ESPOO 2003 VTT PUBLICATIONS 510

Ilkka Kauppi

Intermediate Language for Mobile Robots

A link between the high-level planner and low-level services in robots

VTT PUBLICATIONS 510

Intermediate Language for Mobile Robots

A link between the high-level planner and low-level services in robots

Ilkka Kauppi

VTT Industrial Systems

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Automation and Systems Engineering, Helsinki University of Technology, for public examination and debate in Auditorium AS1 at Helsinki University of Technology (Espoo, Finland) on the 12th of December, 2003, at 12 noon.

ISBN 951–38–6251–8 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6252–6 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

VTT, Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Tuotteet ja tuotanto, Otakaari 7 B, PL 13022, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 5888

VTT Industriella System, Otsvängen 7 B, PB 13022, 02044 VTT
tel. växel (09) 4561, fax (09) 456 5888

VTT Industrial Systems, Otakaari 7 B, P.O.Box 13022, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 5888

Technical editing Leena Ukskoski

Otamedia Oy, Espoo 2003

Kauppi, Ilkka. Intermediate Language for Mobile Robots. A link between the high-level planner and low-level services in robots. Espoo 2003. VTT Publications 510. 143 p.

Keywords mobile robot languages, industrial robot languages, autonomous service robots, autonomous field robots, natural language control, field robotics, high-level languages

Abstract

The development of service and field robotics has been rapid during the last few decades. New versatile and affordable sensors are now available, and very importantly, computing power has increased very fast. Several intelligent features for robots have been presented. They include the use of artificial intelligence (AI), laser range finders, speech recognition, and image processing. This all has meant that robots can be seen more frequently in ordinary environments, or even in homes.

Most development work has concentrated on a single or a few sophisticated features in development projects, but even work to design control structures for different levels in robot control has been done. Several languages for industrial and mobile robots have been introduced since the first robot language WAVE was developed in 1973. Tasks can be given to robots in these languages, but their use is difficult and requires special skills of users.

In the future, robots will also be used in homes, and ordinary people should be able to give tasks for robots to perform. This should be done descriptively using natural language as in describing tasks to another person.

In this work an intermediate language for mobile robots (ILMR) has been presented. It makes it easier to design a new task for a robot. ILMR is intended for service and field robots and it acts as an intermediate link from user, an intelligent planner or a human-robot interface to a robot's actions and behaviours. The main principle in development work has been simplicity and ease of use. Neither any deep knowledge of robotics nor good programming skills are required when using ILMR. While easy to use, ILMR offers all the required features that are needed to control today's specialised service and field robots. These features contain sequential and concurrent task execution and response to exceptions. ILMR also

makes it easier to manage the development of complicated software projects of service robots by creating easy-to-use interfaces to all of several subsystems in robots.

It is possible for users to use ILMR to give direct commands or tasks to a robot, but it is intended to be used with higher-level abstract languages, such as sign language or even natural spoken language through a high level planner. An action in ILMR can be given coarsely, i.e. in an abstract way, or in detail. Due to this coarseness, ILMR is suitable to be used with higher-level abstract languages and the set of elementary commands supports directly the use of natural language. With ILMR no complicated models of robots and the world are needed. Only a few measureable parameters for robots are needed and a simple map of the environment is maintained.

ILMR has been implemented in two different kinds of robots, and its use and performance has been studied with simulators and actual robots in a wide variety of tests. The structure and operation of ILMR has proved to be useful and several tasks have been carried out successfully using both test robots.

Preface

This research has been carried out at the Technical Research Centre of Finland (VTT) during two projects. The first project MCLMR (Motion Control Language for Mobile Robots) was done during years 2000–2001 and was funded by VTT. The results of this first project were a starting-point for a second project, which was carried out during years 2001–2003. The aim of this second project was to develop an intermediate language for a robot Workpartner at Technical University of Helsinki (HUT). This second project was financed by the Technology Development Centre of Finland (Tekes) and VTT. An important part of the funding for this work came from the Research Support Foundation of Helsinki University of Technology, which received a donation from Sandvik/Tamrock Ltd.

Professor Aarne Halme was the conductor of this thesis and I thank him for his advice during the work.

Researchers Petri Kaarmila and Marko Blom from VTT have provided valuable help when testing and programming features for the developed language ILMR and I thank them both.

I thank also MSc (tech) Mikko Heikkilä, MSc (tech) Jouni Sievilä and Dr. Jorma Selkäinaho for their help when doing tests with Workpartner at HUT.

I also thank my preliminary examiners, Dr. Hannu Mäkelä and docent Tapio Heikkilä for their valuable suggestions on my thesis.

Finally I thank my wife Iris and children Inga and Henrik for their patience during the days and nights when I was preparing this thesis.

Espoo, October 2003

Ilkka Kauppi

Contents

List of symbols

α	turning angle of the robot
c_{now}	current curvature of the robot
c_i	calculated curvature of the robot
dh	change in heading
ds	cumulated distance in time Δt
Δc	maximum rate for c
h	heading-angle of the robot
h_{start}	starting point heading
h_{end}	ending point heading
l_{obs}	length of checked area in obstable avoidance
p	pitch-angle of the robot
q	robot's position
q_g	goal position
q_{goal}	goal position
q_{start}	starting position
r	roll-angle of the robot
t	time
Δt	time period
U_a	attractive potential
U_r	repulsive potential
w_{obs}	width of checked area in obstacle avoidance
x	local position coordinate

x_{end}	ending point coordinate
x_{ref}	reference path coordinate
x_{start}	starting point coordinate
y	local position coordinate
y_{end}	ending point coordinate
y_{ref}	reference path coordinate
y_{start}	starting point coordinate
ξ	positive scaling factor
ρ	distance function
η	positive scaling factor
ρ_o	positive constant

List of abbreviations

AGV	Automated Guided Vehicle
AI	Artificial Intelligence
ALGOL	ALGOrithmic Language
API	Application Program Interface
ARCL	A Robot Control Language
ASR	Automatic Speech Recognition
BASIC	Beginner's All-purpose Symbolic Instruction Code
C	A Programming Language
C++	A Programming Language
CAN	Controller Area Network
CCD	Charge Coupled Device
cc-Golog	An extension to GOLOG
CURL	Cambridge University Robot Language
DOS	Disk Operating System
DTD	Document Type Declaration in XML and SGML
ESL	Execution Support Language
FDTL	Fuzzy Decision Tree Language
FSTN	A Finite-State Transducer Network
GOLOG	alGOL in LOGic
HAL	Behaviour Language for Robots
HLC	High Level Control of a Robot
HMI	Human Machine Interface

HTML	Hypertext Markup Language
IBL	Instruction Based Learning
ILMR	Intermediate Language for Mobile Robots
IREDES	International Rock Excavation Data Exchange Standard
KL1	Concurrent Logic Programming Language
KRL	Kuka Robot Language
Lazy-QPRM	A Variant of The Probabilistic Roadmap Method
LHD	Load, Haul and Dump
LISP	A Language for Symbolic Data Processing
LLC	Low Lwvwl Control of a Robot
MATLAB	The Language of Technical Computing
MCLMR	Motion Control Language for Mobile Robots
MLC	Middle Level Control of a Robot
MML	A Model-based Mobile robot Language
MRL	Multiagent Robot Language
NASREM	NAtional Standard REference Model
NC	Numerically Controlled
PASCAL	A Programming Language That Enables Structured Programming
PC	Personal Computer
PC104	A Standard for Small PC-Cards
pGOLOG	An Extension to GOLOG
PILOT	Programming and Interpreted Language Of actions for Telerobotics
PRS	Procedural Reasoning System
QNX	Realtime operating system (RTOS) software

RAP	Reactive Action Package
RAPID	A Programming Language for ABB Industrial Robots
RCL	The Robot Command Language
RIPL	Robot Independent Programming Language
RPL	Reactive Plan Language
RoboML	Robotic Markup Language
ROSIV	NC like Language
SGML	Standard Generalized Markup Language
SPLAT	A Simple Provisional Language for Actions and Tasks
TCP	Tool Center Point
TCM	Task Control Management
TDL	A Task Description Language for Robot Control
UNIX	A Powerful Operating System Developed at The Bell Telephone Laboratories
VDT	A Visual Design Tool
VTT	Technical Research Centre of Finland (Valtion Teknillinen Tutkimuskeskus)
WAVE	The First Robot Language
XML	Extensible Markup Language
ZDRL	Zhe Da Robot Language

1 Introduction

1.1 Overview of the history, current state and future of robotics

The roots of robotics go back to the year 1920 when the term robot was first used in a play called “R.U.R” or “Rossum’s Universal Robots”. The writer was a Czech, Karel Capek, and the word robot comes from the Czech word robota, which means tedious labour [Fakta 2001, 1985]. In the play the destiny of the robot inventor was unhappy, first the man makes a robot, then the robot kills the man. This might have inspired a famous writer, Isaac Asimov, to define the Three Laws of Robotics in 1942 [Asimov, 1989]. These laws were built-in in every robot in Asimov’s science fiction books and they prevented robots from causing any harm to human beings.

Since Willard Pollard and Harold Roselund designed the first programmable paint-spraying mechanism in 1938 and the first electronic computer, Eniac, was built in 1946, the development of robotics has gone ahead with rapid strides. People’s hopes of robots were, however, still higher, and it was expected that robots would affect our lives much quicker than they have.

Industrial applications turned out to be an excellent field for robots, and the use of industrial robots increased rapidly since the first industrial robot, Unimate, was in production use in a GM automobile factory in 1961 [Tiedon maailma 2000, 1999]. Applications in industry were similar repetitive tasks, robots were not mobile, and only a few sensors were necessary. When a new task for an industrial robot was prepared, it stayed the same for a long time. From this starting point the development of industrial robots proceeded under control of standards.

Also mobile robots started to develop since Shakey, a mobile robot with vision capability, was built by Stanford Research Institute in 1968 [Nilsson, 1984]. The development of robotics evolved into two separate branches, industrial robots and mobile robots, because the same solutions were not possible within both branches. With mobile robots the tasks are complex and changeable, and environments are changing and not always well known, and a lot of sensors are needed. One of the most important differences between mobile and industrial robots is the fact that mobile robots should be able to act in ordinary environments and among people, whereas industrial robots work inside restricted areas.

Today there are worldwide at least 760 000 robots working in industry, and they are expected to be over 965 000 by 2005. The service robots are mainly used in professional applications (e.g. medical robots, underwater robots, surveillance robots and demolition robots) and it is estimated that 12 000 service robots were in use at the end of 2001 and during the period 2002–2005 there will be another 25 000 units. [Unece and IFR, 2002]

In the future the growth is expected to be highest in entertainment robotics and the number of robots in that area will be about 1.2 million by 2005. One example of entertainment robotics is a riding robot [Kuhnhen, 2002]. A few commercial robots for homes are also available today. The applications are chosen to be simple enough, such as lawn mowers [WWW-reference 1] and vacuum cleaners [WWW-reference 2]. There are also some robotic toys available, such as the Aibo robot dog [WWW-reference 3].

Author's opinion is that in the far future robots will enter our homes and offices. First they will be specialised and take care of special tasks such as cleaning, material transporting, cooking, driving a car etc. Gradually they will become closer to androids, human-like robots. They will become general-purpose robots, which have limited intelligence, good communication skills and the ability to learn. They can use tools, and one robot can handle all usual tasks in an ordinary environment alone or perhaps co-operating with other robots or humans.

More complex and versatile tasks place high demands on programming and task description. Most robot systems today have been programmed in ways described later in chapter 2. Usually robots today can do exactly what they have been programmed to do, and they can even handle some exceptions. A completely new task means new program design, compilation and testing. This phase has to be done usually by an expert with a knowledge of programming environment and designing tasks. In the future, ordinary people should be able to give tasks to robots. This should be done descriptively using natural language as in describing tasks to another person.

The work described in this thesis tries to take a short step towards easier use of service and field robots for application developers, and through easier human-machine interfaces, also for ordinary people. This thesis concentrates only on mobile robotics and all issues related to it, especially programming, control, task planning and execution, supervising and navigation and leaves out industrial robot issues except for comparison purposes.

1.2 Motivation and aims for the study

During the last few decades several intelligent and still more intelligent features for mobile robots have been introduced, and this has made it possible to see autonomous mobile robots more frequently in ordinary environments, or even in homes.

Minerva [Thrun et al., 1999] is a tour-guide robot, which guides people through a museum, explaining what they see along the way. Minerva navigates using odometry, a laser scanner and a camera pointed at the ceiling. The software architecture in Minerva contains about 20 distributed software modules, which communicate asyncronously. There are modules for hardware interface, navigation, interaction and Web-interface. Interaction here means the emotional state of a robot, control of the head and control of speech and sounds. Using a Web-interface a Web-user can select a tour and watch camera images recorded by Minerva and see the robot's position displayed on the map. The tours for Minerva are described in a high-level description language called RPL.

Amadeus [Kamada and Oikawa, 1998] is an autonomous decentralized transportation system, which is aimed to transportation tasks among cells on the shop floor. It consists of mobile agents (AGVs) and cell agents which cooperate in transporting objects. When transportation is needed in a cell, a cell agent negotiates with mobile agents to determine vehicle allocation and assigns the transportation task to the unassigned mobile agent that is able to receive the object to be transported with the least delay. Mobile agents also negotiate with each other to avoid collisions in difficult areas such as a narrow passage. The architecture used in Amaneus is behaviour-based and there are four main behaviour styles, which are normal-running, approaching-station, leaving-station and changing-course. Amadeus was in operation at Fujitsu's Kanuma Plant in 1998.

Helpmate [Evans, 1994] is a famous trackless robotic courier, which has been developed in a project at Helpmate Robotics inc. during 1992–1994. The aim of the project was to develop the technology for intelligent, autonomous mobile robots, or robot carts, that can find their way around a factory, hospital, or similar place by sensing and avoiding obstacles and taking alternative routes if a path is blocked. Nowadays, Helpmate Robots are delivering medicines, supplies, prepared food, X-ray images and other material in about 100 hospitals in the United States and Canada, and there are already marketing arrangements with companies in Europe and Japan. Helpmate is equipped with sonar, vision and contact sensors and it can use elevators without assistance. The software is based on its own behaviour language (HAL) with rule files. Typical installation time for a hospital is estimated to be 15 man-days.

Autonomous load-haul-dump (LHD) device [Mäkelä, 2001] has been developed for underground mine applications. A new route for the LHD is taught by driving the machine through the route, either by an operator or using tele-operation. The new route is then sent to the Mining Control System (MCS) and the LHD is ready to start the new production route. The navigation is based on odometry with correction by laser-scanner readings from the tunnel walls.

DAVID is a robot system that was meant to accomplish a variety of office tasks, such as collecting and delivering mail or cleaning up the offices [Prassler et al., 1997]. DAVID has functions for perception, planning, motion and manipulation. To combine these functions in complex tasks DAVID is implemented with AUTOGNOSTIC [Stroulia and Goel, 1999]. It is implemented in Lucid Common Lisp and is responsible for the configuration, the monitoring and the repair of the task structure.

The problem from the point of view of a developer with all of these examples, and with many other robots too, is the way they are programmed. They all have their own software architecture, which is difficult to understand if you are not familiar with that specific architecture. If you understand one system, you still have difficulties to understand some other system. They are ready systems for users, but closed systems for developers. They operate only in the way they were originally programmed, and an ordinary user has no way to define a totally new type of task for any of these robots.

During the last decade VTT has also developed autonomous vehicles for various applications and environments like underground mines, electronics factories and container yards [Lehtinen et al., 2000]. During continuous development work it was noticed that a new application always requires a lot of software modification and generation despite of earlier software development and experience in that field. To make it easier an intermediate motion control language for mobile robots

(MCLMR) [Kauppi et al., 2001] was first developed. It gave means to flexibly design a new task for a robot and change it when needed.

This previous work gave guidelines for further development. At the starting point of this development work there were drivers and interfaces available for sensors and actuators and control software for several subsystems of a robot, such as navigation, perception and motion control. It was assumed that a real-time operating system would be available in robots and several processes could be run concurrently. Some of the processes could even be run in separate processors. The missing factor was a structure which would link all subsystems together in a way that would enable the developer of one subsystem to control and utilize other subsystems easily and in a desired way and would give the user or the high level planner the means to design and control a new task or a subtask for a robot. This missing factor was named ILMR (the Intermediate Language for Mobile Robots) and its requirements are stated in Table 1.

Table 1. The requirements of ILMR.

1	Usability with service and field robots
2	Separation of robot dependent and independent parts
3	Easy control and access to all subsystems in a robot
4	Use is akin to natural language
5	Parameters for primitive operations can be given roughly or in detail
6	Interpretative system
7	Possibility to use behaviours
8	Easy way to use concurrent and sequential subtask/operation execution and monitoring
9	Inhibition of harmful/damaging commands
10	Intelligent halt of a running task
11	Monitoring of the status of subsystems
12	Emphasis on movement commands

The first requirement states that ILMR should operate with service and field robots. Those are usually mobile robots that are in the future going to operate in homes and offices among humans and other robots. The commands for those robots should be given in a spoken or written natural language by ordinary people. The goal of this thesis is to make the use of these robots easier in everyday life.

The second requirement states that ILMR should have two separate parts. One part contains the algorithms of ILMR and is independent of any robot, and the other is used to connect it to a certain robot and environment. This implies that a task description with ILMR is independent of the robot, and the task could still be valid if the robot is replaced by a different robot.

The third requirement states that, on the one hand the developer should have easy access to all subsystems in a robot, and on the other hand the user should be able to utilize all subsystems to perform designed tasks without knowing many or even any details of them. In addition, managing software development is also easier in complicated service robots when the developer has a standard interface to other subsystems by means of ILMR.

