becomes:

$$\theta_m = \begin{cases} \beta_m + \frac{\pi}{2} & \text{if } 0 < \angle(\beta_m, \theta_s) < \pi \\ \beta_m - \frac{\pi}{2} & \text{otherwise} \end{cases} \qquad (6)$$

with

$$\beta_m = arctan\frac{y_m - y_l}{x_m - x_l} \quad (7)$$

6.3 handling uncertainty in the sensor-values

Because of the uncertainty in the sensor-values we need a Kalman filter to estimate the parameters accurately. The Kalman filter needs $\sigma^2_{xy_l}$ and $\sigma^2_{\theta_l}$ to determine the Kalman gains for position and orientation.

$$K_{xy} = \frac{\sigma^2_{xy_l}}{\sigma^2_{xy_l} + \sigma^2_{xy_m}} \quad (8)$$

$$K_\theta = \frac{\sigma^2_{\theta_l}}{\sigma^2_{\theta_l} + \sigma^2_{\theta_m}} \quad (9)$$

The estimation of the line parameters now becomes:

$$x_l = x_l + K_{xy}(x_m - x_l) \quad (10)$$

$$y_l = y_l + K_{xy}(y_m - y_l) \quad (11)$$

$$\theta_l = \theta_l + K_\theta(\theta_m - \theta_l) \quad (12)$$

Before this Kalman filter [3] can be applied $\sigma^2_{xy_l}$ and $\sigma^2_{\theta_l}$ must be calculated:

$$\sigma^2_{xy_l} = \sigma^2_{xy_l} + \sigma^2_{\delta xy} + max(x_l^2, y_l^2)\sigma^2_{\delta\alpha} \quad (13)$$

$$\sigma^2_{\theta_l} = \sigma^2_{\theta_l} + \sigma^2_{\delta\alpha} \quad (14)$$

The values $\sigma^2_{\Delta xy}$ and $\sigma^2_{\Delta\alpha}$ are dependent on the accuracy of the (relative) shaft-encoder. To minimize the variance in the parameters of the line bundle-correction [1] is applied.

6.4 new line-segments

When a new measurement point presents itself and if it is part of a linear subset of previous points that contain enough points and that shows no large gaps a new line segment can be estimated. x_l now becomes the new average of x_p's, y_l the new average of y_p's and θ_l is calculated using equation (6) and (7).

6.5 implementation details

The implementation of the wall-sensor is done in the C programming language, I will not go into detail about the all the support-functions but I will briefly describe the working of the main routine. The main-source for the wall-sensor can be found in the file *ws-task.c* The main routine begins with initializing a lot of parameters which I will describe when they are needed. Next it sets up a connection with an eo-client and the virtual cart. Then this module checks in which mode the program currently is. There are two essential modes (there are more modes but they are not important for our discussion here): WS-SEARCH-MODE and WS-FOLLOW-MODE. If the program is currently in search-mode than it calls the function 'find-wall'. If a wall is found and it is within a certain range of the robot than the wall parameters are stored in a data-manager and a call to 'eo-finish' using the flag WS-FOUND-WALL is used to terminate this operation. If the program is currently in follow-mode than, after the call to 'find-wall' only the updated parameters of the wall are stored if a wall is still found. If in this case no wall is found (so 'find-wall' returns FAIL) 'eo-finish' is called using the parameter WS-LOST-WALL.

6.6 wall-searching

The sonars on the robot scan from left to rigth with a firing rate of 1Hz [1] so there are 8 measurements per second. A wall is represented by a struct 'ws-wall-rel',this struct is defined in the file ws-sensor.h. The struct contains 5 doubles and a line (which is another struct), a line also has an id (int) by which it is recognizable. The 5 doubles are:

- r : distance of cart to line.
- θ : angle of line relative to cart.
- x, y, ϕ : pose of cart at the time of measurement.

The function 'find-wall(double gamma-low, gamma-high, gamma-step, int wall-id)' returns the id of a line that has an orientation between gamma-low and gamma-high. The set of lines is searched in steps of gamma-step. It used to return the first wall it found, but I have rewritten this such that it returns the line that is closest to: $\theta = \frac{\gamma_{high} - \gamma_{low}}{2}$ When a line is found a call to 'get-wall-position(int wall-id, double *r, double *theta)' gives the distance and orientation of the wall relative to the cart. The search-angle, the parameter of the wall-sensor is actually this θ.

6.7 corridor-searching and following

In order to be able to test the platform described in this thesis an second, different version of the wall-sensor is very useful. Therefore I tried to implement a corridor-sensor, i.e. an extension to the wall-sensor that is able to find corridors as well as walls. To extend the wall-sensor to a *corridor-sensor* the scanning process need to be divided into two parts: one that searches for a wall on the left side and another that searches for a wall on the right side. However there is obviously no need for defining a second search-angle, we can just use $+\theta$ and $-\theta$. Corridors can be defined by a struct 'ws-corr-rel' but I chose to simply define a corridor by defining two walls with a degree of confidence that they in fact are parallel to each other. Of course new modes for the wall-sensor need to be defined: WS-SEARCH-CORR-MODE and WS-FOLLOW-CORR-MODE. Because WS-FOLLOW-MODE is in fact equal to WS-SEARCH-CORR-MODE this latter one is not necessary. They are equal because when you are following a wall but are currently not following a corridor (this is what WS-FOLLOW-MODE means) a corridor must be found next. When no corridor can be found the robot keeps following the single wall. The expectation is of course that this version of the wall-sensor works better than the old one when MARIE can follow a corridor. If no corridors are present it should work as well as the old one. I tested it and it still was able to find a wall but due to some unknown problems it never found a corridor. In the section on results I will argue about the reason for this failure.

7 the task-planner and the action dispatcher

7.1 action dispatcher

The action dispatcher is in fact a somewhat different module in the sense that it is always used. The action dispatcher recieves a tasktree (see next section) and parses it, next it activate actions and passes parameters to the elementary operations out of which the action consists. Then it interprets the return values of the elementary operations to get a description of the state of the system at that point in time and it moves on with parsing the tree using the newly acquired state. When all leaves of the tree are done the system stops when the goal-state is reached, else it re-parameterizes the operations.

7.2 task-planner and scenarios

MARIE makes use of a high-level on-line task-planner [6]. When this task-planner is called it queries the user for the nessecary information, like local and global flags that need to be on or off and the parameters for the elementary operations. The flags are used to keep track of the internal state of the system. The task-planner creates a *task-tree* which it sends to the action dispatcher for execution. The planner first needs a list of flag and operation definitions, this can be found in the file *TP-TABLE.TXT*. The task-tree can contain five different types of nodes:

- AND-nodes.
- OR-nodes.
- Action-nodes.
- Simple operation nodes.
- Elementary operation nodes.

The root of the tree always is an AND-node. Obviously the AND-node represents tasks that all need to be done, while OR-nodes represent alternative branches. The action-node only has elementary operations as children, these elementary operations are executed in parallel. The AND, OR and Action nodes all have initial, during and final conditions. Children of an AND-node are all executed except when after the execution of an earlier child the final conditions of the AND-node are already met. The plan is executed successful if the final conditions of the root-node are met. The elementary operations are the ones that you can view as having scheme properties,

they must be executed in parallel, they all must recieve parameter values to instantiate them. The simple operations are executed sequentially and describe simple actions like loading the home or goal positions, throw away unnessecary data, etc.. The simple and elementary operations are the ones that actually change the internal state of the robot. The internal status of the plan-execution is monitored by flags. There are global flags that are used in the entire plan and local flags that are only used by the part that is currently executed. Because a lot of questions are asked at this state it is very error-sensitive and time consuming to do this each time you do an experiment. Therefore the task-planner can read in scenario files that contain all this information. Scenarios can be nested, this makes it possible for the user to distinguish between the different subtasks that need to be performed.

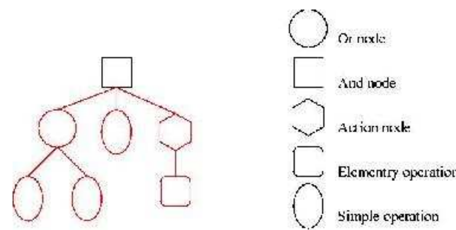

Figure 11: examples of a task-tree.

In the next section I will briefly describe the elementary and simple operations.

8 other modules and their parameters.

The modules the robot works with are divided into three classes:

- perception modules.
- reasoning modules.
- control modules.

In the next sections I will briefly describe all modules and their parameters.

8.1 wall-sensor and wall-follower.

The only perception module is the wall-sensor, because it only looks for a wall and publishes its representation to a datamanager. The wall-follower, which is a sensor-based control module, can use this information to follow this wall by keeping its parameters up to date i.e. keep on sensing, constructing the wall in memory and throwing old values away. The three parameters this module needs are:

- gamma: the search-angle.
- dT: the scanning step (see section 5)..
- mode: integer value representing the mode which the module uses. These can be :
  - SEARCH : it must search for a wall.
  - FOLLOW : when found it must follow a wall, this mode is most of the time combined with SEARCH.
  - EXPLORE : a experimental mode in which the system explores its surroundings and constructs a map.
  - DUMP-MANY : publish a lot of extra information, like points, lines, etc..

Because the search-angle is probably the most significant parameter, when trying to optimize the workings of the wall-sensor we focus on optimizing this parameter for a given task. You can see that gamma is actually the most significant by varying first only this gamma and then only dT and find that in the first case the succes-rate (fitness) changes more drastically than in the second case. I checked it for my experiment, for different experiments this dT migth be important. So when an experimenter wants to optimize this parameter he can easily adapt the fitness function described in section 8.3 to one in which dT becomes the most important factor.

8.2 path-planning and trajectory controller

The path-planner is a reasoning module, this module plans a path that later can be used by the trajectory controller to actually drive this path. This of course makes the trajectory controller a control module. The parameters used by the path-planner are:

- map-readDM : the datamanager in which a map is stored.
- map-class : type of map.
- map-source : who produced the map, i.e. sonar or human.
- plan-type : there are three types of path-planning :
  - global path planning : using a set of via-points one or more curves are generated defining a path from start to end-point.
  - docking path-planning : using only a begin- and end-point a path is generated from start into a docking spot.
  - via point-planning : only used in combination with the fuzzy controller. Only the via point planning part of the path-planner is used.

8.3 collision avoidance

The collision avoidance module (and the full stop) is in fact a submodule of the virtual car that is used for the trajectory controller, when collision avoidance is active the controller sends its parameters to this module which adapts them to the current situation. For collision avoidance three modes are defined:

- none : used when it is nessecary that the robot comes close to objects, finding a docking spot for instance.
- controlled stop : the vehicle stops when an object is detected which is too close. The parameters for the trajectory controller are left unchanged.
- obstacle evasion : the parameters for the trajectory controller can be changed (speed and the description of the path-segments i.e. curvature). The robot will not back up, when this seems nessecary the robot will come to a full stop.
- full collision avoidance : the parameters for the trajectory controller can be changed and the robot is allowed to back up when nessecary.

8.4 sensor-based controller

When the environment is only partially known direct sensor-feedback can be used for driving. At this moment only wall-following is provided for this module. The module directly recieves sensor-information representing lines. The following parameters are defined for the sensor-based controller:

- class datamanager from which to read.
- source datamanager to which path-specification must be written.

Also a fuzzy controller based on fuzzy logic has been implemented using the same kind of parameters.

8.5 docking-spot detector

The docking-spot detector is a sensor-based module that is able to detect a docking-spot. This can be used in the parking and docking application. Its parameters are:

- spot-length
- spot-depth

These parameters speak for themselves. This module constructs a docking spot from sensor-data by finding five lines making a docking spot (rear, sides, walls next to the sides).

8.6 simple operations

There are some operations that do not run concurrently like the elementary operations. These are operations for loading home and goal positions for instance, or for purging unnessecary data. These operations are called simple operations. In the task-tree they are, unlike elementary operations, not children of an action node.

9 experimentation

As mentioned before a lot of experiments with MARIE have been done in the past. But what exactly is an experiment with MARIE? To make MARIE achieve a given goal one can design a scenario structure as described in the previous section. This structure is parsed into a task-tree and fed to the action dispatcher. It tells the action dispatcher what to do and in what order. Some operations are elementary (run in parallel) some are simple operations (run sequential). When modules run in parallel on UNIX they are in the form of POSIX-threads. On VxWorks these are called tasks. As described in [6] some macro's for constructing these scenario-structures are defined. These macro's define certain sub-tasks that can be used within a full task. Because there are macro's for activating, for example, the wall-sensor these can be used for the evaluation of a module within an experiment.
The macro describes how the module must behave (which mode, parameters, etc..). So these macro's can be used as the specification of the task that a given module needs to perform and therefore can be part of the experiment-entry in the database. Because I focus on the wall-sensor only, this is not really necessary.

10 experimentation platform

10.1 introduction

In this section I will describe the theoretical and implementational details of the experimentation platform. Which is actually my own contribution to the software for MARIE. This is a method for finding optimal instantions with therefore optimal parameter-values for the different modules. Also it will make it possible for a user to select software versions and parameter settings for new experiments based on old ones. This is very similar to frame-theory in the sense that it also selects existing structures from memory that currently are the best available and adapts them to the current situation. I have implemented it only for the wall-sensor. The wall-sensor/follower is actually a combination of a perceptual and a motor scheme-like module. It not only searches for a wall and is able to recognize one, but it also is used for following walls. The wall-sensor actually has only one significant parameter: *search-angle*. It represents the direction in which the wall-sensor searches for walls. To find an optimal value for this parameter a lot of experiments need to be done, where different values are used. You also need a measure of success in order to be able to sort the experiments according to their success-rate. The experiments exist in the form of (wall-sensor) log-files, these are often very large and contain a lot of unnessecary information. Therefore I made a script using the 'awk-language' that filters the usefull information from a log-file. I have also made an awk-script that can retrieve the parameter value used from the task-planner log-file.
Many methods for optimizing control parameters can be found in the literature on robotics, one that is often used is a genetic algorithm. Genetic algorithms are based on evolution and natural selection to find a set of optimal values for a given problem. Because it uses randomness it can easily escape local minima and can handle NP-complete problems. The reason why I didn't choose for these sorts of methods is not difficult to explain. I would have to do new experiments for each generation involved, and with a population size of for instance twenty, and an evolution of only fifty generations, a thousand experiments must be done to measure the successes. This can be done when you work with a simulator but not with a real robot.
Of course there are already a lot of experiments done with MARIE and I could simply use the logfiles they produced. However unfortunately in most of these experiments the parameters of the modules

(in my case the wall-sensor) are instantiated with the same value, so no efficient parameter optimization is possible.
For the frame-like system I've constructed a database in which each entry can be seen as a frame, because each entry represents a experiment done with the robot. In the next section I will describe my design considerations. Then I will go over some widely used optimization and problem-solving methods. Next I will explain what I mean with a fitness-function and give an example for the wall-sensor. The rest of the thesis is about the experimentation platform and its necessary ingredients.