The fourth requirement says that no deep knowledge of robotics nor good programming skills are required when using ILMR. The given commands are akin to natural language and simple tasks can be given like one person to another even directly without any high-level planner. The development of a planner is also easier because it can be done in steps from direct mapping from input to output in order to form an intelligent interpretation of spoken language, and all the time the output of the planner remains a readable task description with the commands approaching natural language.

The fifth requirement states that the operations can be given roughly or in detail. This coarse description enables the easy connection to the more abstract planning layer in robots and detailed parameters enable the precise movements of the robot. With coarse parameters the task description can be more abstract without exact figures. Those will be solved by ILMR during the execution according to the sensor readings.

The sixth requirement states that ILMR should have an interpreter. It is necessary because of the need to give, stop and change tasks quickly at any time. The interpreter should also check the following commands in case of any effects on the current command. This is useful when it is desirable that two subtasks be combined together smoothly.

The seventh requirement states that there should be a possibility to control and use behaviours in robots. One example of behaviour is obstacle avoidance. If it is switched on, then it will overdrive the output of the position controller of ILMR in case of obstacles.

The eighth requirement says that both sequential and concurrent operations should be available and they should be available without any complex control structures. Also here the ease of programming is taken into account and the implementation of the language should take care of problems of concurrency. The user should however be able to control the concurrency/sequential execution if desired.

The ninth requirement says that ILMR should be aware all the time of the status of the robot and reject any given command that could somehow harm or damage the robot.

The tenth requirement states that the user or other software should be able to stop the current task or operation at any time, but this should be done intelligently so that safety is maintained concerning the robot, the cargo and the environment.

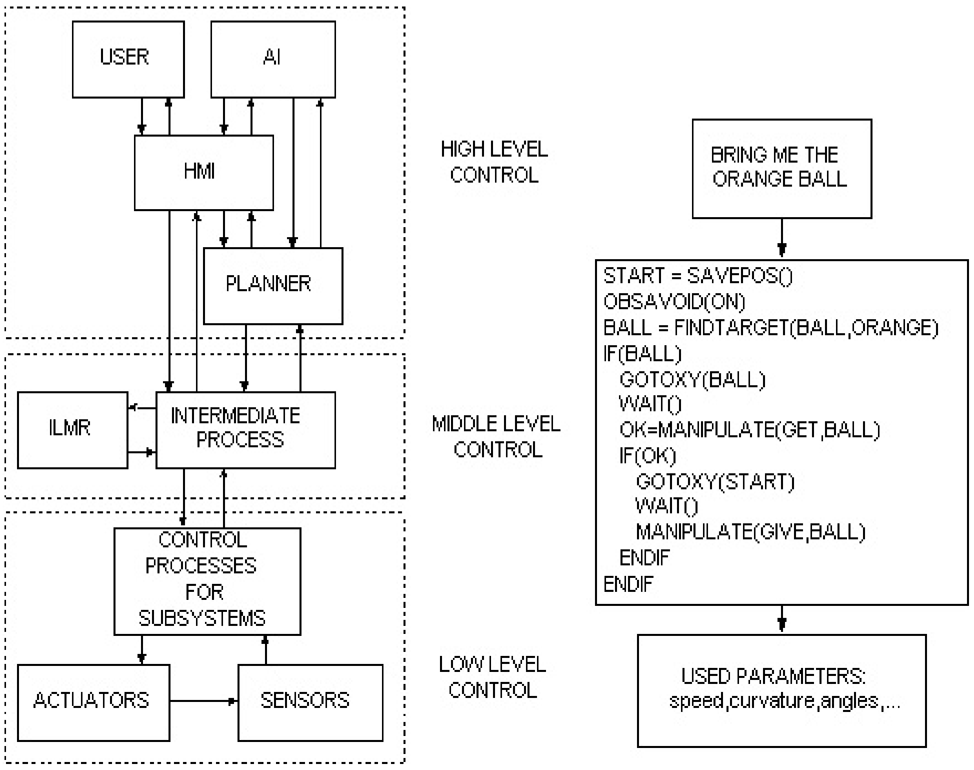
The eleventh requirement states that ILMR should monitor the status of the robot and its subsystems and do any necessary operations when an emergency or other exception occurs.

The last requirement states that most of the commands at this stage handle the movements of the robot, which are maily generic for different robots. In the future the commands for manipulators will be added, but at the moment it is enough that ILMR only triggers the sequence of manipulation, which is controlled by an external control process and acknowledges the calling software or the user when manipulation is done.

It is difficult to put the requirements in order of importance. Requirements 3–8 are, however, more important than the others.

In this thesis the control of a robot is divided into three levels, and the left part of the Figure 1 gives an overview of the control levels in a service robot. The high level control (HLC) contains all intelligent features such as learning, reasoning and concluding. The task of a HLC is to decompose the intentions of the user into subtasks and monitor their execution. The abstract task descriptions come from the user or an artificial intelligent unit (AI). From the user the descriptions go through a human machine interface (HMI) to a planner, which can decompose them into the form which is understood at the middle level control (MLC). From the AI unit the descriptions can go also through the HMI or directly to the planner. The middle level control enables the execution of behaviours, primitive actions and supervision of robot’s subsystems. The algorithms of ILMR are running at this level and a robot dependent intermediate process is used to connect ILMR to a certain robot environment and HLC. The low level control (LLC) contains drivers and controllers for all sensors and actuators and the control of all subsystems in a robot.

The control levels high, middle and low also describe the levels of abstraction. At high level the descriptions and goals are expressed with abstract terms of natural language such as “bring me the orange ball”. At middle level this goal is decomposed into subgoals, but the descriptions can still contain some abstract terms. At low level the descriptions are precise without any abstraction. This can be seen from the right part of the Figure 1. The output of the high level can also be a list of scripts such as “findball(orange)-bringball-giveball” but finally these scripts are decomposed into the commands shown in Figure 1.



```
START = SAVEPOS()
OBSAVOID(ON)
BALL = FINDTARGET(BALL,ORANGE)
IF(BALL)
  GOTOXY(BALL)
  WAIT()
  OK=MANIPULATE(GET,BALL)
  IF(OK)
    GOTOXY(START)
    WAIT()
    MANIPULATE(GIVE,BALL)
  ENDIF
ENDIF
```

Figure 1. The control levels in a service robot and the task description at different abstraction levels.

1.3 Scientific contribution of the dissertation

Mobile robots have been evolving during the last few decades alongside with the means to control robots. One way to make the control of robots easier is the use of robot languages. With a suitable language a developer can easily design a new task for a robot and utilise different services and sensors in a robot. A robot language can be a high-level or a low-level language. With high-level languages a task description can be abstract and low-level languages enable access to actuators and sensors. Some languages can contain features from both levels. A working solution with current specialized service and field robots is a combination of two languages. One language is operating at a low level and another at a high level. The high level language could be for example a sign language, which is interpreted from camera images with pattern recognition and decomposed into lower level language commands.

Most languages are extensions of existing programming languages such as LISP, C, C++, ALGOL, PASCAL, etc. Design of a new task for a robot with these languages is like writing a new program for a computer. A new task should be designed, coded, compiled and tested before it can be used in a robot and the developer should have good programming skills.

Some languages contain an interpreter and databases containing skills, behaviours, object descriptions, etc. With these languages a new task can be given using an interpreter, but it must be defined beforehand, or at least the subtasks, which are used to construct it, must be defined beforehand in a database, and this definition can be difficult for users. This can be seen from examples shown in chapter 2.

The first scientific contribution of this thesis is a new method which makes it easier to design a new task for a robot. This can be seen from the chapter 4.1. It also gives means of easily control and utilize the subsystems of a robot and so make the software development and management easier by providing a standardized interface to subsystems. This is important with today’s complicated service robot, which is equipped with several subsystems and many people are developing software for it. An example of such a robot is Workpartner, which is explained in the chapter 5.

The second scientific contribution of this thesis is a new method above the operating system to handle concurrency and sequential actions without any special control structures. This method is explained in chapters 4.3 and 4.4.

The third scientific contribution of this thesis is a method to give commands coarsely or in detail. The coarse parameters enable the easy connection to the more abstract planning layer in robots and detailed parameters enable the precise movements of the robot when desired. For example turn(left) and turn(d=90,c=0.45) both turns robot to the left, but first one is easier to give and understand and it contains internal intelligence to define left according to environment with help of perception data. Due to this coarseness, ILMR is suitable to be used with higher-level abstract languages and the set of elementary commands supports directly the use of natural language.

The fourth scientific contribution of this thesis is the way that variables are handled. The user can handle the variables with their names not knowing their content and structure. Each command and control structure in ILMR knows how to handle the different fields in variables.

The final aim with ILMR is to develop it to a common description language for mobile robots such as MATLAB for matematical problems. The language itself contains all the basic elements which are needed to control a robot, and the user can concentrate only on tasks to be carried out. When a task is defined by ILMR, then the robot can be replaced by another, different kind of robot, and still the task definition could remain valid.

1.4 Outline of the dissertation

This dissertation is a descriptive case-study which describes the research to develop an intermediate language to control mobile robots and studies its application and use with two different test vechicles.

The dissertation is organised as follows:

Chapter 1: *Introduction.* Introduction gives a short overview of the history and future of the robotics. It also describes how the topic for this dissertation has been chosen.

Chapter 2: *State of the Art*. This chapter clarifies the concept language and describes how the language development for mobile robots has proceeded during last 15 years. Also differencies between industrial robot languages and mobile robot languages are studied.

Chapter 3: *Autonomous Service and Field Robot.* This chapter explains the concept of an autonomous robot and its subsystems. It clarifies what functions are needed and how control and issuing commands can be arranged. It also handles the terms teaching, learning, skills and language.

Chapter 4: *Intermediate Language for Mobile Robots (ILMR)*. The structure and use of ILMR is explained in this chapter. An example task is studied in detail. Software architecture and associated processes are explained as well as the interpreter, command classes and elementary commands. Obstacle avoidance, trajectory generation and position controller are also explained in detail. Features of concurrent and sequential execution in ILMR are also examined in this chapter.

Chapter 5: *Experiments with ILMR*. Two test vehicles are used to study the feasibility and use of ILMR in robots. In this chapter both test vehicles, Workpartner, a centaur-like robot which was developed at HUT, and MoRo, which is an AGV developed at VTT, are described. A wide variety of tests have been carried out with both test vehicles to study the use and feasibility of ILMR in various situations and environments.

Chapter 6: *Conclusions*. Summary of the research and recommendations for future work are presented in the last chapter.

2 State of the art

According to definition a robot is a machine, which can flexibly do many kinds of tasks, and this is accomplished by just modifying a program and without any modification of the mechanical parts. So as long as there have been robots there has also been more or less a need for an easy way of giving new tasks to a robot. However, the development of mobile robots during the last few decades has mostly concentrated in special, single and intelligent features with only one particular task in mind. Since the first robot language WAVE was developed in 1973 at the Stanford Artificial Intelligence Laboratory [Paul, 1977], there have been many attempts to develop control languages for robots, but this study concentrates only on attempts during last 15 years.

First the concept of language is clarified, and then a typical language for industrial robots is examined to show why it is suitable for industrial applications but not for mobile robots. After that the development of languages for mobile robots is examined by means of example languages and it is shown how languages have developed from simple action-trigger commands to abstract high-level languages. It can also be seen from examples that robot languages can be based on different approaches. The basis can be, for example, behavioural or hierarchical. It can also be natural language or hybrid of these all. The language and its run-time implementation in a robot are not separated in this thesis but they are handled as one unity.

2.1 Language

A language is a system of communication, which usually is connected to human spoken language and which is based on an arbitrary system of symbols. To be more explicit it can be understood to be any means of any living creature to report, warn or express something [Fakta 2001, 1981]. One example of such languages is the honeybee’s dance language, which is used by honeybees to tell their nestmates about discoveries they make beyond the hive [Kirchner and Towne, 1994]. The most important feature of a language is its ability to produce messages. Where a message is an object which has a meaning for a certain receiver [Turchin, 1997].

In a computer the executable control program is formed of a sequence of machine-language commands. A machine-language command consists of a numerical code, which contains the type of the command and the source and destination addresses of the information. To make programming easier several high-level programming languages have been developed. Instead of numbers and addresses the developer can now use words and names. Before the use of such a high-level control program it must be compiled to machine-language code. This is done by compilers which have been developed for each language. There are also interpretative languages, which means that commands are interpreted when they are encountered. These languages are less powerful than compiled languages, because errors must be checked at run-time.

Programming languages for computers can be classified according to several criteria. One classification is shown in Table 2 [WWW-reference 4]. The same source also lists 133 languages, but there are much more available around the World.

Different languages have different aims and are suitable for different purposes. For example MATLAB is a mathematical language, which has been developed to solve mathematical problems. It has built-in functions for powerful mathematical analysis, but it is not suitable for real-time control of a mobile robot. HTML is a markup language to describe how information appears in Web browsers, but it is not suitable to solve mathematical problems.

Table 2. A classification of programming languages.

Compiled	Imperative	Open Source
Concurrent	Interface	Parallel
Constraint	Interpreted	Procedural
Database	Language-OS Hybrids	Prototype-based
Dataflow	Logic-based	Reflective
Declarative	Markup	Regular Expressions
Distributed	Multiparadigm	Scripting
Functional	NET	Specification
Garbage Collected	Obfuscated	Visual
Hardware Description	Object-Oriented	Wirth

The requirements of ILMR implies that it can be classified as concurrent, imperative, interface, interpreted and scripting language.

2.2 Industrial robot languages

There are several languages available for industrial robots. One group is high-level languages such as PASCAL-like languages RAPID (ABB) and KRL (KUKA) or BASIC-like languages such as MBA4 (Mitsub.) and Karol (Fanuc). Another group is NC-like languages such as ROSIV (Reis) and the third group is proprietary languages such as V+ (Adept). [Freund et al., 2001]

RAPID is a programming language for ABB industrial robots [WWW-reference 5]. RAPID has means to define constants, variables and other data objects. It has commands for motion, tool and I/O manipulation and monitoring mode. It has arithmetic and logic operators and a control structure for programs. It can also communicate with an external PC and via serial interface. The movements of a robot are programmed as pose-to-pose movements. The basic motion characteristic must be defined before motion can start. This contains position data (end position and external axes), speed data, zone data, tool data and work-object data. The user can also define maximum velocity, acceleration, management of different robot configurations, payload, behaviour close to singular points and so on. There are for example 10 different move commands availabe in RAPID and they are MoveC, MoveJ, MoveL, MoveAbsJ, MoveCDO, MoveJDO, MoveLDO, MoveCSync, MoveJSync and MoveLSync. An example of MoveL command is shown below.

MoveL ToPoint := [940,0,1465,0.707,0,0.707,0], Speed := v50, Zone := z50, Tool := gripper1

World zones can be defined within the working area of the robot to indicate that the robot's TCP is a definite part of the working area or to delimit the working area in order to prevent a collision with the tool or to create a common working area for two robots. RAPID has also special commands for spot welding, arc welding and glueware.

The commands of RAPID are suitable for industrial applications, where actions are happening inside a well defined working area and movements are in fact tool movements to get desired manipulations for workpieces done correctly. All required actions and movements can be done with a huge set of commands, definitions and parameters.

A task definition for a robot with RAPID is a demanding job. One should know which command is the right one for a certain situation and usually the programs are first tested with simulators which are available for industrial robots. A task for an industrial robot has to be well designed, because time is money in business and all movements should be optimal.

With mobile robots task definition can't be so strict, since the environment can be partly or totally unknown, and unforeseen events can occur any time. Task definition and even a totally new goal for the mission must be replanned in case of unforeseen events. Defining tasks for mobile robots should also be easy, since it can be occasional.

2.3 Mobile robot languages

In the following chapters the mobile robot languages are divided into six categories. In each category the languages are studied and finally they are mapped to the requirements of ILMR.

2.3.1 Reactive robot languages

2.3.1.1 Behaviour language

The Behaviour Language utilizes reactive behaviour-based methods for robot programming. It has a Lisp-like syntax and behaviours are represented by a set of rules. These rules are compiled to augmented finite state machine (AFSM) representations which can be compiled for the target processors. This language is based on subsumption architecture [Brooks, 1986] [Brooks, 1991] (see Figure 18) and an AFSM encapsulates a behavioural transformation function where the input to the function can be suppressed or the output can be inhibited by other components of the system. The details of the subsumpion architecture are explained in the chapter 3.2.

When an AFSM is started, it waits for a specific triggering event and then its body is executed. In the body it is possible to perform primitive actions or to put a message in order to interact with other AFSMs. The events can depend on time, a predicate about the state of the system, a message deposited into a specified internal register, or other components being enabled or disabled. There are no plans or procedures available, but it is possible to define macros and use them in the definition of behaviours. [Pembeci and Hager, 2001]

2.3.1.2 RAP and RPL

RAP system task-net (Reactive Action Package) was originally released by James Firby in two publications [Firby, 1987] and [Firby, 1989]. Later he added an extension to the RAP to enable the effective control of continuous processes [Firby, 1994]. The RAP system was applied with the animate agent architecture, which is shown in Figure 2.