10.2 design

When designing a platform like this for a mobile robot we need to look back at the goals that we set at the beginning. There were two questions on which we will focus. These were:

- is it really possible to gain more insight in the experiments done with MARIE using the described experimentation platform?
- has this sort of long-term memory effect on the success-rate of experiments with MARIE?

In order to answer the first question we need:

- a database to relate the different ingredients of an experiment and to detect relations beween experiments.
- tools to fill and query the database. To answer the second question we need in addition an evaluation procedure for modules to gain a measure of success for the module within a given experiment.

In the next sections these elements are described.

10.2.1 parameter optimization

A major topic within robotics, especially behaviour-based robotics, is parameter optimization. The main issue here is to find an optimal set of parameters for the modules (or schemes) for the current situation. Unfortunately most of the research in this field was done using simulations of robots and environments. For real-time systems there are often different optimal-values for the same parameter in different situations, which makes it hard (maybe even impossible) to find the optimal values all the time. There are several methods known for optimizing parameters, I will briefly mention three of them:

- hill climbing : as long as we're becoming more successfull keep adding a small value to the parameter-value. This is one of the simplest ways of finding maximums but is unable to escape local maxima.
- neural networks : the state of a network of interconnected neurons is updated according to some rule based on actual- and desired states. This evolves to a state that can represent a solution to a given problem. This is especially used in statistical pattern recognition but also for example in speech-recognition.
- genetic algorithms : controlled random search based on natural selection. Because this method is random to a degree it is able to solve some problems quickly without getting stuck at local maxima.

Hill climbing has as the disadvantage that it gets stuck in local maxima, it is possible that the evaluation goes down at one moment in time while further on in the search-space it gets higher than it ever did. Neural networks are not really suitable here for two reasons: there is not really one function to be found because of dynamic nature of the environment. Also you need many learning-examples to adequatly approximize a function. For genetic algorithms, as I said earlier, the same argument holds, you need to do too many experiments before you can say anything useful about the result of the algorithm. Full optimization is not possible for real-time robot parameters because the environment will never be exactly the same twice.

10.2.2 fitness-function; measure for success

In this section I will try to give an idea of a fitness-function for a MARIE module. Because I focused my work on the wall-sensor, I describe a possible fitness-function for this module. To measure the success-rate I used the amount of times a wall is found and lost again. Because the system can reinitialize during the run, the amount of times this happens is also recorded. Beforehand I must say that this function is far from perfect, it should only give insight in how to think about evaluating behaviour-modules. The fitness function that I use (the name fitness-function I borrowed from genetic algorithms) is the following:

$$fitness = \begin{cases} \frac{x}{2x+1} & \text{if L + I = F} \\ \frac{x+1}{2x+1} & \text{otherwise} \end{cases} \tag{15}$$

with:

- F: the amount of times a wall is found within the experiment.
- L: the amount of times a wall is lost within the experiment.
- I: the total amount of times the system is reinitialized.
- x: F + L + I.

In this way the fitness will be between 0 and $\frac{1}{2}$ if the amount of times the wall is found is less than the sum of L and I. The fitness will be between $\frac{1}{2}$ and 1 if F is equal to the sum of L and I. This latter situation is always better than the first because at least at the end of an experiment a wall is actually found and not lost again. However the fitness-function for this situation is decreasing as x increases. This is because the faster the final wall is found the better. While in the situation in which L + I is equal to F no wall is found in the end so to keep trying is good and therefore when x increases the fitness also increases. All this keeps the fitness-values between 0 and 1, which is important later on at the selection process (this of course depends on the selection procedure).

When a lot of experiments are done you can closely determine a

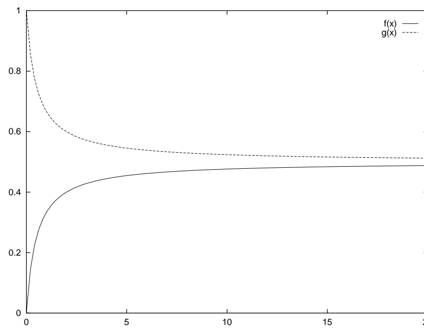

Figure 12: fitness function.

description of the function that maps parameter-values to fitness-values, lets call this function EP. When a new experiment is done the log-file will be placed in the data-base and EP can be updated. When in future experiments only the best parameter according to the current EP is picked the learning process stops and this is of course not desirable. The selection procedure therefore needs to not only select old values using EP but also to generate new values.
A good possible addition to this function would be include the amount of measurements done before a wall can be constructed. Each time a new measurement is done and added to the model this is mentioned in the log-file. Of course the least measurements it

needs the better. In theory a line (and therefore a wall) can be represented by two points, because measurements in the model are represented by points the least amount of measurements to construct a wall would be two. Because this would yield too many false walls, the minimum amount of measurements is actually five. Also the amount of time the wall-sensor is searching can be taken into account when optimizing this function. Because this fitness function

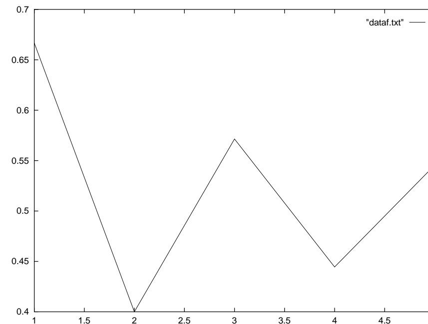

Figure 13: fitness values during wall-searching experiment.

always converges to 0.5 for large x this function is only suitable in short wall-searching experiments such as mine. On the other hand when it takes a large x to follow a wall for a short time 0.5 migth in fact be a suitable evaluation-value, i.e. it works but is far from optimal.

10.2.3 selection

For selecting already used parameter-values there is nothing wrong with using the standard roulette-wheel selection. This method I also borrowed from genetic algorithms, it is a method in which the best parameter value gets highest probabillity that it is selected. The algorithm generates a random number between 0 and 1, next the fitness-values are added and normalized such that the sum of all fitness-values becomes 1. So each fitness-value gets its own space within the interval [0,1]. The value in which space the random number falls is selected. Because the best values takes up the most space within the interval it has the highest probabillity of being selected. Of course there are several other methods for selecting parameters but I will not decribe them because they are unnessecary complex for our discussion.