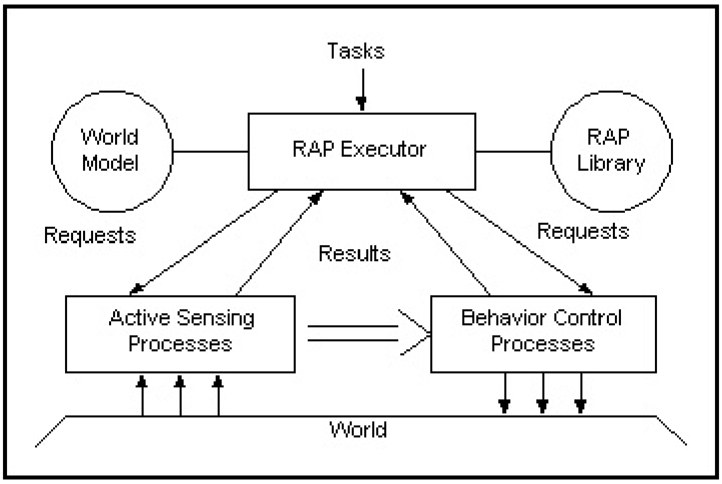
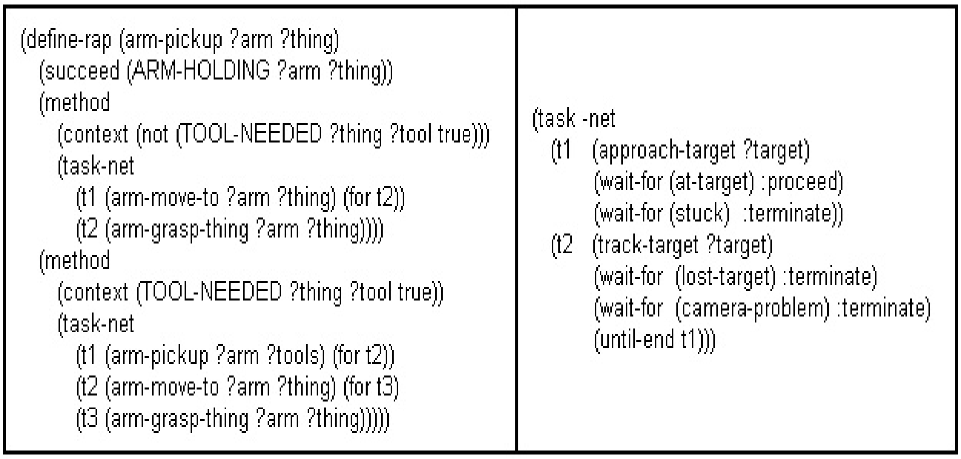


Figure 2. The Animate Agent Architecture [Firby, 1994].

In the RAP system the task is first selected for execution. If the task represents a primitive action, it is executed directly, otherwise the corresponding RAP is searched from the RAP library. Then the success of the task is checked, and if it is true, then the task is complete and the next task is allowed to run. If the task has not yet completed, then one of the methods is selected according to suitable tests, and the task waits until the selected method is complete. The selected method can also contain several subtasks. When the method has completed then task success is checked again, and if it is true, then the next task can be run. If it is not, the next method is selected for execution. An example of a RAP is shown in Figure 3 on the left and a concurrent task net on the right.



```
(define-rap (arm-pickup ?arm ?thing)
  (succeed (ARM-HOLDING ?arm ?thing))
  (method
    (context (not (TOOL-NEEDED ?thing ?tool true)))
    (task-net
      (t1 (arm-move-to ?arm ?thing) (for t2))
      (t2 (arm-grasp-thing ?arm ?thing))))
  (method
    (context (TOOL-NEEDED ?thing ?tool true))
    (task-net
      (t1 (arm-pickup ?arm ?tools) (for t2))
      (t2 (arm-move-to ?arm ?thing) (for t3))
      (t3 (arm-grasp-thing ?arm ?thing)))))
```

```
(task -net
  (t1   (approach-target ?target)
        (wait-for (at-target) :proceed)
        (wait-for (stuck)  :terminate))
  (t2   (track-target ?target)
        (wait-for  (lost-target) :terminate)
        (wait-for  (camera-problem) :terminate)
        (until-end t1)))
```

Figure 3. An example RAP and A Simple Concurrent Task Net.

The RAP in Figure 3 defines how to pick up something. There are two methods to reach this goal.

A task-net system has been implemented and used to control a real robot doing vision-based navigation tasks at the University of Chicago.

The RAP-system is efficient with mobile robots and it allows concurrent task execution and response to exceptions. It needs, however, some expertise to define RAPs for a task.

RPL (Reactive Plan Language) is mainly based on RAP. It is geared to high-level robot planning, and to represent advisory plans for emergency situations. Many of the concepts of RAP have been carried over in RPL. The syntax of RPL is however more recursive and expressed in Lisp style. More high-level concepts (interrupts, monitors) have been made into explicit constructs, and the interpreter does not attemp to maintain a world model that tracks the situation outside the robot. [WWW-reference 6]

When these reactive languages are compared to requirements of ILMR it can be noticed that only a part of them are satisfied. For example RAP and RPL satisfy the requirements 1, 2, 6, 7, 8, 10 and 11 in Table 1 but the important requirements 3, 4 and 5 are not satisfied.

2.3.2 Mobile robot control languages

This category contains a set of languages which are not suitable for other gategories. Many of them are extensions of existing programming languages such as Pascal, Lisp, Algol, etc. The languages are presented in chronological order. The oldest are handled first and in a shorter manner, while the newest are handled in more fine detail.

2.3.2.1 ARCL

ARCL (A Robot Control Language) [Elmaghraby, 1988] was based on Pascal-like syntax. It was a compiled language and the developed cross-compiler required three passes before the executable code was ready to be downloaded and executed in a robot. This language has a Pascal-like syntax with sensory-control and motion-control commands. An example of an ARCL-language command is MOVA(GRIP,HI,CONT,MED) which opens the gripper on the robot. This language was implemented on the HERO-1 robot.

ARCL emphasized sensory-based programming rather than planned trajectory motion and was designed for educational robot HERO-1. Concurrent execution was not possible with ARCL and the use of ARCL was also difficult because it required allways three passes before it was ready to be executed in a robot.

2.3.2.2 ZDRL

ZDRL (Zhe Da Robot Language) was a motion oriented robot language [Chen and Xu, 1988]. It was an interpretative system and the language was composed of system commands and program instructions. System commands were used to prepare the system for execution of user-written programs. ZDRL included 32 system commands and 37 program instructions, and it contained capabilities for program editing, file management, location-data teaching, program executing and program debugging. This language was implemented in a Rhino XR-1 robot.

This language was a combination of a high-level language and a low-level language. Assembly language was used to do very time-consuming operations and a high-level language was used to store program steps and variables and to perform trajectory planning.

2.3.2.3 MML

MML was a Model-based Mobile robot Language which was developed at the University of California at Santa Barbara [Kanayama, 1989]. It is a high-level off-line programming language which contains functions for high-level sensor functions, geometric model description and path planning and others. This language contains an important concept of slow and fast functions, which architecture is essential for real-time control of robots. A slow function is executed sequentially, while a fast function is executed immediately. The second important concept is the separation of the reference and current posture, which makes precise and smooth motion control and dynamic posture correction possible. An example program to execute a smoothed square path with MML is shown in Figure 4.

MML was a good attempt to propose a candidate for standard set of locomotion functions for mobile robots.

```
user()
{
 POSTURE p,a,b;
 int i;

      def_posture(200.0,0.0,0.0,&a);
      def_posture(200.0,0.0,0.0,&a);

      set_rob(def_posture(-100.0,-200.0,0.0,&p));
      for(i=0;i<4;i++)
      {
         move(comp(&p,&a,&p));
         if(i<3)
           move(comp(&p,&b,&p));
         else
          stop(comp(&p,&b,&p));
      }
}
```

Figure 4. An example program with MML.

2.3.2.4 RIPL

RIPL (Robot Independent Programming Language) [Miller and Lennox, 1990] is based on an object-oriented Robot Independent Programming Environment [RIPE]. The RIPE computing architecture consists of a hierarchical multiprocessor approach, which employs distributed general and special-purpose processors. This architecture enables the control of diverse complex subsystems in real time while co-ordinating reliable communications between them. The software classes in RIPE are defined to represent the physical objects in robots and work cells and thus the communication interfaces to these generic software classes in RIPE become the general device independent language which can be used to programme the cell or a robot. RIPE has four primary layers: task-level programming, supervisory control, real-time controls and device drivers.

RIPL is in fact a set of routines, which are defined inside classes of RIPE, and the language is a compiled language.

2.3.2.5 RCL

RCL (The Robot Command Language) [Fraser and Harris, 1991] is based on NASREM architecture (NAtional Standard REference Model) [Albus et al., 1987]. NASREM is a hierarchical and collateral architecture, with representations at increasing levels of abstraction in higher levels of the architecture. Upper levels represent symbolic, intelligent system commands, while at lower levels the model is geometric and dynamic intelligent control. The details of NASREM architecture are explained in the chapter 3.2. The syntax of RCL is hierarchical and the language elements are defined from other RCL elements. Telerobot exits at level 4, Arms and Cameras at level 3 and so forth, e.g.

```
<Telerobot#1>::= Arm#1><Arm#2><Platform><TV_Camera#1>

<Arm#1>     ::= <Joint#1><Joint#2><Hand#1>
```

Example commands with RCL can be as follows

```
position_trajectory(time1, x1,y1,time2,x2,y2 …)

turn_on_place(time1,direction1,time2,direction2…)
```

These commands will be interpreted to produce actuator commands.

2.3.2.6 FDTL

FDTL (Fuzzy Decision Tree Language) is a language, which is based on a computational model that combines fuzzy rule-based control with the hierarchical nature of decision trees [Voudouris et al., 1995]. FDTL contains a compiler which produces C code, which can be executed on a variety of platforms as embedded or independent programs. The language has been tested with simulators and a controller called GC (Garbage Collection) for an autonomous test vehicle that steers using the differential velocities of its two driving wheels. The vehicle is equipped with ultrasonic proximity sensors together with other sensors that give the angle and distance to various goals. The task of the robot is to collect and dispose of garbage, while avoiding obstacles, and moving to a recharging point, if the battery condition is low, and to service point if damage is detected. Fuzzy decision tree architecture is a feed-forward system. The incoming information from the sensors together with internal variables (goal position, battery level, damage detectors, state variables etc.) are fed to the tree. The tree spreads activation from the root to the leaf nodes. All leaf nodes in this application have only two outputs, the velocities of each wheel. Where a rulebase outputs to an internal node the output value will act to “scale” the relative contribution of the rulebase at that node. For instance, root of the tree spreads activation progressively to Obstacle Avoidance as the vehicle approaches obstacles, otherwise it spreads activation to the goals-subtree. Thus any of the leaves can have an effect on the output velocities but this effect is moderated by the activation level of its ancestor nodes.

The FDTL description for GC rulebase is shown in Figure 5 and the FDTL definition for d, one of the distance sensor readings, is shown in Figure 6 on the left and the expression for the state of the “has_rubbish” detector is shown on the right in Figure 6.

FDTL is a language in which fuzzy decision trees are described and a tool for programming robots and intelligent systems. It contains a compiler that produces ANSI standard C-code, which can be compiled to executable code in a variety of platforms. The readability of the language is good, but the weakness is its difficult use, which requires two steps in compilation phase.

```
rules
    begin
      GC
        begin
            if dmin==NEAR  then ObstacleAvoidance;
            if dmin==FAR  then Goals;
                ObstacleAvoidance
                    begin
                        if d1==VN and d2==VN and d3==VN and d4==VN
                            then Vleft:=NS and Vright:=NS;
                        if d1==VN and d2==NE and d3==NE and v4==VN
                            then Vleft:=PS and Vright:=PS;
                    end
                Goals
                    begin
                        if energy<=20 and damage=='yes'
                            then go_to_charger;
                        if energy>20 and damage=='yes'
                            then go_to_service;
                        ...
                        if energy>20 and damage=='no' then do_jobs;
                            go_to_charger
                                begin
                                ...
                                end
                            go_to_service
                                begin
                                ...
                                end
                            do_jobs
                                begin
                                    if hold_rub=='yes' then go_to_bin;
                                    if hold_rub=='no'  then go_to_rub;
                                        go_to_bin
                                            begin
                                            ...
                                            end
                                        go_to_rub
                                            begin
                                            ...
                                            end
                                end // end of do jobs
                    end // end of Goals
        end // end of GC
    end; // end of rules
```

Figure 5. FDTL Description for GC Rulebase.

```
numeric d
{
  trapezoidal VN  0   0   80  120;   // Very Near
  triangular  NE  80  120 160;      // Near
  trapezoidal FR  120 160 255 256;  // Far
}
```

```
// carry rubbish
symbolic hold_rub
{
  'yes';
  'no';
}
```

Figure 6. Examples of FDTL definitions.

2.3.2.7 PRS

PRS (Procedural Reasoning System) is a high-level Control and Supervision language for autonomous robots [Ingrand et al., 1996]. It executes procedures, plans and scripts in dynamic environments. The kernel of PRS is composed of three main elements: a database, a library of plans and a task graph. The database contains facts representing the system view of the world and is updated automatically when new events appear. The information in database may be such as the position of the robot, the container it carries, pointers to the trajectories produced by the motion planner, the currently used resources etc. The database can also contain predicates, which trigger some internal C code to retrieve their value. Each plan in the library describes a particular sequence of actions and tests that may be performed to achieve given goals or to react to certain situations. PRS does not plan by combining actions, but by choosing among alternative plans/or execution paths in a plan which is executing. Therefore this library must contain all the plans/procedures/scripts needed to perform the tasks for which the robot is intended. The task graph is a dynamic set of tasks currently executing. In a mobile robot application, the task graph could contain the tasks corresponding to various activities the robot is performing (one activity to refine its current mission, another to monitor incoming messages from a central station giving orders, another one managing the communication layer with low-level functional modules, etc).

An interpreter manipulates these components. It receives new events and internal goals, checks sleeping and maintained conditions, selects appropriate plans (procedures) based on these new events, goals and system beliefs, places the selected procedures on the task graph, chooses a current task among the roots of the graph and finally executes one step of the active procedure in the selected task. This can result in a primitive action or the establishment of a new goal. In PRS, goals are descriptions of a desired state associated to the behaviour to reach/test this state. The goal to position the robot is written (**achieve** (position-robot 20 10 45)). This goal is satisfied if the robot is already located at this position. The goal to test, if the robot is located in a particular position, would be written (**test** (position-robot 20 10 45)). Finding out the current location of the robot can be written using variables (test (position-robot $x $y $theta)). This implies that either the database contains this information or there is a procedure to find out the position. The goal to wait until the robot is ready to move could be written as (**wait** (robot-status ready-for-displacement)). A "guarded" action (behaviour) is shown in the next example:

> (& (achieve (position-robot 20 10 45)) (**preserve** (~ (robot-status emergency))))

This insures that robot status is not emergency during movement to the position (20,10,45). The goal to maintain the battery level above 20% while executing a trajectory is written:

> (& (achieve (ececute-trajectry @traj)) (**maintain** (battery-level 0.200000)))

An example of a plan for long-range displacement with PRS is shown in Figure 7.

This language is able to run multiple tasks in real time. It can condense a lot of information on a single command-line, but its readability is poor.

```
(defka |Long Range Displacement|
:invocation (achieve (position-robot $x $y $theta))
:context (and (test (position-robot @current-x @current-y @current-theta))
                    (test (long-range-displacement $x $y $theta @current-x
                    @current-y  @current-theta)))
:body ((achieve (notify all-subsystems displacement))
 (wait (V (robot-status ready-for-displacement) (elapsed-time (time) 60)))
 (if (test (robot-status ready-for-displacement))
   (while (test (long-range-displacement $x $y $theta @current-x @current-y
                    @current-theta))
   (achieve (analyze-terrain))
   (achieve (find-subgoal $x $y $theta @sub-x @sub-y @sub-theta)
   (achieve (find-trajectory $x $y $theta @sub-x @sub-y @sub-theta @traj)
   (& (achieve (execute-trajectory @traj))
     (maintain (battery-level 0.200000)))
   (test (position-robot @current-x @current-y @current-theta)))
  (achieve (position-robot @x @y @theta))
 else
 (achieved (failed)))))
```

Figure 7. A Plan for Long Range Displacement with PRS.

2.3.2.8 GOLOG

GOLOG (alGOL in LOGic) is a logic programming language for dynamic domains [Levesque et al., 1997]. The programs in GOLOG can be written at a high level of abstraction and a GOLOG program is actually a macro, which expands during the evaluation of the program to a sentence in the situation calculus. The world model is dynamically maintained according to user-defined axioms about the preconditions and effects of actions and the initial state of the world, and the program reasons about the state of the world and considers the effects of different actions before committing itself to a particular behaviour. The original GOLOG language had several missing features that enabled implementations work only with completely known initial situations without sensing handling and other concurrent processes. To overcome these problems extensions to the GOLOG language have been presented by [Grosskreutz and Lakemeyer, 2000]. With these extensions, cc-Golog and pGOLOG, it is possible to apply this system also for mobile robots. An example of cc-Golog plan for a mobile robot is shown in Figure 8. The expression loop(σ) is a shorthand for while(true, σ). This task makes a robot to "(1) deliver mail to the offices; (2) say 'hello' whenever it passes near Henrik's room; (3) interrupt its actual course of action whenever the battery level drops below 46 Volt and recharge its batteries" [Grosskreutz and Lakemeyer, 2000].

```
withPol(loop(waitFor(battLevel<=46,
  seq(grapWhls,chargeBatteries,releaseWhls)))),
 withPol(loop(waitFor(nearDoor6213,
   seq(say(hello),waitFor(nearDoor6213))))
  withCtrl(wheels,deliverMail)))
```

Figure 8. cc-Golog Plan.

The first line in this plan means that the robot is waiting until the battery level drops to level 46. At this point an atomic action *grabWheels* sets the fluent *wheels* to false and the program *deliverMail* is blocked. When the task *chargeBatteries* is completed the action *releaseWhls* will set the fluent *wheels* to true and *deliverMail* may resume execution. The task *deliverMail* will make use of actions *startGo(x,y), waitFor(atDestination)* and *stop*. Relations and functions whose truth values vary from situation to situation are called fluents in ccGolog.

This language is powerful and it can handle concurrent tasks and events. The readability of the plans in cc-Golog is fairly good, but it is far away from natural language.