As I said earlier EP must not only select old values but also has to generate new ones to keep the learning process alive. Of course

the learning process would stay alive if you simply generate new parameter-values randomly, but smart choices might speed up the process considerably. You can, for instance, use a heuristic in the selection procedure. Once a suitable experiment is selected from the database you can use the same parameter-value and update the fitness by taking the average of the two. However you can also change the value of the parameter slightly to find a new parameter-fitness pair which can be used to update the database. When a new experiment that uses the wall-sensor will take place, the user can first select all previous experiments where the wall-sensor was used. Next the one with the highest fitness for the ws-module can be selected. By now a single 'frame' is selected that can be adapted to the new situation by online-parameter tuning. This is basically what happens in Arkin's scheme-method. However, we can ask ourselves the question whether or not precise selection is really useful, given the fact that full optimization is not possible. I would say that our database is more appropriate for giving extra insight into what happens internally during an experiment, rather than selecting optimal configurations.

10.3 implementation architecture

In this section I will give an overview of the different important elements of the platform. Also I will describe the database that we use. In the next section I will go over the implementation details. But first some definition of terms that I will use often.

10.3.1 explanation of terms

There are some different kinds of files that I will be discussing later. When I talk about sourcefiles I mean regular sourcefiles usually written in C (so with the extension ".c"). These can be linked and compiled to (at least) two different executable-types. Executables for VxWorks (".vx") and UNIX-executables. One the task for my platform is to distinguish these two. Logfiles are the files generated during an experiment. The experimenter can log the system by using the UNIX-command "script". He can switch certain modules on or off by switching there printstatements on or off (I will get to printstatements in the next section). That's where we get to the logparts. I define a log-part as the logfiles consisting only those lines (printstatements) which belong to a single module. In my case only

the wall-sensor is interesting so most of the experiments I did only monitored the wall-sensor. Often an experiment only generates log-parts because modules run in different shells or the experimenter (as in my case) is only interested in one module and switches only its statements on.

10.3.2 prints-statements

To monitor the behaviour of the robot in a certain experiment, the print-statements appearing in the source- and logfiles play an important role. These statements have three different natures:

- errors.
- warnings.
- messages.

they appear in the source code as follows:

DPRINT (MESS, ("WsTask: following wall r %lf theta %lf n", (wall-¿r), (wall-¿theta)));

As you can see this is in fact a message statement.

The evaluation of an experiment can be based on these statements. For instance, the amount of error-statements can be a measure of failure. This is different than the fitness-parameter pair method described in the previous section in the sense that it evaluates a complete experiment and not only the separate modules. When a experiment gets a low evaluation, you can look at the evaluation of the modules to determine the main reason for failure. However it appears that not all statements defined as errors actually are errors, so unfortunately this cannot work properly. This can of course be changed, creating new versions of the software.
I have in fact seen logfile of succesfull experiments containing more error messages than unsuccesfull experiments. This is probably because a succesfull experiment runs for a longer period of time, generating more printlines which are defined as an error but are not.
When in an experiment a specific print-statement appears in a logfile, the user can select all previous experiments in which this statement appears in the same logfile (ws-log, ad-log, tp-log etc..). Next all similar experiments that are more recent can be selected to get an idea of what changes were made to, for instance, lose an error message. With similar you can think of the same modules, the same

parameters for the modules, etc.. Of course not all the terminals in the frame have equal values, that would mean that we have two entries representing the same experiment. The different values of a given print-statement can be plotted in a graph to visualize changing of variables in time.

To find the right print-statements that generated the lines in the logfiles a matching-procedure is needed. This particular section of the software I've implemented using SWI-PROLOG. PROLOG is very good in text-matching and therefore it was easier to use than C. A drawback of this PROLOG-module is that it tends to be slow, when inserting a large logfile (the average wall-sensor logfile has about 1500 lines) and for instance 25 print-statements are present in the source-files used, than the PROLOG-routine needs to run 1500*25 = 37500 times. This makes entering experiments with large logfiles a time-consuming business, but because the software runs off-line, not during run-time of the robot this is not really a big problem.
The PROLOG-algorithm is fairly simple, first it filters out the parentheses and the DPRINT at the beginning of the printstatement. Then it just traverses through the characters of both lines, checking whether they are same or of some matching pair like "successfully the lines match.

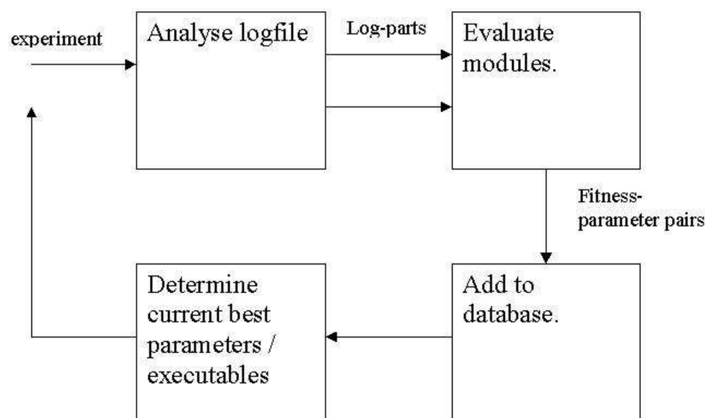

Figure 14: data-flows within EP.

However when printstatements are turned off, the system can perform differently due to the fact that it becomes faster. When modules run on vxworks this is a real problem. However as I said earlier recent work makes it possible to run all modules (except the virtual

car) on UNIX workstations, which are much faster than MARIE's onboard computer running vxworks.

10.3.3 the database

For creating the database for storing the experiments I used a Matisse database. Matisse is an object-oriented database including classes of objects, their attributes and relations between the objects. Also the cardinality of a relationship must be set, the minimum and maximum amount of successors the relationship can have. The inverse of a relationship must be declared as well. In my design the top-level class is *experiment*. This class consists out of subclasses, the date the experiment was done, the user that did the experiment, a list of executables used and a list of logfiles generated. The software for MARIE uses DPRINT statements for debugging. These statements thus appear in the sourcefiles as well as in the logfiles, each experiment in the database therefore contains sourcefiles and logfiles both including a list of DPRINT-statements. The hierachy of an experiment in the database is shown in figure 8.

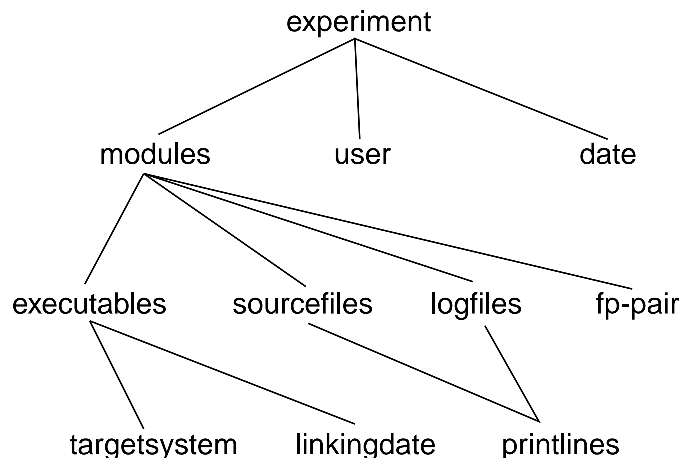

Figure 15: The hierarchy of the database.

Because of the nature of Matisse the lines in the graph work both ways. So for example each experiment has a one to one relationship with a user, while a user has a one to many relationship with an experiment (a list of experiments). Just as in Minsky's frames the terminals are filled with default values when no more specific value is available. These values are just the ones occuring most in the

rest of the database entries. When a new value appears to be the most occuring, the value of the defaults is changed to this value. For some attributes defaults are determined from the file specifications, the default user for instance is the owner of the file. This is determined using the pathname of the file. The experimenter is queried for other values than the defaults.