2.3.2.9 TDL

TDL (A Task Description Language for Robot Control) is a language, which is an extension of C++ and contains means to create, synchronise and manipulate task trees [Simmons and Apfelbaum, 1998]. It is based on three-tiered architecture, which is common in a modern robot and is shown in Figure 9. The behaviour layer controls the actuators and sensors. The executive layer translates abstract plans to low-level commands, executes them and handles exceptions. The planning layer specifies the mission at abstract level. The basic representation used in TDL is a task tree and a TDL-based control program operates by creating and executing task trees. Each node in a task tree has an action associated with it. The action can perform computations, dynamically add child nodes to the task tree or perform some physical action in the world. Actions can include conditional, iterative and even recursive code.

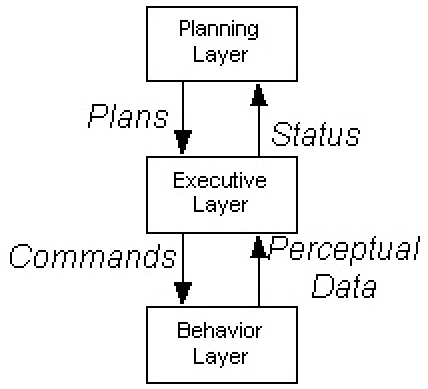
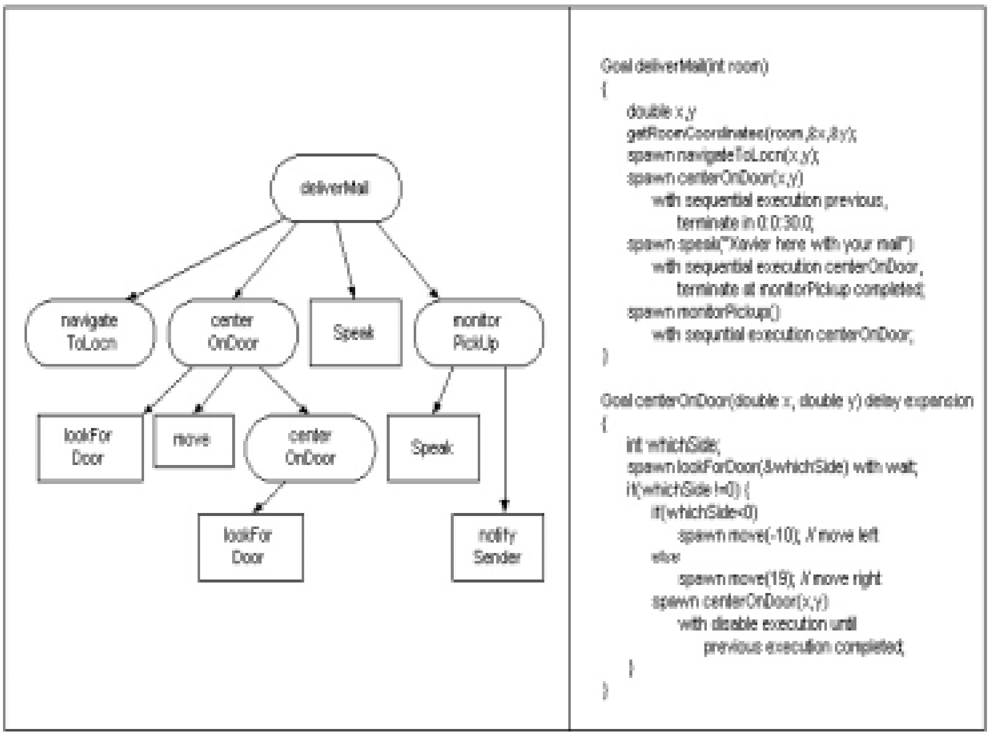


Figure 9. Three-Tiered Control Architecture [Simmons and Apfelbaum, 1998].

The actions associated with nodes use current sensor data to make decisions about what nodes to add to the tree and how to parameterise their actions. Thus, the same task-control program can generate different task trees from run to run. TDL contains a compiler that transforms TDL code into efficient, platform-independent C++ code that invokes a Task Control Management (TCM) library to manage task-control aspects of the robot. The robot action is thought to be formed of a set of concurrent tasks such as moving, sensing, planning and executing etc. TDL gives means to decompose a task, synchronise subtasks, monitor execution and handle exceptions. An example of a task tree with related task definition is shown in Figure 10.

A visual design tool (VDT) is also available for programmers to design TDL tasks.



```
Goal deliverMail(int room)
{
    double x,y
    getRoomCoordinates(room,&x,&y);
    spawn navigateToLocn(x,y);
    spawn centerOnDoor(x,y)
        with sequential execution previous,
            terminate in 0:0:30.0;
    spawn speak("Xavier here with your mail")
        with sequential execution centerOnDoor,
            terminate at monitorPickup completed;
    spawn monitorPickup()
        with sequential execution centerOnDoor;
}

Goal centerOnDoor(double x, double y) delay expansion
{
    int whichSide;
    spawn lookForDoor(&whichSide) with wait;
    if(whichSide !=0) {
        if(whichSide<0)
            spawn move(-10); // move left
        else
            spawn move(10); // move right
        spawn centerOnDoor(x,y)
            with disable execution until
                previous execution completed;
    }
}
```

Figure 10. Example Task Tree and Task Definition [Simmons and Apfelbaum, 1998].

Task definition with TDL looks like and is as readable as conventional C++ software. The real-time use of TDL is limited by the fact that it requires two compilation phases.

In Table 3 the languages are mapped against the requirements of ILMR. The numbers 1–12 stand for requirements and '+' means that the requirement is satisfied and '–' means that it is not. A blank space means that information is not available.

Table 3. Mapping of languages against the requirements of ILMR.

	1	2	3	4	5	6	7	8	9	10	11	12
ARCL	+			-	-	-		-				
ZDRL	+			-	-			-				
MML	+			-	-	-		-				
RIPL	+			-	-	-						
RCL	+			-	-	-						
FDTL	+				-	-						
PRS	+		-	-	-							
GOLOG	+			-	-							
TDL	+			-	-	-						

It can be seen from the table that at least two of the requirements are not satisfied by any of the languages.

2.3.3 Visual languages

2.3.3.1 Lingraphica

Lingraphica is a visual-language prosthesis which was meant to be used for robot command and control in unstructured environments [Leifer et al., 1991]. This method suggests that robot motion and task planning can be specified efficiently and with a minimum of training using text-graphic primitives. This system support communication through access to a Picture Gallery of linguistically and situationally organised graphic objects. The thesaurus is divided into six categories: actors, actions, placements, modifiers, things and other. Motion primitives in this system contain such actions as rotation (about an axis), swinging (a door), turning (a knob), and constrained motion (e.g., inserting an object into a slot) or moving in a straight line.

With over 2000 pictographs even the uninitiated can define a task for a robot. It could, however, be difficult to define the details of the task with pictures or at least it could take a long time to find suitable pictures from the selection.

2.3.3.2 PILOT

PILOT (Programming and Interpreted Language Of actions for Telerobotics) is a visual, imperative and intepreted language for mission programming in telerobotics [Fleureau et al., 1995]. In PILOT the primitives are graphical boxes with symbols that describe their effects. There are seven primitives available and they are *sequentiality*, *conditional*, *iterative*, *parallelism*, *pre-emptive*, *reactivity* and *alternative*. The first one means that the second action is started when the first one is ended. The *conditional* primitive is used to make a choice according to boolean variables or equations. The first item which is true generates the execution of the associated actions. The *iterative* primitive generates a loop while a boolean evaluation is false. The evaluation can be a mathematical comparison or a sensor reading. The primitive *parallelism* starts n actions in parallel, but it terminates only when all the actions are stopped. The *pre-emptive* primitive is used to stop parallelism as soon as one action is finished. With the primitive *reactivity* the mission can be stopped and it waits for the event given by the operator or a sensor. The primitive *alternative* allows the operator to make a choice between actions. In PILOT the following actions are modelled: Move(V), Turn(α), Detect L1, Initialize, Siren, Detec line, Line-guiding(V) and Approach line(α,V). Where V means velocity and α means angle.

This language is derived from the YALTA (Yet Another Language for Telerobotics Application) [Paoletti and Marce, 1990] and is suited to planning missions and modifying them during execution. It allows sequential and concurrent actions execution and response to sensory events. The set of commands is too limited but its readability is good. A mission designed with PILOT is shown in Figure 11.

The Table 4 shows the mapping against the requirements of ILMR. At least two of the requirements are not satisfied.

Table 4. Mapping of languages against the requirements of ILMR.

	1	2	3	4	5	6	7	8	9	10	11	12
Lingraphica	+		-	+	-			-				
PILOT	+				-	-		+				

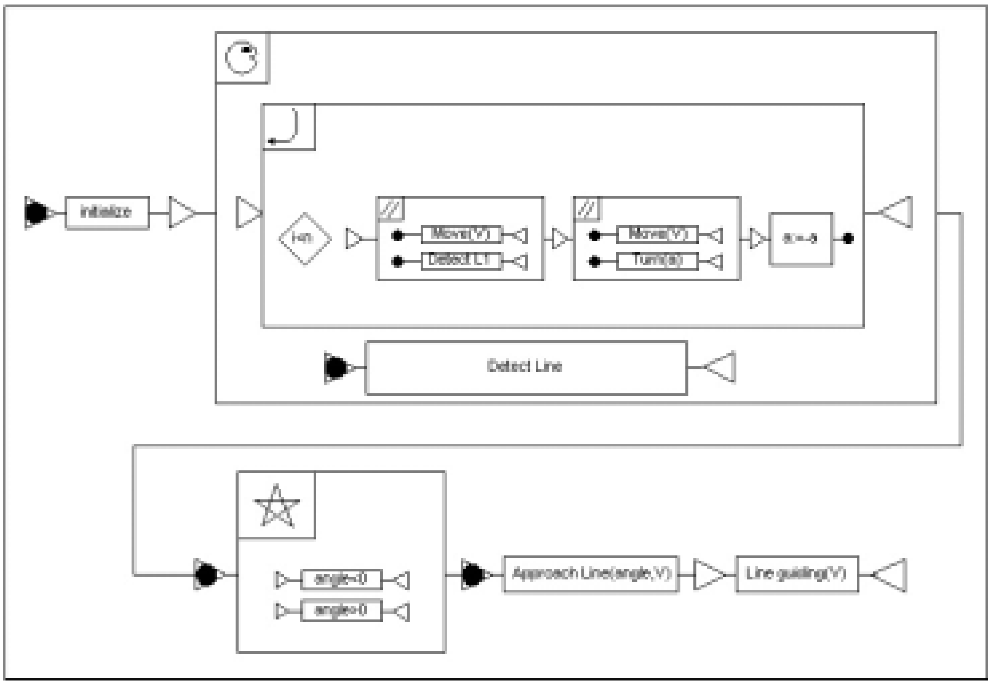


Figure 11. A mission designed with PILOT.

2.3.4 Languages for agents

2.3.4.1 ESL

ESL (Execution Support Language) is a language for encoding execution knowledge in embedded autonomous agents [Gat, 1997]. ESL has features from other languages e.g. RAP and RPL. ESL was meant to be a powerful, flexible and easy-to-use tool and in practise it is an extension of Common Lisp. Contingency handling in ESL is based on the cognizant-failure approach. This approach presumes that multiple possible outcomes of actions are easily categorised as success or failure, and when a failure occurs, the system can respond appropriately. The construct FAIL signals that a failure has occurred and WITH-RECOVERY-PROCEDURES sets up recovery procedures for failures. A Cleanup Procedure is called in ESL with WITH-CLEANUP-PROCEDURE construct and it is called when any other procedure cannot deal with the situation. An example of ESL code is shown in Figure 12. Three different recovery procedures for recovering are given from widget failures. The third procedure will not give up until three retries.

```
(defun recovery-demo-1 ()
     (with-recovery-procedures
      (   (:widget-broken
                (attempt-widget-fix :broken)
                (retry))
          (:widget-broken
                (attempt-widget-fix:severely-broken)
                (retry)
          (:widget-broken :retries 3
                (attempt-widget-fix
      :weird-state)
          (retry)) )
     (operate-widget)))
```

Figure 12. An example of ESL code [Gat, 1997].

ACHIEVE and TO-ACHIEVE constructs can be used to decouple achievement conditions and the methods of achieving those conditions. An event can be used to synchronise multiple concurrent tasks in ESL in such a way that tasks signal each other. A task can wait for several events simultaneously. When any of these events is signalled, the task will become blocked. Multiple tasks can also wait for same event simultaneously. The constructs for synchronisation are WAIT-FOR-EVENTS and SIGNAL. There are also constructs for checkpoints, task nets, guardians and property locks in ESL. The language ESL has been used to build the executive component of a control architecture for an autonomous spacecraft.

This language seems similar to RAP and it supports multiple concurrent tasks. A task can be waited-for or signalled in ESL and ESL provides a mechanism called a checkpoint for signalling task-related events. It will keep a record of the event having happened.

2.3.4.2 MRL

MRL (Multiagent Robot Language) is an executable specification language for multiagent robot control [Nishiyama et al., 1998]. Physical robots and sensors are regarded in MRL as intelligent agents and MRL concentrates on semantic level communication between them. Each robotic agent has it’s own knowledge source for rules and procedures to perform tasks requested by an external agent. MRL contains means for concurrency control, event handling and negotiation of the agents. An agent in MRL can contain several sub-agents but only one super-agent and thus the agent has two unique communication channels, one for super-agent and another for sub-agents. An agent never sends a message to a certain sub-agent, but sub-agents always check whether the message is available. The uppermost agent, called the root agent, manages indirectly all the agents. An example of applying MRL in co-operation between two robots is shown in Figure 13. The key mechanism for co-operation comes from the following message:

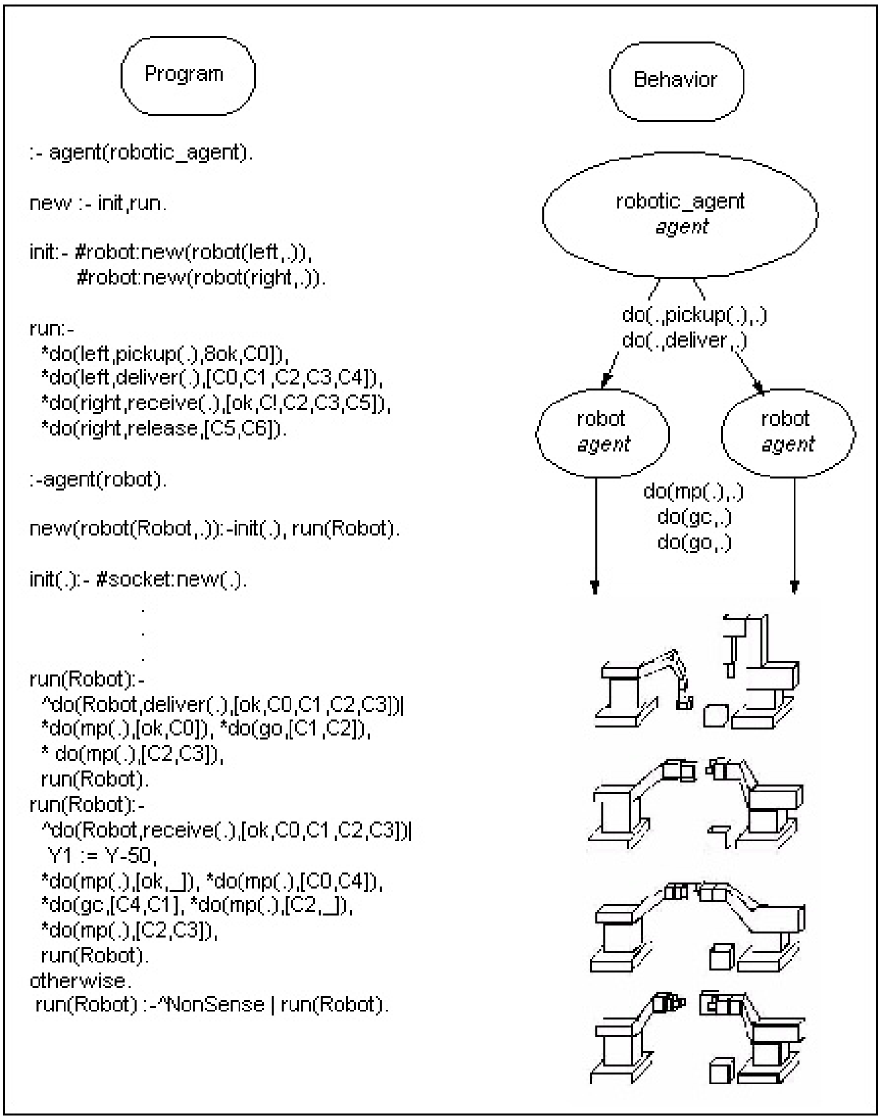
```
do(<AgentID>,<Command>,<A list of variables for
synchronization>)
```

where variables are used for concurrency control between two robots.

The language MRL was tested with a multirobot system, which included four manipulators, a vision sensor and a mobile camera. These objects were connected to several workstations through serial interface. MRL was also applied to mobile robots successfully according to [Nishiyama et al., 1998].

All programs with MRL were first compiled into KL1 language, which is a parallel logic programming language designed in the Japan’s Fifth Generation Computer Systems Project. KL1 programs were then compiled into C programs and these executable programs were running in parallel UNIX systems and even DOS-based computers.

This language enables concurrency control, priority control and negotiation but the readability of the language is poor and it requires three compilation phases until the code is executable in robots.



```
:- agent(robotic_agent).

new :- init,run.

init:- #robot:new(robot(left,.)),
       #robot:new(robot(right,.)).

run:-
  *do(left,pickup(.),8ok,C0]),
  *do(left,deliver(.),[C0,C1,C2,C3,C4]),
  *do(right,receive(.),[ok,C!,C2,C3,C5]),
  *do(right,release,[C5,C6]).

:-agent(robot).

new(robot(Robot,.)):-init(.), run(Robot).

init(.):- #socket:new(.).
                .
                .
                .
run(Robot):-
  ^do(Robot,deliver(.),[ok,C0,C1,C2,C3])|
  *do(mp(.),[ok,C0]), *do(go,[C1,C2]),
  * do(mp(.),[C2,C3]),
  run(Robot).
run(Robot):-
  ^do(Robot,receive(.),[ok,C0,C1,C2,C3])|
   Y1 := Y-50,
  *do(mp(.),[ok,_]), *do(mp(.),[C0,C4]),
  *do(gc,[C4,C1], *do(mp(.),[C2,_]),
  *do(mp(.),[C2,C3]),
  run(Robot).
otherwise.
 run(Robot) :-^NonSense | run(Robot).
```

Figure 13. A co-operative program with two robots by MLR.

The Table 5 shows the mapping against the requirements of ILMR. At least three of the requirements are not satisfied.

Table 5. Mapping of languages against the requirements of ILMR.

	1	2	3	4	5	6	7	8	9	10	11	12
ESL	+		-	-	-							
MRL	+		-	-	-	-						

2.3.5 Natural language control of robots

2.3.5.1 FSTN

In [Drews and Fromm, 1997] a natural-language approach for mobile service robot control has been described. A language-processing unit has been realised and installed in a wheelchair, which is equipped with a navigation and environment control system. For the acoustic word recognition, a commercial software toolkit (Dragon Tools) is used and the sampled word sequence is transmitted via a wireless RS232 channel to an external PC, which is attached to the environment control module. High-priority control for wheelchair like “Stop” are intercepted and processed directly by the wheelchair CPU. The sentence is parsed by a finite-state transducer network (FSTN), which checks the grammatical structure and vocabulary. The output of the FSTN contains a formatted version of the sentence. The database in FSTN contains grammatical word clusters like "noun", “verb”, “negation” etc. In the next phase the formatted sentence is processed by the transformation module. The sentence is split into the basic components “instruction”, “intervention” and “question” and the splitting is based on the recognition of the grammatical word pattern. These patterns are in transformation database and can be adapted by the individual user. The environment database describes all the objects, the possible activities linked to the objects and the individual status information. The semantic database contains the terms describing the individual objects. For example, the term “Switch off” implies that the variables containing speed, light, steering etc. are all set to zero. When processing missing information, like a sentence “No, the other” the system checks the context memory to find missing data. The context memory is based on “last in first out” structure. In case of no answer the question/answer module is initiated and the question is generated. Finally a control sequence is generated and transmitted to the controller. The system had a vocabulary of approximately 500 words and it allowed navigation in home environment. The system worked well with simple sentences, but there were problems when sentence formulation became more complex.

This language is a good example of the solutions that we shall see in the future. With this system even the uninitiated can control a mobile robot in a home environment.

2.3.5.2 IBL

IBL (Instruction Based Learning) is a method to train robots using natural language instructions [Lauria et al., 2001]. IBL uses unconstrained language with a learning robot system. A robot is equipped with a set of primitive sensory-motor procedures such as turn_left or follow_the_road that can be regarded as an execution-level command language. The user's verbal instructions are converted into a new procedure and that procedure becomes a part of the knowledge that the robot can use to learn increasingly complex procedures. With this procedure the robot should be capable of executing increasingly complex tasks. Because errors are always possible in human-machine communication, IBL verifies whether the learned subtask is executable. If it is not, then the user is asked for more information.

IBL was evaluated in a miniature town (170 x 120 cm) with a miniature robot (8 x 8 cm) equipped with a CCD colour TV camera, a TV VHF transmitter and an FM radio and image processing was done in an external PC.

The linguistic and functional data was collected using 24 humans to give instructions to describe six routes from the same starting point to six different destinations. A human operator in another room guided the miniature robot according to given instructions seeing only the image from the robot's camera. The needed primitives were concluded from this data, and when a human specified a nonterminating action, such as keep going, it was classified as move forward until. The number of primitives was 14 and they are shown in Figure 14.

	Count	Primitive procedures
1	308	MOVE FORWARD UNTIL [(past | over | across)<landmark> | [half_way_of | end_of) street] | [after <number><landmark>[left | right] | [roab_bend]
2	183	TAKE THE [<number>] turn [(left | right)] | [(before | after | at) <landmark>]
3	147	<landmark> IS LOCATED [left | right | ahead] | [(at | next_to | left_of | right_of | in_front_of | past | behind | on | opposite | near) <landmark>] | [(half_way_of | end_of | beginning_of | across) street] | [between <landmark> and <landmark>] | [on <number> turning (left | right)]
4	62	GO (before | after | to) <landmark>
5	49	GO ROUND ROUNDABOUT [left | right] | [(after | before | at) <landmark>]
6	42	TAKE THE <number> EXIT [(before | after | at) <landmark>]
7	12	FOLLOW KNOWN ROUTE TO <landmark> UNTIL (before | after | at)<landmark>
8	4	TAKE ROADBEND [left | right]
9	4	STATIONARY TURN [left | right | around] | [at | from <landmark>]
10	2	CROSS ROAD
11	2	TAKE THE ROAD in_front
12	2	GO ROUND <landmark> TO [front | back | left_side | right_side]
13	1	PARK AT <location>
14	1	EXIT [car_park | park]

Figure 14. Primitive navigation procedures collected from route descriptions [Lauria et al., 2001].

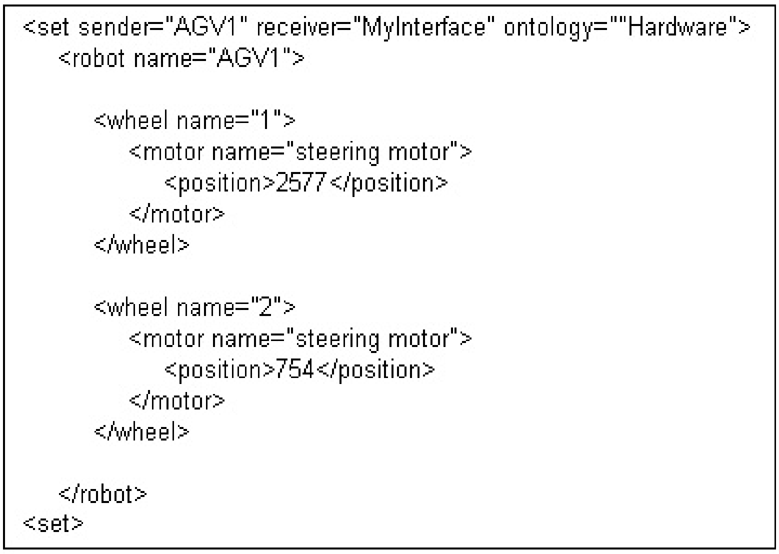
A dialogue manager was designed to act as an interface between the user and the robot manager. The robot manager intercepts new messages from the dialogue manager while executing previous procedures. Three execution processes are available in the robot manager: learn, execute and stop. If there is a corresponding procedure for a given message, then an execute process starts. Otherwise a learn process starts with instructions from the dialogue manager. The robot manager was written using C and the Python scripting language.

These two languages are difficult to map against the requirements of ILMR. They are mainly concentrated on high-level control and information was not available to see how the middle level control was arranged. The basic idea in both languages was, however, akin to the basic idea of ILMR. They both have means to control robots with spoken natural language and they can be used by the uninitiated.

2.3.6 XML-based languages

XML (Extensible Markup Language) is based on SGML (Standard Generalized Markup Language) and SGML is a meta-language that is used to describe markup languages. HTML (Hypertext Markup Language) is one of SGML-applications, that is designed to be a simple tagging language and what is currently used to show text in web-browsers. XML contains the best points of SGML and HTML and it is expected to replace them in the near future. [North and Hermans, 2001]

An XML-based language, RoboML, is designed to serve as common language for robot programming, agent communication and knowledge representation [Makatchev and Tso, 2000]. Elements for agent communication in RoboML are set, get and subscribe, and each can have an optional attribute sender, receiver or ontology. In RoboML the following container elements are defined for Hardware ontology: robot, wheel, motor, sensor, controller, and so forth. These elements can be specified by the name and the ontology attributes. An example RoboML message from the AGV1 embedded agent to the MyInterface user interface agent is shown in Figure 15.



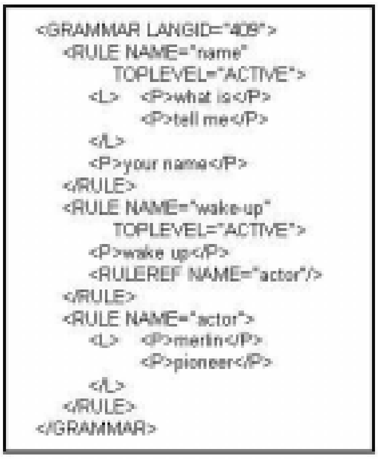
```
<set sender="AGV1" receiver="MyInterface" ontology=""Hardware">
   <robot name="AGV1">

      <wheel name="1">
         <motor name="steering motor">
            <position>2577</position>
         </motor>
      </wheel>

      <wheel name="2">
         <motor name="steering motor">
            <position>754</position>
         </motor>
      </wheel>

   </robot>
<set>
```

Figure 15. RoboML message from AGV1 to the MyInterface.

For the robot programming, RoboML only gives the framework. Available languages should be translated into RoboML.

RoboML is powerful with agent-based human-robot interfaces via the internet. XML is supported by browsers, parsers, translators, browser plug-ins, and other software components for XML are also available.

Another example of XML usage with robot languages is presented in [Kulyukin and Steele, 2002]. They integrated voice-based natural language instruction and action in the three-tiered robot architecture. The user's instructions are first mapped into strings. As the Automatic Speech Recognition (ASR) system freely available Microsoft's Speech Api (SAPI) 5.1 SDK is used. The grammar is defined for recognition by XML Data Type Definition (DTD). Three used rules are shown in Figure 16 and the executive layer is implemented with RAP system.



```
<GRAMMAR LANGID="409">
   <RULE NAME="name"
         TOPLEVEL="ACTIVE">
      <L>   <P>what is</P>
            <P>tell me</P>
      </L>
      <P>your name</P>
   </RULE>
   <RULE NAME="wake-up"
         TOPLEVEL="ACTIVE">
      <P>wake up</P>
      <RULEREF NAME="actor"/>
   </RULE>
   <RULE NAME="actor">
      <L>   <P>merlin</P>
            <P>pioneer</P>
      </L>
   </RULE>
</GRAMMAR>
```

Figure 16. Rules with XML format.

The use of XML seems to be inevitable with mobile robots in the future when communication is done via internet networks. An example of this is the IREDES Initiative which is designed to be the common language in the mining industry. IREDES means International Rock Excavation Data Exchange Standard and it is a standard for electronic data exchange between rock excavation equipment and central computer systems. IREDES is based on XML [WWW-reference 7].

2.3.7 Other related work

An interesting robot-language idea is described in [Yanco, 1992]. It shows how two robots can learn a simple two-signal language using reinforcement learning. Another robot receives auditive signals from humans and both robots can communicate between themselves. A human operator can give two possible tasks to another robot: both spin and both go straight. In a given sample run the desired behaviour was learned after thirteen iterations. Another research of the same kind is shown in [Billard and Hayes, 1997].

Other related works not discussed here are for example Humanoid Motion Description Language [Choi and Chen, 2002], SPLAT (A Simple Provisional Language for Actions and Tasks) which provides a language for defining robot plans and control laws [WWW-reference 8], Supervisory Control of Mobile Robots using Sensory EgoSphere [Kawamura et al., 2001], CURL (Cambridge University Robot Language) the human-oriented robotic programming language [Dallaway and Jackson, 1994] and Famous [WWW-reference 9].

2.4 Summary

Robot languages, considered above, can be classified according to several criteria. They can be on-line or off-line languages, high-level or low-level languages, compiled or interpreted languages, extensions to existing programming languages or own novel designs and so on. Regardless of this classification the development of robot languages has proceeded from simple direct motion commands via more abstract languages towards the use of natural spoken language in robot control.

Some languages are very efficient and can condense a lot of information in a single command. Such languages are for example PRS and GOLOG. It is, however, not clear for a new reader what is meant by their condensed commands, and they are dissimilar to natural language. The intermediate language, ILMR, described in this thesis is not condensed but on the contrary is close to descriptive natural language and is easy to understand and use. In the future the service robots are used also in homes and commands are given with spoken natural language. A high level planner interprets these commands into executable tasks in a robot. This is easier if the robot language is akin to natural language as is the case with ILMR.

Several languages are such that task-giving requires many steps before the code is executable by a robot, and even a slight modification of a task requires that all the steps be done again. With compiled languages a new task should first be compiled and then executed. In worst cases the task should be compiled first to one language and then to another language and finally to executable code.

ILMR is an interpretative language that is most suitable for service and field robots. New tasks can be given or stopped any time without any compilation phase. The executable code for each command is at once available as soon as the interpreter has created and started an object for each command.

These languages described earlier are operational and efficient in their original applications, but their use in different applications requires a lot of work to redesign tasks.

The sequential and concurrent execution control is usually arranged with special commands to create a sequential or a concurrent task. With ILMR no special commands are needed. It is concurrent by nature, and sequential tasks can be arranged using a wait-command or conditional control structures. With movement commands the sequential execution is automatic.

ILMR is not based on any particular existing robot or programming language, but it has features from many languages. The control structures are similar to ones in MATLAB and C, the use of scripts is origin from PRS, the wait-command is similar to the waitfor-command in RAP. ILMR is meant to be an easy link between the user or a higher-level planner and services and resources in a robot and to be close to descriptive natural language.

The starting point for ILMR has been the need to have a simple robot control language, which can be used with several robots and different higher-level planners or other intelligent systems, and also the easy conversion of natural language commands into the robot executable language. The primary need for ILMR was in Workpartner-project.

3 Autonomous service and field robot

3.1 Subsystems of service and field robots

Service and field robots evoke in many of us androids and other human-like robots from science-fiction books and movies. Intelligent and flexible robots that can serve people doing any kind of task, might become a reality in the far future. Today, we are at a stage where service robots are taking their first steps and saying their first words. The current specialized robots can contain several intelligent subsystems, artificial intelligence, vision, and they can even understand spoken language, but we cannot associate intelligence and versatility with today's robots. On the contrary, they are stupid and can clumsily perform only simple tasks that must be designed beforehand and that are restricted to only simple exceptions. They cannot respond in unknown situations and do not know what to do.

Most mobile robots today consist of a set of subsystems. The combination of subsystems is dependent on the application where the robot is used. The set of subsystems, which is discussed in the following, is shown in Table 6. Each subsystem is treated to such an extent that it is later possible to show how the control language can be connected with it.

Table 6. Subsystems of a mobile field and service robot [Halme, 2003].

Power and energy system
Motion system
Motion control system (piloting system)
Navigation system
Perception system
Motion and action planning system
Man-machine interface and remote control system
Work tool system including manipulator

The power and energy system is indispensable in all robots. The energy source can be incorporated into the device or energy can be supplied via a cable. Without any cable the robot is free to move around, but operation time is limited by the capacity of the energy source. An autonomous robot with a built-in energy source must have a supervision system, which guides the robot to the recharging dock when the energy capacity is low.

A service robot can be stationary in applications where movement is not necessary. An example of a stationary service robot could be a robot playing a musical instrument. However, most service robots are mobile, and they need a motion system. Motion can be realised in many ways. Usually it is done with wheels in even and easy environments. In somewhat more difficult environments a track arrangement is useful. A more complicated way is the use of leg mechanisms. Legs are useful in environments that are uneven or not easily accessible. Flying or diving robots use propellers and propulsion motors. Ballast tanks or buoys can also be used in underwater applications. It is also possible to use any combination of means mentioned before to realise motion.

The motion control system, which is also called piloting system, is needed to control the motion of a robot. The piloting system must know the kinematics of a robot in order to control it correctly. The kinematics of a robot with wheels or tracks are simple, but the leg system is much more complicated. In nature, there are e.g. several modes how a four-legged horse moves, and these modes can also be applied to a legged robot that tries to keep its balance.

The navigation system localises the robot in a local or global co-ordinate system, but it is not necessary for a robot. The movement can also be behaviour-based or random. For example in a robot soccer game, the location of a robot might not be relevant information. The whole game could be based on three behaviours: get the ball, avoid another robots, go to the goal with a ball. However, the tactics of the game could be much more efficient with knowledge of the positions of all robots. Usually the task is easier to describe and execute with location information. The navigation system can be based on many different methods and a combination of different methods. Dead reckoning is a usual way to calculate the location of a robot. It means that the position of a robot can be calculated by integrating the speed vector with respect to time. The position error, however, cumulates in this method as a result of slipping of wheels or drift in sensors or many other reasons. Therefore, another method is needed to correct the position error when it becomes too high. This correction method can be based on a beacon system, vision, landmarks and so on.

Some kind of perception system is indispensable in every robot. A robot cannot act properly without sensors. Sensors are needed to sense the environment and the internal state of the robot. Usually robots have at least sensors for location and velocity measurements, actuator state measurements and collision detection. The perception system can combine sensor data from different sensors and preprocess data so that it can be used more easily within other systems.

The motion and action planning system is needed to interpret the intentions of the user and translate them to low-level actuator controls and sensor measurements. This system is the most demanding part of the whole robot. More intelligence here means easier use. If the system is intelligent, the user can lie on a sofa and just say "get me another beer". If the system is less intelligent, the user has to say "Go to the kitchen. Look around and look for the freeze. Go to the freeze and open the door. Look inside and locate the beer. Take the beer and close the door. Come here and give me the beer". Given a still more simple system, the task description has to have more precise details.