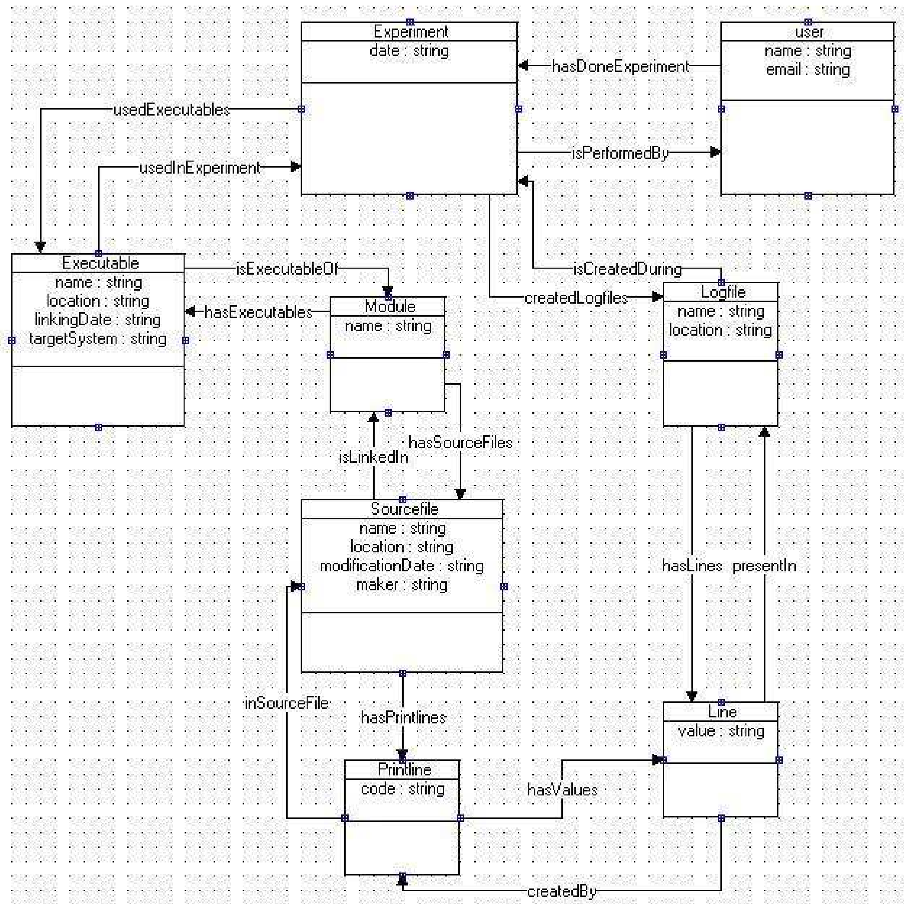

Figure 16: UML visualization of the database.

Because I implemented the platform only to work for the wall-sensor the task-descriptions were ommitted, i.e. they are not included in the database. When we want to evaluate more modules working together to achieve a certain task these 'macro's' [6] are included as element of an experiment and must be considered when constructing a fitness value.

10.3.4 implementation details

In this section I will describe the implementation details of the experimentation platform. I used three languages for implementing the tools needed to fill the database and to evaluate the experiments. These are:

- C/C++ : most of the implementation is done in C. For filling and querying the database I used the Matisse C-API support-functions and Matisse SQL interface.
- awk : awk is very suitable for finding certain structures within text-files. I used awk for finding the parameter values of modules and for extracting the print-statements from the source- and logfiles.
- SWI-prolog : Prolog is a logical programming language based on deduction. It is very suitable for creating matching procedures, also because of this it is often used in natural language processing. As I said earlier I used Prolog for implementing a matching procedure to match printstatements in the sourcefiles with lines occuring in the logfiles.

I made a command line interface querying the user for files and modules used. There are four questions asked:

- which modules are used? (action dispatcher is used by default)
- which executables are used?
- which sourcefiles are used?
- which logfiles are generated?

When a module is entered it is simply placed in database by its first two letters. All modules are separable by these two letters. By these two first letters you can also deterimine which sourcefile or executable belongs to which module for the names of these files also start with these two letters (wall-sensor : ws). When an executable is entered the following information is needed (see figure 16):

- the filename.
- the file location.
- the linking date.
- the target system.

A simple procedure using C-support functions determines these. The filename is simply the part of the full name after the first slash. The location is the rest of the full pathname. The linking date can be found using the UNIX command "ls" and a simple awk script to get rid of the unnessecary information "ls" produces. The target system can be found using the command "file" and again a simple awk script.
Sourcefiles are a bit more difficult because these contain the printstatements that must be matched with the lines in the logfiles. I made an awk-script to find the printstatements and use Matisse functions to place them in the database. The sourcefile has the same attributes as the executable except for the targetsystem, the programmer of the file however is important because there might be a need for feedback.
The logfiles are the most time-consuming to enter, as mentioned before, this is because of the slow PROLOG routine. All the printstatements currently present in the database, as part of the information about sourcefiles, are matched against the lines in the logfile. All lines in the logfile are entered in the database and if there is a corresponding printstatement it is related to it.
Because printstatements occur in sourcefiles and these statements are related to lines in logfiles, the process of creating logfile-parts (see figure 14) is immediate. Sourcefiles are linked to modules so the lines in the logfiles too.

When we have logparts we can evaluate these to gain a fitness value for the corresponding module. As mentioned before I only implemented this for the wallsensor. Again I used an awk-script to find the amount of times the wall was found and lost. The fitness is calculated according the defined function (see section8.3). I could also have uses SQL to select the lines containing this information but I had already implemented the fitness procedure before I did the database. This however is completely equivalent. When the fitness-parameter pairs are added to the database, the current best parameter is determined in relation to an experiment (so also the best set of executables until now is determined).

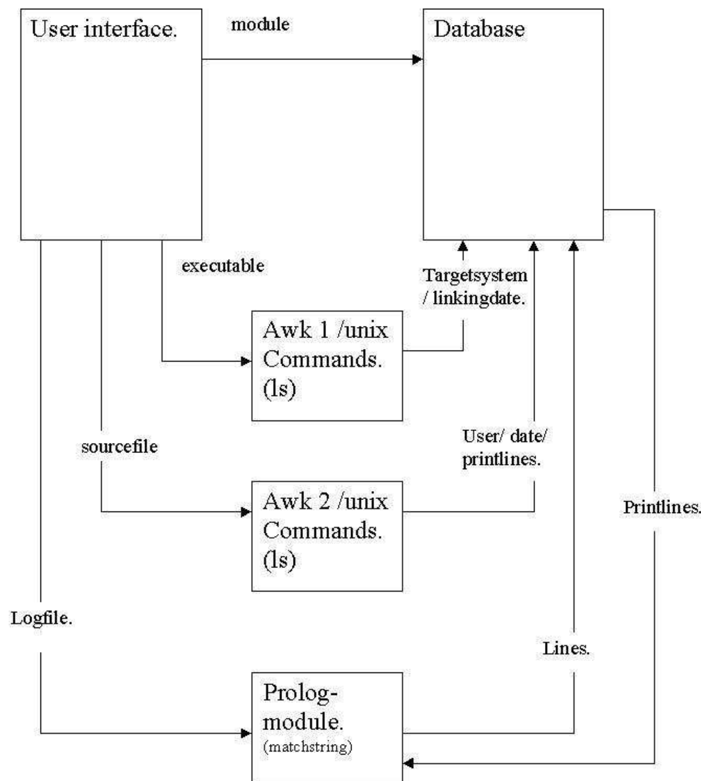

Figure 17: implementation architecture.