Human-Machine Interface (HMI) is ineluctable. Today's robots cannot reason, so they need instructions from the user. The typical tasks for HMI are mission preparation and initialisation, monitoring of tasks and the state of the robot. An important task for HMI is teleoperation. There are many situations which are impossible for a robot to handle. In these cases the user can control the actuators directly via HMI. The virtual reality technology can be used with HMI to enable telepresence and thus make the control of robot easier [Yong et al., 1998].

If the service robot is meant to do something useful, then it needs tools with usable manipulators and a work tool system which controls the manipulator and the tools.

3.2 Control architecture

Robots can be controlled in three ways. Control can be hierarchical, reactive and hybrid. In the hierarchical control the robot updates the internal world model, chooses appropriate action for the situation and executes it. This cycle is repeated constantly. An example of hierarchical control architecture is NASREM [Albus et al., 1987].

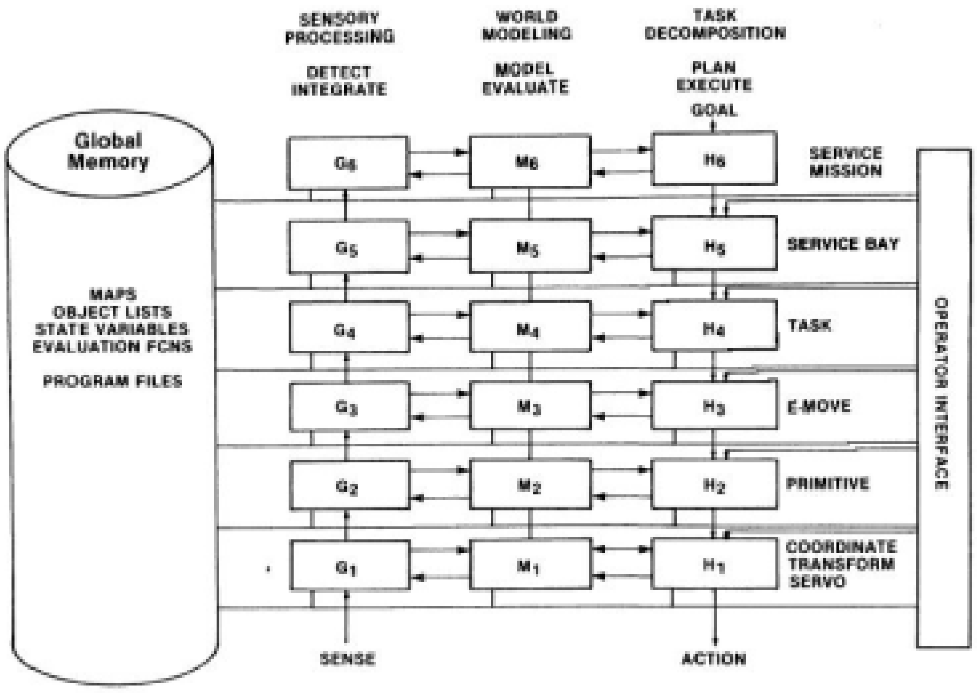


Figure 17. The principle of the NASREM architecture [Albus et al., 1987].

The principle of the NASREM architecture is shown in Figure 17. It is based on three hierarchical lines of computing modules that are serviced by a communications system and a global memory. The sensory processing modules handle the sensory information and observe the external world. The world modeling modules keep the global memory database current and consistent. Global memory is a database that keeps the best estimate of the state of the external world. The task decomposition modules perform real-time planning and task monitoring functions.

The reactive control excludes the internal world model and connects perception and action directly. The creator of a reactive or behavioural approach was prof. Brooks [Brooks, 1986].

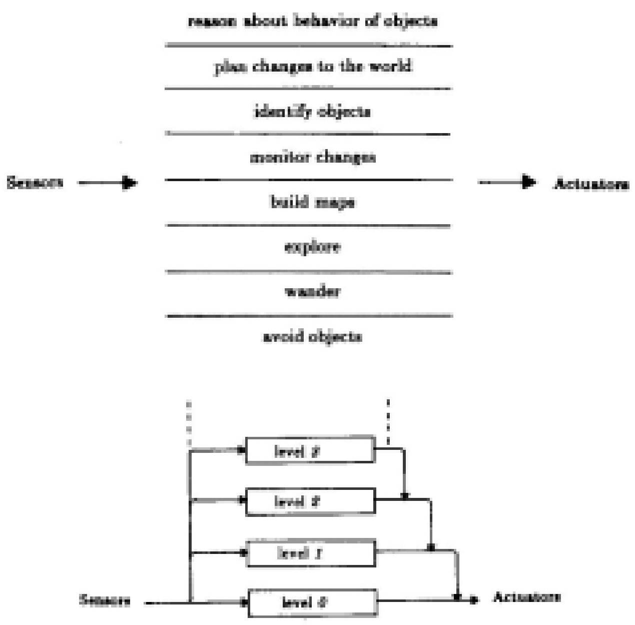


Figure 18. Subsumption architecture of a robot control [Brooks, 1986].

The task is decomposed into task-achieving behaviours in this architecture which is called a subsumption architecture. This is shown at the top of Figure 18. Each slice is implemented explicitly and finally all of them are tied together to form a robot control system.

The control system is built using levels of competence. This is shown at the bottom of Figure 18. First a complete robot control system which achieves level 0 competence is built. When it is working as well as designed it will not be altered any more. This is the zeroth level control system. Next another control layer is built. This first level control system is able to read data from the level 0 system and it is also permitted to inject data to the internal inputs of level 0 suppressing the normal data flow. The zeroth level continues to run unaware of the layer above it. The same process is repeated to achieve higher levels of competence.

There is no need for a central control in behaviour-based architecture. The control system is a set of agents that take care of their own world.

The hybrid control is a combination of both approaches. The three-tiered (3T) control architecture is hybrid. It was explained in chapter two.

The control architecture of ILMR (see Figure 1) has features from both NASREM and subsumption architecture. Behaviours such as obstacle avoidance obey the rules of subsumption architecture. If an obstacle is found, the behaviour suppresses the normal data flow from the position controller to the actuators and replaces it by its own data flow. The task decomposition is, however, hierarchical as it is in NASREM. The task and the goal are both more abstract at higher control levels and the time scale is long at high level and short at low level. ILMR cannot be based only on the subsumption architecture because planning is an important part of it whereas NASREM is too heavy and complicated to be used as it is. That is why the three level architecture has been selected to be the basis for ILMR.

The efficient control of a robot requires that the robot do things concurrently. When it is moving, it should also be sensing. It should also plan and execute concurrently. For example, when a robot approaches an object, it should sense the orientation of the object and plan the final stage of an approach according to the sensed orientation. It should also start to prepare the manipulator and gripper for final manipulation whenever there is enough sensed information available. These concurrent activities need to be scheduled and synchronised in order to co-ordinate activities, and exceptions should be handled as well.

Several languages have been developed to solve these concurrent control problems and some of them have been explained in chapter two.

3.3 Teaching, learning and skills

A simple way to teach a robot a new task is to show how it is done. This teaching can be done for example by teleoperation. The user controls the actuators in the robot directly and the robot records the inputs and outputs of the controllers. Afterwards the robot can repeat the recorded actions when executing a task. The recorded data can be processed after a teaching phase to smoothen the movements of the robot. A taught task can be stored in a database as a new skill of a robot.

Learning in robotics is an intricate question. One extreme view would be that the whole operation of a robot would be the result of continuous learning. One could imagine that in the far future a robot could be built in a way that after construction it knew nothing except basic information and it had the ability to learn. First it learns to handle sensors and actuators, and then it learns skills according to user information and by downloading information from other robots. Another extreme view would be that learning could consist of adding shown and learnt tasks into a task library. In practise, learning is today something between these two extremes.

A skill is an important thing for a robot and it can be for example a set of primitive operations organized in a proper way with respect to the order of execution to reach some abstract or concrete goal. A good control system for a robot is such that the robot can widen its skills.

In Workpartner, which is another test case in this thesis, all intelligent features are located in a planner, which is responsible for the decomposition of the intentions of the user into subtasks and monitors their execution with means of ILMR. The co-operation between the planner and ILMR is handled later in chapter 4.6.

3.4 Control of an autonomous service and field robot

As mentioned before, a robot cannot reason. Thus there must be a system that enables the user to control and command the robot. In this chapter the control of a robot is examined from bottom to top.

3.4.1 Low-level control

An autonomous robot contains a set of actuators and sensors. Each of them contains a driver for detailed operation. A suitable controller can also be included in a driver to entail meaningful actions. For example a speed controller gets a desired speed value as a reference value. It reads the current speed with a sensor and using a suitable algorithm it adjusts the voltage or current for wheel-motors to reach the desired speed.

A low-level control (LLC) of a robot contains drivers and controllers for all sensors and actuators, and it is unique for each different robot. A low-level control contains also an interface which enables upper levels to communicate with it. This interface can be, for example, a set of C-functions, a low-level language or a message-exchanging system.

Teleoperation of a robot could have direct access to low-level controllers, but it could also be arranged through a middle-level control.

3.4.2 Middle-level control

Above a low-level control there is a control level that enables the execution of behaviours, primitive actions and supervision of a robot’s subsystems. This middle-level control (MLC) or execution control is usually implemented using a suitable robot language.

A robot language enables users or AI-systems on higher control levels to use all available resources and services in a robot. Behaviours and tasks can be activated and disabled through a robot language, and task definitions can be given with control structures of a robot language.

This level already contains some autonomous actions, which can be regarded as reflexive functions. They can be compared with reflexes in a human body. They are used, for example, to monitor the balance, energy level and excess of maximum load in a robot and respond with adequate actions when needed.

The type and quality of a language used here has an effect on the usability and versatility of high-level control.

3.4.3 High-level control

The most demanding part of a robot control is the high-level control (HLC). A motion and action planning system in Table 6 can be regarded as a high-level control. The task of a HLC is to decompose the intentions of the user into subtasks and monitor their execution. A HLC can also contain artificial intelligence (AI) which enables independent decisions, and even complicated tasks can be carried out even without any intervention of a human being.

All intelligent features such as learning, reasoning and concluding belong to this level and they form the most important part of intelligent robot control in the future.

An important subsystem to be used with a HLC is a HMI. It enables the user to give instructions or commands to the robot. Commands can be given for example via keyboard, sign language or natural language.

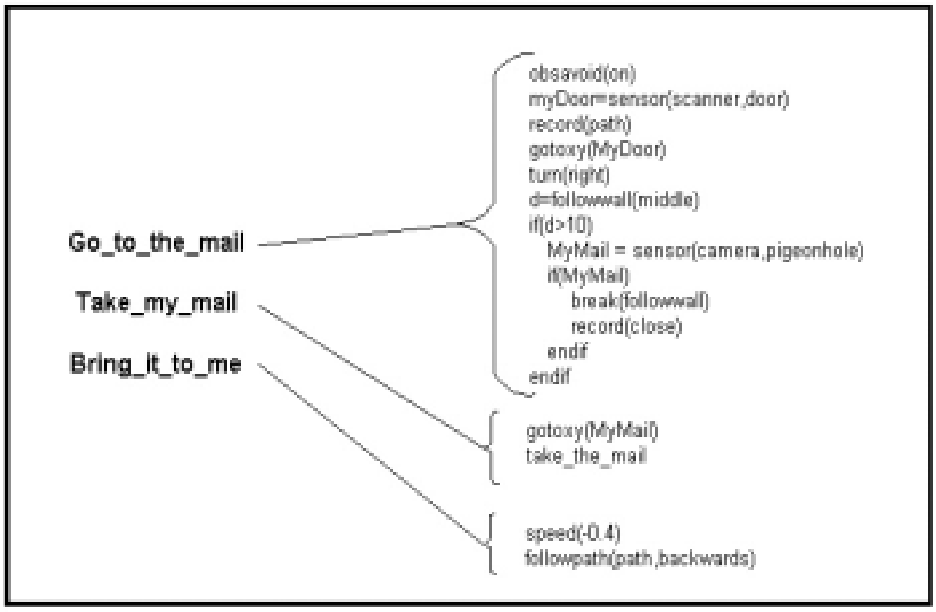
During the last few years the interest of using natural language for training personal robots has increased [Lauria et al., 2002], [Lopes and Teixteira, 2000]. The process of understanding instructions in spoken English can be divided into four subtasks according to [Lauria et al., 2002]. These subtasks are speech recognition, linguistic analysis, ambiguity resolution and dialogue updating. Using a natural language we can express rules and command sequences that even contain symbols. Thus it is well suited to interaction with robots that use symbols and rules to represent knowlegde as well. Using a natural spoken language even the uninitiated can instruct robots to adapt to their particular needs.

A HCL presumes that there is a lower level control in a robot that can execute given subtasks. When a MLC is implemented in a way that is close to natural language, then the co-operation between HLC and MLC is easier.

4 Intermediate language for mobile robots (ILMR)

4.1 Structure of ILMR

ILMR is a way to translate a high-level task description for a mobile robot to low level actuator controls and sensor measurements. Using ILMR the user can easily design a new task for a robot simply by writing a list of English action-describing words. The main principle in development work has been simplicity and ease of use. Neither any deep knowledge of robotics nor good programming skills are required when using ILMR. Using ILMR a simple task is given descriptively as if it would be given to another person. An example of a simple task with ILMR is shown in Figure 19. It contains three commands, which all are in fact subtasks learnt beforehand. The structure of subtasks is shown on the right in Figure 19.



```
obsavoid(on)
myDoor=sensor(scanner,door)
record(path)
gotoxy(MyDoor)
turn(right)
d=followwall(middle)
if(d>10)
   MyMail = sensor(camera,pigeonhole)
   if(MyMail)
      break(followwall)
      record(close)
   endif
endif
```

```
gotoxy(MyMail)
take_the_mail
```

```
speed(-0.4)
followpath(path,backwards)
```

Figure 19. A simple task with ILMR.

When the interpreter in ILMR gets the string **Go_to_the_mail** it searches for a subtask named Go_to_the_mail and if it is found then associated commands are created and executed. No compilation is done, only parameters for commands are examined, and commands are at once ready for execution. The subtask **Take_my_mail** contains also another subtask **take_the_mail**.

Below, in Figure 20, is an example of a more complicated task given in ILMR. It contains also control structures which are described in the following.

```
Init()
Turns = 0
Obsavoid(on)
While(Turns<2)
  MyDoor=sensor(scanner,door)
  Wait(MyDoor)
  Speed(0.3)
  Gotoxy(MyDoor)
  Ok=Turn(right)
  If(Ok)
    Speed(0.6)
    Followwall(middle)
    Door=sensor(scanner,door,left)
    If(Door)
      Break(Followwall)
      Record(ToDoor)
      Speed(0.3)
      Turn(Door)
      Turn(left)
      Goahead(3)
      Over=Takebox
      Wait(Over)
      Record(close)
      Speed(-0.3)
  Followpath(ToDoor,backwards)
  Turnok=Turn(d=-90,c=1)
    Wait(Turnok)
    Speed(0.6)
    Followwall(middle)
    Door=sensor(scanner,door,left,forever)
    If(Door == MyDoor)
      Break(sensor)
      Speed(0.3)
      Turn(Door) // or Turn(MyDoor)
      GoReady=Goahead(3)
      Wait(GoReady)
      Speed(-0.3)
      Turn(d=180,c=1)
      Turns = Turns + 1
    EndIf
  EndIf // Door
EndIf // Ok
EndWhile
Battery=sensor(battery)
If(Battery<LowBattery)
  Priority=Highest
  Go_Charger
EndIf
```

Figure 20. An example task description by ILMR.

In this task a robot is asked to bring two boxes from another room. In the following the action of every command in this example is explained.

Init() initialises the robot and all subsystems and sensors; the task will not continue until Init() has finished.

Turns=0 is a variable statement. A variable named Turns is initiated with a value 0.

ObsAvoid(on) means that the robot's obstacle avoidance behaviour has been switched on.

While(Turns<2) starts a repetition loop, which ends when the variable Turns gets a value 2.

MyDoor=sensor(scanner,door) means that the robot starts to look for a door using a sensor scanner. Both scanner and door have been explained in a model of the robot, which contains parameters for each robot, sensor and object. Scanner in this case means a laser-scanner and a door means an opening, whose width is 0.8–1.2 m. When the door has been found, its parameters will be stored in a variable **MyDoor** and the command sensor will terminate.

Wait(MyDoor) waits until the door has been found. No further commands in current command block are executed before the command wait has terminated.

Speed(0.3) sets the speed value for the robot. Here the value is 0.3 m/s.

Gotoxy(MyDoor) moves the robot to the door which has been found.

Ok=Turn(right) moves the robot to the right following an arc segment of a default circle, which is predetermined by maximum curvature of a robot. The scanner readings are also used to check that a turn is possible. If it is not, then a new, free path to the right is determined according to scanner readings. When the turn has been done, the variable **Ok** gets a value 1, which can be used in the following command.

If(Ok) starts a conditional command block, which is executed when the statement inside the brackets is true. The conditional command block will end with a command **EndIf**.

Speed(0.6) sets a new speed value (0.6 m/s) for the robot.

Followwall(middle) moves the robot along the corridor in the middle.

Door=sensor(scanner,door,left) will start to look for a door on the left side of the robot. This command is executed simultaneously with the previous FollowWall-command. When the door has been found, the command block between the commands **If** and **EndIf** will activate.

If(Door) starts a new conditional command block which will be executed when the variable Door is true (or its value is 1).

Break(Followwall) will terminate the command Followwall.

Record(ToDoor) will start recording the robots position to the file ToDoor.

Speed(0.3) sets again the speed value (0.3 m/s) for the robot.

Turn(Door) moves the robot in the middle of the door opening.

Turn(left) behaves the same way as previously explained **Turn(right)**.

Goahead(3) moves the robot 3 meters ahead.

Over=Takebox is not an elementary command, but it is a subtask learnt beforehand. It is in fact a list of elementary commands or subtasks. When subtask Takebox is over, the variable Over gets a value 1.