10.4 module as the most stable element

Another way of constructing the hierarchy of the database is to place the *module* in the centre. Module is the most stable element because most of the time we want to explore the working of a single module. Often the experimenter knows in which module something is going wrong and therefore only wants to be able to trace logfiles of that particular module. Recently the software for MARIE was adapted such that all modules can run independently on different machines, making them platform independent. Also the executables, source- and logfiles essentially belong to a module which makes it an attractive starting point for evaluating experiments.

10.5 results

By now we can try to find out why the corridor following experiment went wrong. First we filter out the logfile parts of the wall-sensor. From this we find that during the experiment this module actually found walls at both sides of the vehicle but not at the same time

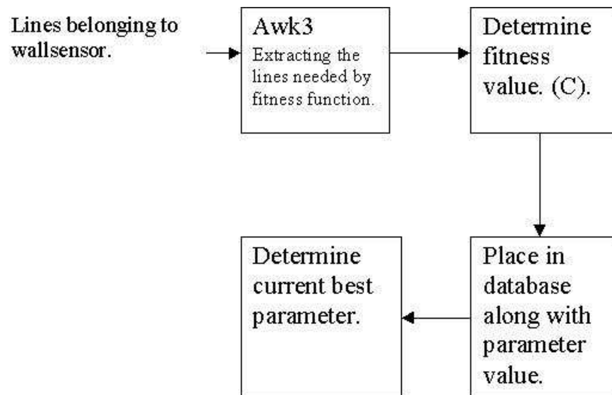

Figure 18: implementation architecture of the evaluation procedure.

thus never finds a corridor. This means that the error is not to be found within the software of the wall-sensor. So it seems that the implementation of the higher level corridor-sensor fails. We can also find that this is probably because the datamanager only accepts one wall at a time and therefore when a second wall is found releases the old one. So the corridor-sensor routine needs to be adapted such that it stores a corridor instead of a single wall. In general the idea of an experimentation platform for mobile robotics has proven to be worthy to investigate. The time researchers spend analyzing logfiles can be drastically minimized. Also the platform can be extended to offer visualization tools for creating data-visualizations like graphs to gain a better insigth into the data-flows and errors that occur.

Figure 20 shows the Matisse editor where an experiment is defined. This particular experiment used two executables and generated a single logfile. Figure 21 a printline with its list of values.

Figure 22 shows the fitness-values through a serie of fifteen experiments. You can see that failures (fitness = 0) really get the average down but overall there is an increase in fitness-values. These experiments of course used different parameters determined by the selection-method described in section 10.2.3.

Figure 23 shows the outcome of a sql-query through the web-interface. It shows all the lines where sensor 225 was responsible for giving distances needed for the model (of the environment).

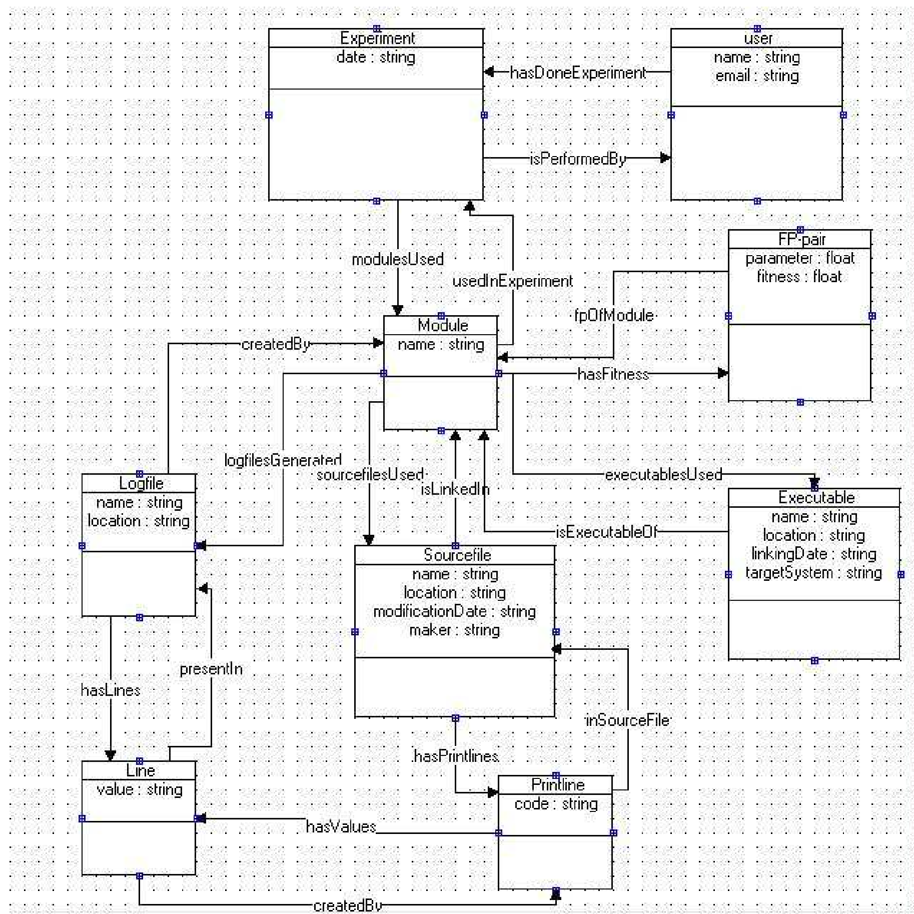

Figure 19: UML visualization of the database with the module at a central position.

11 EP for other modules

When more then one parameter must be set in a certain module, the parameter values must somehow be combined to yield one value. This value can then be associated with a fitness value just like above. Of course the combining of these values is somewhat tricky because one parameter can have much more influence on the success of an experiment. Actually, to make the database useful other modules are already included, however they are not evaluated. The action dispatcher can be included because this is the one module that always must be used in an experiment. The evaluation of the other modules can be the topic of future work. Of course when more modules need to be evaluated fitness-functions must be made for them.

Figure 20: screenshot Matisse editor showing an experiment and its contents.

However when a module depends highly on the input from other modules, like the trajectory controller which depends on the pathplanner for input, in EP we should combine these modules to gain a single evaluation. When we have a low evaluation for the trajectory controller alone, we can never be sure whether the problem lies with the module itself or with the pathplanner.
What can be done of course is make a distinction between the printlines that occur in the logfile, divide them into those belonging to the first module and those belonging to the second. This is one of the most important functionalities of EP, but unfortunately we can also never be sure if an error produced by the trajectory controller actually is due to something the pathplanner does wrong.