Wait(Over) waits until the variable Over gets the value 1.

Record(close) stops position recording.

Speed(-0.3) means that the robot will move to the reversed direction.

Followpath(ToDoor,backwards) moves the robot to the door backwards along the recorded path.

Turnok=Turn(d=-90,c=1) reverses the robot to the right 90 degrees using a value 1.0 for curvature (which is equal to 1/(radius of turning circle)). Here the parameters for command must be precise since there is no scanner pointing backward in the robot. When the turn has been done the variable **Turnok** gets a value 1.

Speed(0.6) sets again the speed value 0.6 m/s for the robot.

Followwall(middle) makes the robot to move in the middle of the corridor.

Door=sensor(scanner,door,left,forever) is the same as before, but the last parameter tells that the command will not terminate when it finds a door. When a door has been detected the command will start to look for another door and the same will continue forever. The parameters of the last found door is available in the variable Door.

If(Door==MyDoor) compares if the door is the same as MyDoor. When the comparison is true then the following command block will activate.

Break(sensor) terminates the sensor-command.

Speed(0.3) sets again the speed value 0.3 m/s for the robot.

Turn(Door) turns the robot to the middle of the door opening. The variable MyDoor could have been used instead of Door.

GoReady=Goahead(3) moves the robot 3 meters ahead.

Wait(GoReady) waits until Goahead(3) is ready.

Speed(-0.3) sets the speed value -0.3 m/s for the robot.

Turn(d=180,c=1) turns the robot to the left 180 degrees.

Turns = Turns + 1 increments the variable Turns.

The following **EndIF and EndWhile** commands close the corresponding command blocks.

Battery=sensor(battery) reads the battery voltage and stores it to the variable Battery.

If(Battery<LowBattery) compares the voltage with predefined value LowBattery, which has been defined in the model description of the robot.

It should be noticed that in this command block level, there is no wait or conditional command that could block the execution of these two commands, and so this comparison is executed continuously.

Priority=Highest means that the following command is executed immediately and all other commands are prohibited.

Go_Charger is a learnt skill, which drives the robot to the charging station regardless of its position and status.

4.2 Control architecture

As it was said in chapter 1.2, ILMR is based on a three-level control architecture (see Figure 1). LLC handles the sensors and actuators and other subsystems of the robot. MLC enables the execution of behaviours, primitive actions and supervision of robot's subsystems. The algorithms of ILMR are running at this level and a robot-dependent intermediate process is used to connect ILMR to a certain robot environment and HLC. The task of a HLC is to decompose the intentions of the user into subtasks and monitor their execution.

4.2.1 Intermediate process

The intermediate process is simply a process that somehow receives commands from somewhere and sends them to the language process. It also transmits sensor and actuator requests from the language process to suitable processes in a robot. Also some simple calculations that are dependent on a certain robot are done as a part of this process. At the moment, there are two different intermediate processes available. One is for VTT's test vehicle MoRo and another is for WorkPartner. The detailed description for these robots is given in the chapter 5.

In WorkPartner the intermediate process is called *pilot*. When the process *pilot* is started, it will immediately start the process *kokis_0*, which is the actual language process. The word kokis comes from the Finnish words KOmentoKIeli Sisäinen, which means in English Internal Commanding Language. The user interacts never directly with *kokis_0*, everything is done through *pilot* and *kokis_0* is in a way invisible to the user. The commands in WorkPartner can be given in process *manager*, in which they can be written directly using the keyboard. The process *manager* sends commands to the pilot with qnx-messages as shown in Figure 21. Other processes can also send commands to the pilot in the same way.

When commands for Workpartner have been given via a graphical user interface, they will be sent first to the manager, which in turn sends them to the pilot.

```
message msg,rmsg;
char* s = “LocMode(automatic)”;

int ptu_pid=qnx_name_locate(0,”pilot”,0,0);
msg.sender=ID_MGR;
msg.type=MSG_PILOT;
for(int i=0;i<strlen(s);i++) msg.data[i] = s[i];msg.data[strlen(s)]=0;
Send(ptu_pid,&msg,&rmsg,sizeof(msg),sizeof(rmsg));
```

Figure 21. An example of code for sending a message to process pilot in Workpartner.

The process *pilot* returns immediately giving information about the success of the command. If the command was correct, then rmsg.data[0]=0, and if there was an error, then rmsg.data[0]=1.

The process pilot also calculates the position of Workpartner-robot using dead-reckoning techniques via equations (1). Dead-reckoning calculation is done at 10 Hz. The navigation subsystem of Workpartner updates the position of the robot at 2 Hz. When a new position infromation is available the dead-reckoning position is also updated.

$$\begin{aligned}
ds &= \tfrac{1}{4}\sum_{i=0}^{3}\left[leg\_new(i) - leg\_old(i)\right] c \\
dh &= \tanh\left(\left[leg\_new(1) - leg\_old(1)\right] c - \left[leg\_new(0) - leg\_old(0)\right] c\right) \\
h &= h + dh \\
x &= x + \cos(h)\, ds \\
y &= y + \sin(h)\, ds
\end{aligned} \tag{1}$$

where *c* is a coefficient to convert pulse counts of wheels (*leg_new and leg_old*) on legs into meters. The robot’s posture is defined by (x,y,h), where h is the heading-angle.

4.2.2 Internal language process

The core of the language process can be divided into three main parts. One is the shared memory, which is used to transmit information between the language and the intermediate processes. The second is the interpreter, which checks the grammar and parameters of a new command and creates a runnable object for each command. The third part is a vector which contains all created command objects and a loop in which the vector is manipulated. Naturally, there are additional commands which all are inherited from the base class command.

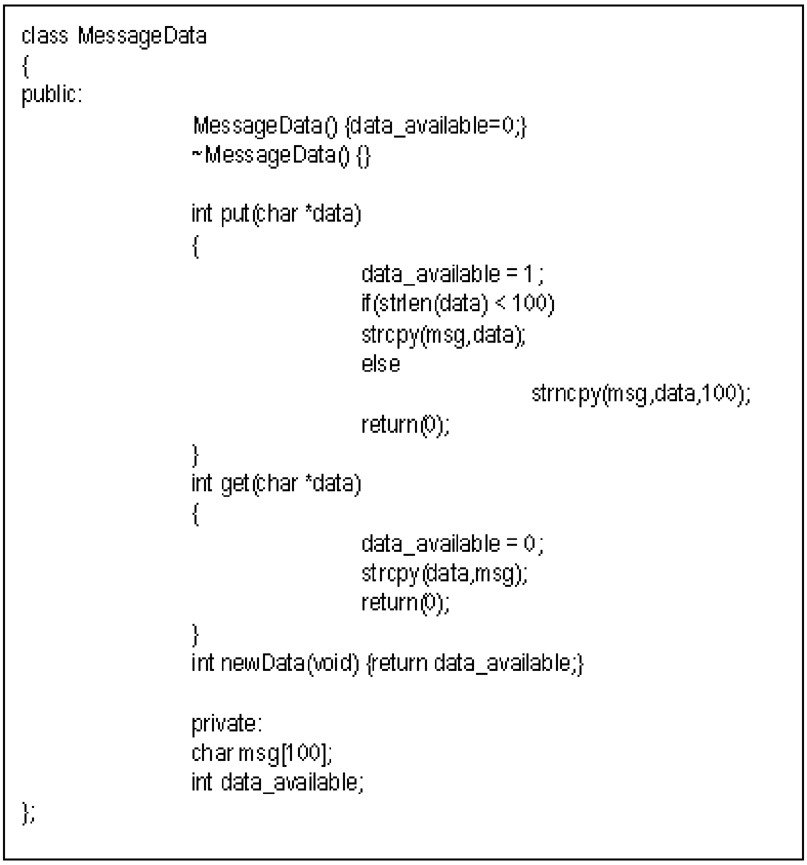
4.2.2.1 Shared memory

The communication between the internal language process and an intermediate process is arranged asynchronously by using shared memory. Shared memory contains fields for sensory data, language, command, command status and message data. Each field is a C++ class that contains information and methods to read and write that information. Figure 22 shows the most important variables for each class and in Figure 23 the C++ source code for the MessageClass is shown. This is the only information needed to transfer between the internal language and an intermediate processes. The communication is asynchronous. When new sensory data is available in the intermediate process, it is written to shared memory. When the internal language process needs a sensory data it checks if that data is available. If it not it will be checked again during the next cycle of the command loop otherwise it is read and used. The command loop is explained in chapter 4.2.2.2. Similarly when the internal language process has data to be transferred to the robot or the user it is written to shared memory and read in the intermediate process and delivered to the right destination. The data flow chart of ILMR is shown in Figure 24.

SensorData in Figures 22 and 24 contains information about the robot.

Class	Most Important MemberVariables
SensorData	x_pos, y_pos, h_pos, speed, distance targ_x, targ_y, targ_a, targ_w ptu_angle, scanner_angle, scanner_dist scans
ILMRData	msgdata[100]
CmdStatus	status,error,msgData[100]
CmdsData	msgData[1000]
MessageData	msg[100]

Figure 22. Member variables for shared memory classes.



```
class MessageData
{
public:
            MessageData() {data_available=0;}
            ~MessageData() {}

            int put(char *data)
            {
                        data_available = 1 ;
                        if(strlen(data) < 100)
                        strcpy(msg,data);
                        else
                                    strncpy(msg,data,100);
                        return(0);
            }
            int get(char *data)
            {
                        data_available = 0 ;
                        strcpy(data,msg);
                        return(0);
            }
            int newData(void) {return data_available;}

            private:
            char msg[100];
            int data_available;
};
```

Figure 23. The class MessageData of ILMR.

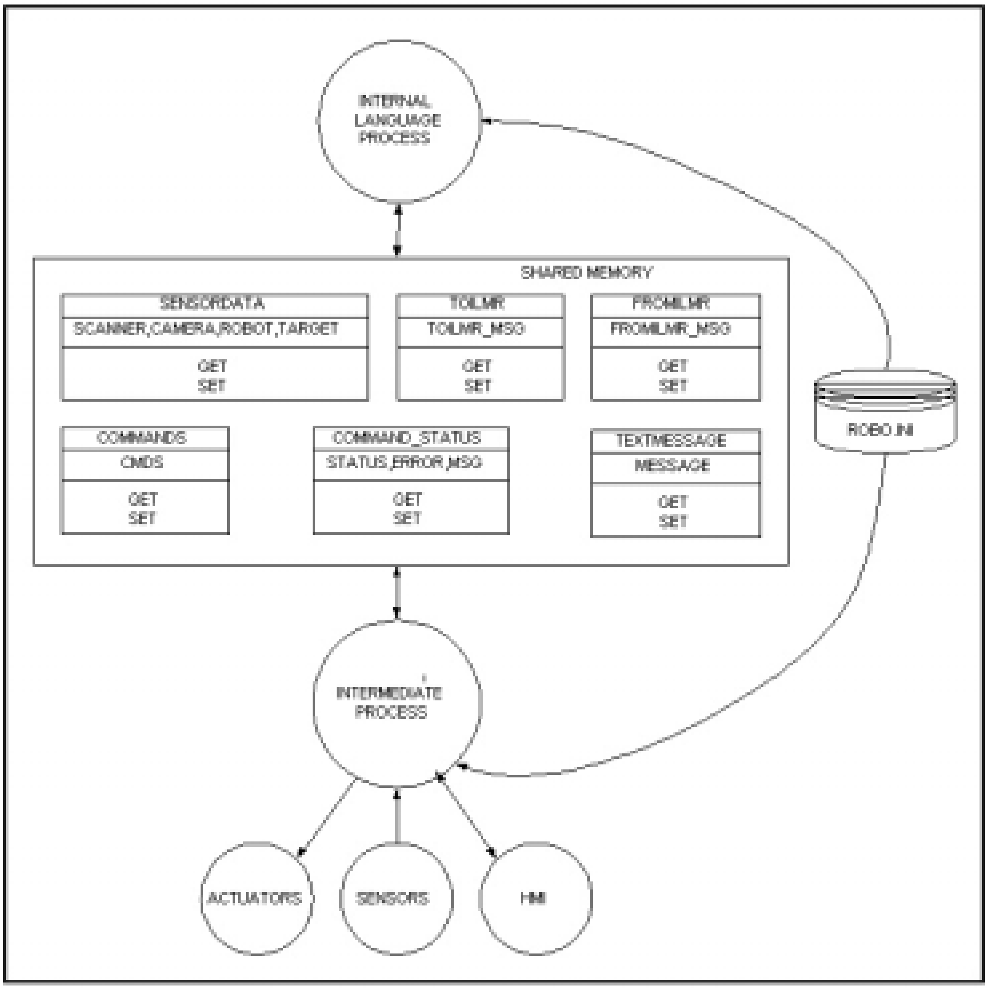


Figure 24. The data flow chart of ILMR.

When the intermediate process has been started, it will first create an object instance of shared memory, and second it will start the language process, which also will be linked to the shared memory object.

4.2.2.2 Interpreter

When the language process is activated it will first read the robot environmental model. Second it will create an interpreter object and the vector for command objects, and finally it will start an everlasting loop. The pseudo-code for this loop is shown in Figure 25. In this loop a new possible command is first checked from the shared memory. If something is found, it will be sent to the interpreter.

When the interpreter receives a string, it first checks if a string contains a variable statement. In case of a new variable is encountered, it will be created, and otherwise the old variable will be used and the given operation will be linked to it. If it was not any variable statement, e.g. Counter = 1, then the interpreter will check if it was a new command. In case of a new command, its parameters will also be checked. If the string will pass these checks, a new command object will be created and appended to the command vector. If the string contains no variable statement or commands, the interpreter will check if it is a subtask learnt beforehand. If a subtask is found, it will be checked and any found commands will be appended to the command vector. If this is not the case, an error message is sent to the intermediate process.

```
While(1)
{
    if(newcommand(cmd)) interpreter(cmd,list);
    Command *cmd;
    for(i=0;i<list->entries;i++)
    {
        cmd = list->find(i);
        status = cmd->run(state);
        if(status==READY) delete(cmd)
    }
    check_robot_status();
}
```

Figure 25. Pseudo-code for the command loop.

The flow chart in Figure 26 shows how a new command is handled in ILMR.

There is no parser available at the moment for ILMR, but it is possible to validate the grammar of the task with the elementary commands check and checkend. If the task to be validated is put in between the words check and checkend, the interpreter will check the grammar and give the error messages, if there are any, but will not create executable objects for commands.

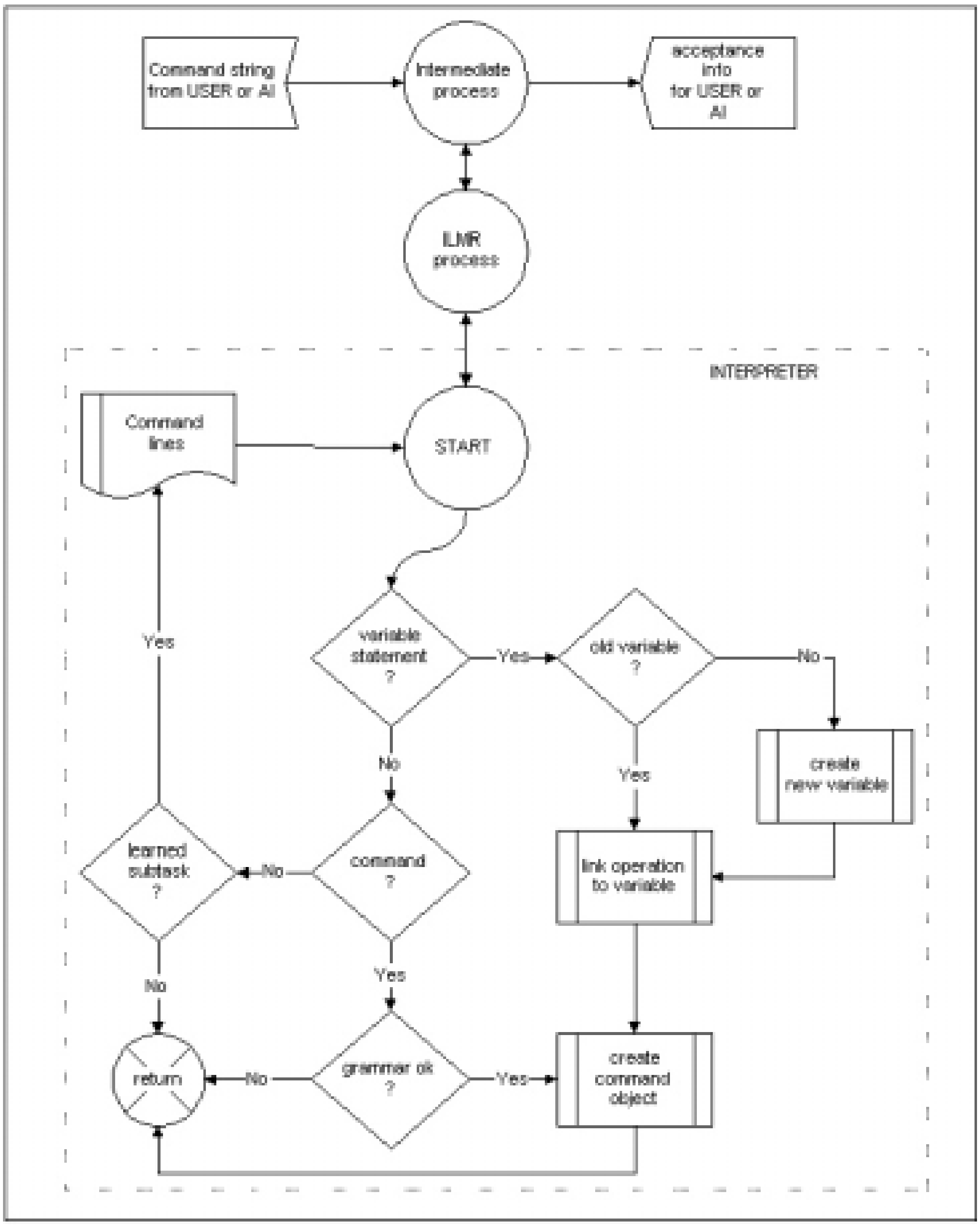


Figure 26. The flow chart of the interpreter.