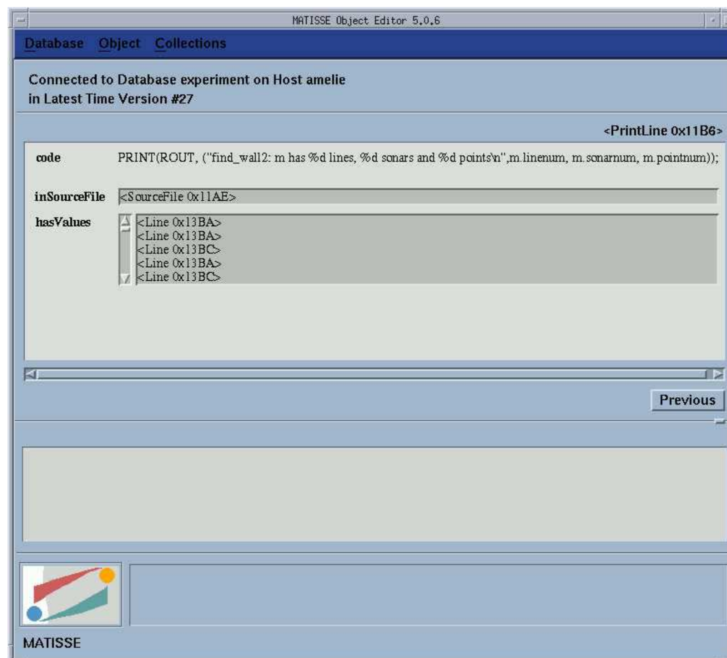

Figure 21: screenshot Matisse editor showing a printline and its instances.

11.1 future work

Some of the possible future work I already mentioned in the previous section, there however are some more technical issues that can be handled in the future. To automatically adapt the default-values at the terminals to the current average, odl-code must be generated. Also the PROLOG-module for line matching can be rewritten (in C) to speed up the process of inserting logfiles to the database. By now the experiments needs to be added to the database by the experimenter after the experiment is finished. The system can be adapted for online use such that the fitnesses can be monitored during the experiment. This means that the PROLOG-module really needs to be rewritten because it is way too slow!

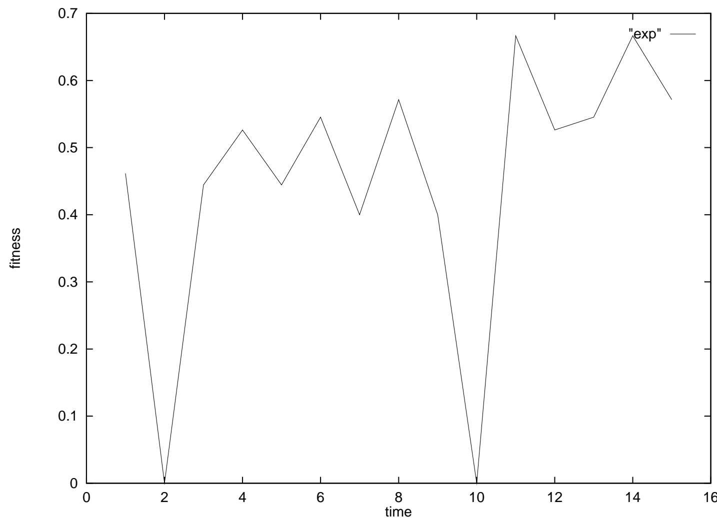

Figure 22: Fitness values of experiments through time.

12 the virtual laboratory; a grid based system for experimental science.

The experimentation platform described in this thesis is in some aspects similar to the virtual laboratory designed for scientific research VLAM-G. In this section I will briefly describe the the grid, the virtual laboratory and the similarities between this and the work described in this thesis.

12.1 the grid

As the computing power grows scientists are beginning to use modelling and simulation more and more to find solutions for their problems. The Grid was was defined during the 1990's as a distributed computing infrastructure for advanced science and engineering. It proposed as a network similar to, for instance, an electricity network which a user can plug into and make use of the resources and data provided by the network. In particular scienctific problems that require a large amount of resources and/or generate huge data-sets are suitable for grid-computation [15].
Several portals to the Grid have been implemented, [15] describes a virtual traffic lab, a portal to the Grid for exploring solutions to (Dutch) traffic problems. Along with the emerge of the Grid, we need applications that can be used on the Grid that are user-friendly and make good use of the efficiency of the Grid. To achieve

select DISTINCT value from LINE where value LIKE 'Model%sensor #225%'

value
Model_Sonar: added sensor #225
Model_Sonarreading: new Polaroid sensor #225 at (1.075000, 0.131000, 0.000000)
Model_Sonarreading: sensor #225 distance 2.422000
Model_Sonarreading: sensor #225 distance 2.516000
Model_Sonarreading: sensor #225 distance 7.480000
Model_Sonarreading: sensor #225 distance 7.598000
Model_Sonarreading: sensor #225 distance 7.606000
Model_Sonarreading: sensor #225 distance 7.635000
Model_Sonarreading: sensor #225 distance 7.960000
Model_Sonarreading: sensor #225 distance 8.541000
Model_Sonarreading: sensor #225 distance 8.574000
Model_Sonarreading: sensor #225 distance 8.724000

Figure 23: outcome of sql-query through web-interface.

this portals and interfaces need to be implemented to support scientists using the Grid. By scientists using the Grid you can think of biologists or experimental physicists. An interesting research question is: "Could roboticists benefit from the power of the Grid?".

12.2 virtual laboratory : VLAM-G

The section CAPS (Computer Architectures and Parallel Systems) at the science faculty of the University of Amsterdam has created a portal for the Grid that can be used for scientific experimentation, in particular for experiments in biology, called VLAM-G (grid-based Virtual Laboratory Amsterdam). The most important aim of VLAM-G is to hide low-level computational details from the scien-

tists. It must provide easy to use applications for doing experiments on the Grid. Also it must provide means of communication between different scientists at different geoqraphical locations, because experiments often require more than one scientist or scientists from different fields of science.

12.3 EP for MARIE and VLAM-G; differences and similarities.

There are several similarities between the experimentation platform for MARIE and VLAM-G. The most important one is the fact that both aim to get more insight into large data-sets, in the case of EP these are the logfiles. With more insight I mean easy access to particular parts of the data and knowledge about the relationship between them. We can for instance find which MARIE-module generated a certain line in the logfile, identifying the "place" where an error occurred.
Also access to previous experiments done by other experimenters is provided by EP, when lots of experiments need to be stored a large amount of storage-capacity is needed. Actually the odl-schema in appendix A that I use is very similar from that of VLAM-G. They also have entries for executable, sourcefiles, etc.. A major difference between the VLAB and EP is of course the fact that EP runs off-line on a single computer and as such is not Grid-based. To return to the question whether roboticists could benefit from Grid-power, I would say "yes they can". In the same way as for instance biologists roboticists generate large amounts of data in the form of logfiles, data-mining such files can therefore be a resource intensive job, and as such suitable for distributed computing. As I said earlier, optimization procedures such as genetic algorithms are suitable within simulation-environments of a robot. However, for efficiently using such algorithms you still need to do a lot of runs. On a single computer this very time-consuming.
Of course all these problems can be solved by a simple network of computers and do not really need an actual Grid-like system. But the advantage of several people doing simulations at different geographical locations and combining the results for a shared optimization solution, you only get from the Grid.
When different roboticists use the same software on different robots, most of the time this is high-level software because specific robot features (size, anatomy, etc..) are low-level details, they can benefit from the experiences of the others. Maybe in the near future we can actually do experiments with a real robot that is at a different loca-

tion, recieving the a completely analysed logfile with details about fitness of the experiment and errors that have occurred. Of course this is very difficult for mobile robots because of the importance of the environment (the environment is "part of the system").
A good example of this "environment being part of the system" is the RoboCup-project. The goal of this project is to set up a team of soccer-robots that can defeat other teams of robots build at different companies or universities. It is obvious that experimenting with these robots at different locations (where for instance no soccer-field is present) is not useful.

13 conclusion

The resulting database makes it possible for MARIE experimenters to get more insight into what happens internally in the software and to get a better idea where and how to fix problems that might arise. Also it can be used to evaluate an experiment by evaluating the modules used. However, as mentioned before, this evaluation isn't the main issue. What is essential is the fact that using this system the robot (or actually the experimenter) learns from its (his) experience as in Minsky's theory. All this to avoid making the same mistake twice. When an error is found and it can be identified as an error which has something to do with a parameter-value, this value can be cleverly adapted using previous experiments in which this error does not occur. This is similar to Arkin's notion of parameter tuning. Different software versions can also be easily compared using the print-statement generated.
The large logfiles that MARIE generates during an experiment can contain printlines from all modules used (depends on whether or not the debug-options for these modules are turned on). A major advantage of EP is that it can filter out the statements that belong to a given module, this makes it possible to determine where the problem must be fixed more easily.
A user is able, after doing an experiment, to generate a graph with the values of a certain printstatement to see how this particular part of the execution evolves through time.

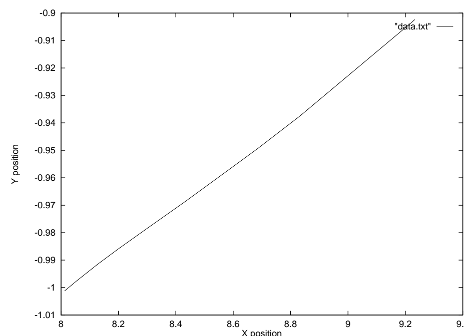

Figure 24: values of printstatement giving the x,y-position of the robot.

By now it is time to answer the two questions mentioned in the introduction. These are:

- is it really possible to gain more insigth in the experiments done with MARIE using the described experimentation platform?

- has this sort of long-term memory effect on the success-rate of experiments with MARIE?

As the screenshots of the Matisse editor and the SQL-trace show, the experimenter can search through the lines in logfiles more easily. In figure 23 you can, for instance see the lines corresponding with the sonar-reading of sensor 225. So if this sensor is the one producing errors this can be easily detected. Also he can relate different experiments in which, for instance the same error occurs, by looking for similarities in both database-entries. I, for instance, found by searching the lines from my corridor experiment that there were never two walls found at opposite angles at the same time. I think this answers the first question.
To answer the second question is little more tricky than the first. As we saw in some cases the fitness of an experiment decreases the overall (average) fitness. This makes the process of learning one that needs a lot of successfull experiments for the fitness-parameter pair to be really beneficial. Also because the fitness can be altered by the experimenter (actually it should be) for new tasks. This makes the design of this function really important for the evolution within experiments.

A odl-schema

This is de odl-schema (object description language) for the database.

```
interface Experiment: persistent
attribute String experimentationDate;
relationship User isPerformedBy[0, 1]
inverse User::hasDoneExperiments;
relationship List¡Executable¿ usedExecutables[0, -1]
inverse Executable::isUsedInExperiments;
relationship List¡LogFile¿ createdLogFiles[0, -1]
inverse LogFile::isCreatedDuring;
;


    interface Person : persistent
attribute String name;
attribute String, NULL email;
;


    interface User : Person : persistent
relationship List¡Experiment¿ hasDoneExperiments[0,-1]
inverse Experiment::isPerformedBy;
relationship List¡Executable¿ hasMadeExecutables[0,-1]
inverse Executable::hasMaker;
relationship List¡SourceFile¿ hasModifiedSourceFiles[0,-1]
inverse SourceFile::isModifiedBy;
;


    interface Module : persistent
attribute String name;
relationship List¡Executable¿ hasExecutables[0, -1]
inverse Executable::isExecutableOf;
relationship List¡SourceFile¿ hasSourceFiles[0, -1]
inverse SourceFile::isLinkedIn;
;


    interface File : persistent
attribute String name;
attribute String location; /* e.g. Path */
```

;

interface Executable : File : persistent
attribute String targetSystem; /* e.g. Operating System Name */
attribute String linkingDate;
relationship User hasMaker[0, 1]
inverse User::hasMadeExecutables;
relationship Module isExecutableOf[0, 1]
inverse Module::hasExecutables;
relationship List¡Experiment¿ isUsedInExperiments[0, -1]
inverse Experiment::usedExecutables;
;

interface SourceFile : File : persistent
attribute String modificationDate;
relationship User isModifiedBy [0, 1]
inverse User::hasModifiedSourceFiles;
relationship List¡Module¿ isLinkedIn
inverse Module::hasSourceFiles;
relationship List¡PrintLine¿ hasPrintLines[0, -1]
inverse PrintLine::inSourceFile;
;

interface PrintLine : persistent
attribute String code;
relationship List¡SourceFile¿ inSourceFile[0, -1]
inverse SourceFile::hasPrintLines;
relationship List¡Line¿ hasValues[0, -1]
inverse Line::createdBy;
;

interface LogFile : File : persistent
relationship Experiment isCreatedDuring[0, 1]
inverse Experiment::createdLogFiles;
relationship List¡Line¿ hasLines[0, -1]
inverse Line::presentIn;
;

interface Line : persistent

attribute String value;
relationship PrintLine createdBy[0, 1]
inverse PrintLine::hasValues;
relationship List¡LogFile¿ presentIn
inverse LogFile::hasLines;
;

B C-source for inserting an executable

References

[1] B.J.A. Kröse, K.M. Compagner, F.C.A. Groen: Accurate estimation of environment parameters from ultrasonic data.*(1993)*

[2] G.A. den Boer: A control architecture for the MARIE autonomous robot.*(1995)*

[3] F.C.A. Groen et al: Organisation and design of autonomous systems; lecture notes.*(1999)*

[4] The c-source of the MARIE wall-sensor module. *(1989-2001)*

[5] WP/UVA-DEL-920924 Software description. (work-package).

[6] F. Terpstra: Een online planner voor MARIE.*(2001)*

[7] D.M. Lyons: A scheme-theory approach to specifying and analyzing the behaviour of robot systems. *(1994)*

[8] R.C. Arkin: Motor scheme-based mobile robot navigation. *(1987)*.

[9] R.C. Arkin: Integrating behavioural, perceptual and world-knowledge in reactive navigation. *(1990)*.

[10] Santamaria, Ram: Learning of parameter-adaptive reactive controllers for robotic navigation. *(1997)*.

[11] M. Minsky: The society of mind. *(1986)*.

[12] M. Minsky: A framework for representing knowledge. *(1974)*.

[13] M.A.Arbib: Parallelism, slides, schemas and frames. *(1975)*.

[14] R.C.Arkin: Behaviour based robotics. *(1998)*.

[15] J.Zoetebier: The virtual traffic lab. *(2002)*.