4.3 Concurrent and sequential execution

The basic element in ILMR is a command block. Command blocks can exist on different levels and command blocks on the same level are executed concurrently.

Usually when commands are given one after another, they belong to same command block level, and are therefore executed concurrently. When a movement-command is running, it is, however, not allowed to run another movement-command and the execution of a movement-command does not begin until the previous movement-command is terminated. Other commands after and before the blocked movement command are however executed. Command blocks on the same level can be distinguished from each other with an empty command and an empty command can be created by giving an empty line to the interpreter.

Command blocks inside each other can be created using if- and while-commands. With a wait-command it is also possible to create a sequential execution flow. The example in Figure 27 contains both sequential and concurrent execution.

```
obsavoid(on)
d=followwall(middle)
cnt=0
while(cnt<5)
        door=findtarget(scanner,door)
        if(door)
                cnt=cnt+1
                claps=0
                while(claps<2)
                        clap_hands
                endwhile
        endif
endwhile
if(cnt==5)
        break(followw)
        break(while(c)
endif
ready=gotoxy(door)
if(d>10)
        shake_head
endif
wait(ready)
make_a_bow

battery=sensor(battery)
if(battery<24)
        goto_charger
endif
```

Figure 27. An example of concurrent and sequential execution in ILMR.

First obstacle avoidance is switched on. It is a behaviour that is executed continuously when the robot is moving. Next the robot starts to follow walls in the middle of the corridor. At the same time it is looking for doors, and if a door is found, the robot will clap its hands twice using a skill learnt beforehand. If the travelled distance is more than 10 meters, then the robot will shake its head once using another skill learnt beforehand. If the door found was the fifth door, then the followwall- and while-commands are terminated, and when the while-command is terminated, then all commands inside the while-block are also terminated. When the command followwall is terminated, it will allow the command gotoxy(door) to run. The last command is blocked with a wait-command, and it is executed as soon as the command gotoxy(door) is terminated. Finally, the robot will make a bow using a skill learnt beforehand. Concurrently the battery level is checked, and if it is below 24 V, then the robot starts a skill goto_charger that has been also learnt beforehand.

Some languages (e.g. GOLOG) contain special commands to create a sequential or a concurrent execution flow. This is however not needed in ILMR. The basic idea is that everything is concurrent except movement commands that are sequential by nature. Sequential execution can be arranged in a natural way using the conditional command “if” or by waiting until the previous command is ready using the wait command.

ILMR contains also a reserved word PRIORITY which is used only when a block needs to be run immediately. Then the first command in that block should be PRIORITY = HIGHEST. If that block is executed, then all other command blocks are terminated.

The obstacle avoidance is arranged as a subsumption behaviour in ILMR. If it is switched on, then the trajectory in front of the robot is checked in view of obstacles any time when an updated environmental map becomes available. This happens at a rate of 2–4 Hz. If an obstacle is found, then the output from any other movement command is suppressed and the obstacle avoidance algorithm controls the movements of the robot. The executable movement command does not know about obstacles and its execution continues unchanged as before encountering obstacles. It uses the current position of the robot and tries to move the robot back to the trajectory. If the obstacle disappears, then the current movement command drives the robot back to the trajectory after a security wait.

4.4 Elementary commands

In ILMR there is a set of elementary commands that can be used directly to control a robot or to construct new, more intelligent features for a robot. At the moment there are 33 elementary commands available in ILMR and the set of used elementary commands is shown in Table 7. In addition to this set of commands, there are also means of handling variables in ILMR.

Table 7. The set of elementary commands in ILMR.

Define	***Movement***	***Action***	***Control flow***
MoveMode	Goahead	Sensor	If – EndIf
LocMode	Turn	FindTarget	While – EndWhile
Speed	FollowPath	Actuator	Break
BodyPose	FollowWall	Manipulate	Stop
Locate	FollowTarget	Init	Cont
AdvanceStop	GotoXY		Wait
ObsAvoid	GotorelXY		Check
Record	Approach		Checkend
Use			Clear
Savepos			
CreatePath			

It is desirable that the set of commands remain as condensed as possible. Therefore, the chosen commands are such that with them all kinds of movements are possible and all features of the used robots can be used. These commands can also be used to build new commands for a robot as a new skill.

It is possible that in the future there will arise needs for new elementary commands from different robot applications.

All of the elementary commands are inherited from the base class *Command* and the definition for that class is shown in Figure 28.

Each of elementary command classes inherits all the methods shown in Figure 28. The four virtual methods are overridden with suitable methods in each command class. As an example the definition for the class *GotoXY* is shown in Figure 29. The number and type of parameters for the constructor-method in each elementary command class can be individual, but the reset- and run-methods have always only one integer-parameter so that calling them is similar in all cases.

The original idea was that a thread would be created for each command so that they could be run concurrently. Unfortunately the threads were not available in the QNX-operating system used, and so another solution was used.

```
#include "cmdid.hpp"

class Command
{
	public:
		virtual Command(){};
		virtual ~Command(){};
		int virtual reset(int)=0;
		float virtual run( int )=0;
		int isMovement() {return movement_cmd;}
		String cmdLine;
		float getRet() {return ret;}
		short getCmdID() {return cmdID;}

	protected:
		int movement_cmd;
		short cmdID;
		float ret; // return value
		int done;
};
```

Figure 28. The base class Command *in ILMR.*

All the created commands will be appended to the command vector and their run-method is called repeatedly in a loop as shown in Figure 25. To ensure a quick response to signals and other exceptions the structure of run-methods must obey certain rules. The basic structure for a run-method is shown in Figure 30. Usually when the command is ready, its object will be destroyed, but in some cases it will be left to wait for further use. That is the reason for the first line (`if(done) return EPSILON;`) where EPSILON is a positive number near zero. Then the current time is checked, and if more than 100 ms has been elapsed, then the calculation phase is done. The calculation phase cannot be time-consuming. If it is, then it shoud be divided into several subphases that are executed during successive run-method calls and in that case the time period 100 ms can be shorter. If the command is ready, it will return –1, and the command object will be destroyed.

The run-method of each command is called at 100 Hz, and for that reason their execution can be viewed as concurrent in practice. The interpreter creates and starts a new command in 2 ms and so a new command given during the execution of other commands will not disturb the concurrent execution of commands.

```
#include <fstream>
#include <time.h>
#include "io.hpp"
#include "command.hpp"
#include "variable.hpp"
#include "traj.hpp"
#include "posc.hpp"

#ifndef GOTOXY_HPP
#define GOTOXY_HPP

class GotoXY : public Command
{
 public:
  GotoXY( IO*, Variable*, Variable*, float, float, float, int, String
);
  virtual ~GotoXY();
  virtual int reset(int);
  virtual float run( int );
  float new_run(float,float,float);

 private:
  IO*        myIO;
  Traj*      trajec;
  Posc*      positionController;
  Variable* my1Variable;
  Variable* my2Variable;
  int        useMyVariable, updateMyVariable;
  int        point, reverse, reverse_going, reverse_ok, first_xy;
  float      start_x, start_y, start_h, end_x, end_y, end_h,
             rev_h, cur_h, t_rev, cur, tn, to, _left;
};

#endif
```

Figure 29. The class GotoXY *in ILMR.*

If the command is a movement-command and the state is RESERVED, it means that there is already a movement-command running and it is not allowed to run another movement-command.

The method *reset* is used to initialize the run-method to be as it was after creation.

```
float SomeCommand :: run( int state )
{
  if(done) return EPSILON;

  tn=clock()/(float)CLOCKS_PER_SEC;
  if(tn-to<0.1) return _left;
  to=tn;

  if(first)
  {
   if(movement_cmd && state!=IDLE) return 100;
   state = RESERVED;
   first = 0;
   // initialization
  }

  // calculations
  if(io->shamsg->sensor->newSickdata() )
  {
    …
    _left = …
  }
  if(_left<=0)
  {
    done = 1;
    state = IDLE;
    this->ret = _left;
    return –1;
  }
  return _left;
}
```

Figure 30. The structure of a run-method.

4.5 Command categories

The elementary commands can be divided into four categories. The first category ***Define*** contains commands that can be used to switch behaviours on or off and set a mode for a subsystem in a robot. If a robot is stationary and some of define-commands are given, the robot will do nothing. The effect of any define-command will be shown only when a robot is moving. The only exception is the command BodyPose, which controls immediately the posture of the robot. The second category ***Movement*** contains commands which actually move the robot. The third category ***Action*** contains commands that handle sensors, actuators and other equipment of the robot, and the fourth category ***Control flow*** contains commands that are used to control the execution of other commands.

Only two of all commands obey the subsumption architecture. ***ObsAvoid*** and ***AdvanceStop*** suppress the output from other movement commands and replace them with their own data flow. Other commands can be compared to levels E-MOVE and PRIMITIVE in the NASREM architecture. The replanning intervals at these levels are accordingly 200 ms and 30 ms. The commands of ILMR, however, do not obey the structure of NASREM. Only the command ***FollowTarget*** have some features from NASREM since it requests services from sensors all the time.

At the moment, most commands handle the movements of the robot, which are mainly generic for different robots. In the future, the commands for manipulators will be added, but now ILMR has no algorithms for manipulations. It only triggers the sequence of manipulation, which is controlled by an external control process and acknowledges the calling software or the user when the manipulation is done. Anyway, there are means in ILMR to control directly the individual actuators in a manipulator, and it is possible to construct manipulator skills using these commands.

To work properly, ILMR needs a simple model of the robot and its sensors and actuators. The model structure is shown in Figure 31. The *HARDWARE* section has three parts. The part *ROBOT* contains all necessary information about the robot such as dimensions, maximum speed and so on. The part *SENSORS* contains the description for each sensor and the part *ACTUATORS* contains the description for each actuator. The SOFTWARE section has also three parts. The part *OBJECTS* contains the identifiers and definitions for each recognizable object. The part *IDENTIFIERS* contains user defined identifiers which can be used with ILMR commands. The part *PARAMETERS* contains user defined default values for certain behaviours of ILMR.

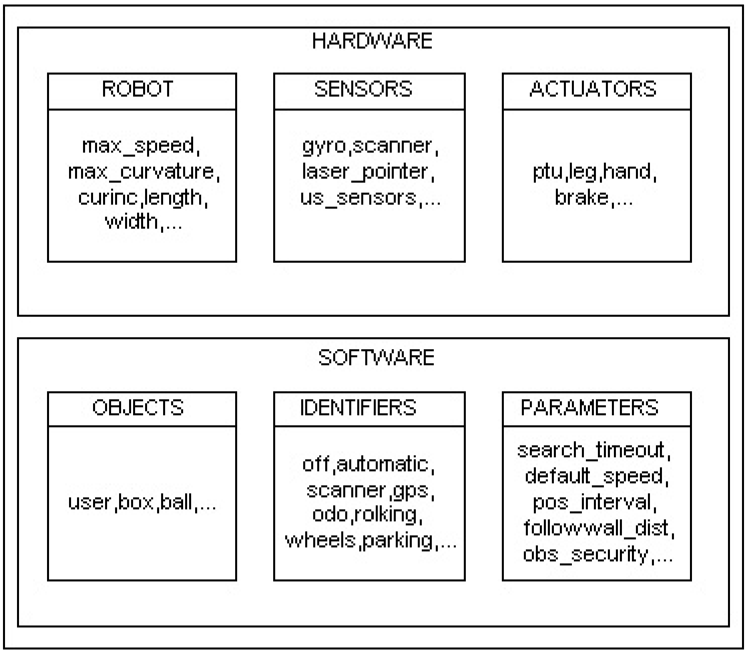

HARDWARE
ROBOT
max_speed, max_curvature, curinc,length, width,...
SENSORS
gyro,scanner, laser_pointer, us_sensors,...
ACTUATORS
ptu,leg,hand, brake,...
SOFTWARE
OBJECTS
user,box,ball,...
IDENTIFIERS
off,automatic, scanner,gps, odo,rolking, wheels,parking,...
PARAMETERS
search_timeout, default_speed, pos_interval, followwall_dist, obs_security,...

Figure 31. The model of the robot and software parameters in ILMR.

This model has critical meaning for ILMR, because it is used in algorithms and it is assumed that the model is sensible.

In the following chapters each command is explained in detail with examples. It should be noticed that commands can be given either in upper-case or lower-case letters or any mixture of them. The commands and parameters are expressed using the BNF-notation (Backus Naur Form).

4.5.1 Define commands

Define commands are useful when a user wants to set a certain mode for a robot or switch on or off some behaviours in the robot. Some of the commands are such that their run-method is called only once, while others are such that their run-method is called repeatedly forever.

MoveMode() is used to select a mode for movement in a robot. The robot Workpartner is used as an example for this command. For another robot the parameters could be as well swim, run and fly. These modes are given in the part identifiers of the robot model.

Movemode	::= **movemode** “(“ mode “)”
mode	::= wheels | rolking | parking

Parameters	wheels	*Workpartner acts as an usual vehicle with wheels*
	rolking	*Workpartner slides its legs along the ground*
	parking	*all joints and actuators are locked*

Example movemode(wheels)

Description This command sends a message to the locomotion process *loco*, which controls the movement of WorkPartner and as a result WorkPartner will change its mode.

LocMode() is used to select a navigation mode for the navigation process. These modes are given in the part identifiers of the robot model.

Locmode	::= **locmode** “(“ mode “)”
mode	::= automatic | scanner | gps | odo

Parameters	Automatic	*mode is selected automatically according to situatio*n
	Scanner	*laser-scanner map is used for navigation*
	Gps	*gps-satellite positioning is used*
	Odo	*only odometry is used for navigation*

Example locmode(odo)

Speed() is used to set speed for a robot.

Speed	::= **speed** “(“ value [“,w”] “)”
value	::= number | variable
number	::= <a real number in range – max_speed…max_speed [m/s]>
variable	::= letter {letter | digit}

Parameters	w	*the command is not executed until the current movement command has terminated*

Description	Without a speed-command a robot uses the default speed that is defined in the robot model. This command can be given any time and it will be executed immediately. It must be noticed, however, that a robot will not move using only this command. It only sets the value for the speed to be used in current or next movement command.

Example	speed(-0.5), speed(-0.6,w)

BodyPose() is used to set values for the robot’s orientation.

Bodypose	::= **bodypose** “(“ setting {“,” setting} “)”
setting	::= sel “=” value
sel	::= “r” | “p” | “rc”
value	::= <real number in defined range>

Parameters	r	*roll-angle of the robot (-30 … 30 degrees)*
	p	*pitch-angle of the robot (-30 … 30 degrees)*
	rc	*road clearance in meters (0…1)*

Example	bodypose(r=10,p=20,rc=0.5)

Locate() is used to initialize the coordinate system for desired position and heading values. It can also be used to ask the current posture information.

Locate	::= **locate** “(“ (x-value “,” y-value “,” h-value “) | “x”|”y”|”h”)”
x-value	::= <real number in meters>
y-value	::= <real number in meters>
h-value	::= <real number in degrees (-180 … 180)>

Example	locate(2.5,7.8,180), x=locate(x)
Description	This command sends a new posture value for dead-reckoning procedure in an intermediate process.

AdvanceStop() will affect only during the following movement command and it will stop the robot before the destination.

Advancestop	::= **advancestop** “(“ distance “)”
distance	::= <real number in meters>

Example advancestop(1.0)

ObsAvoid() is used to select a behaviour for a robot when it encounters an obstacle.

Obsavoid	::= **obsavoid** “(“ mode “)”
mode	::= on | off | stop

Parameters	on	*obstacle avoidance is active*
	off	*the robot ignores obstacles*
	stop	*the robot will stop when an obstacle is detected*

Example obsavoid(on)

Definition The algorithm for this command is explained in chapter 4.5.7.

Record() starts to record a robot's position to the file which is given by a parameter. Recording can be stopped at any time by giving a command record(close).

Record	::= **record** “(“ filename | close “)”
Filename	::= letter {letter | digit}

Example record(path_1), record(close).

Use() command is used to tell the robot to use the last goal position as a new start position when starting a new movement command.

Use ::= **use** “(“ abspos | goalpos “)”

Parameters	abspos	*the robot’s current position is used*
	goalpos	*the last goal position is used*

Example use(goalpos)

Definition	With the parameter goalpos the trajectory for the next movement command is not calculated from the current position but from the destination position and orientation of the previous movement command.

SavePos() command is used to save the current position to be used later.

Use	::= **savepos** “(“ name [“,”device] “)”
name	::= letter {letter | digit}
device	::= letter {letter | digit}

Parameters	name	*the name for the stored position*
	device	*the name for the pointing device to be used to show positions for a robot*

Example	savepos(pos1), savepos(pos2,laser)
Definition	With only one parameter the robot's current position is stored. The other parameter is used to select device for which the position is stored. It can be e.g. a laser that is defined in the robot model. When the parameter laser is used, the position of the place where laser beam hits is stored.

CreatePath() command is used to calculate a path that follows given co-ordinates.

Use	::= **createpath** “(“ name {“,”x-value “,”y-value} “)”
name	::= letter {letter | digit}
x-value	::= <real number in meters>
y-value	::= <real number in meters>

Parameters	name	*the name for the stored path*

Example	createpath(path1,0.0,0.0,5.0,1.0,6.0,5.0)

4.5.2 Path calculation and position controller

When a movement command is given, the object it created is put in the command vector and its run-method is called repeatedly. The language has a state that tells if a movement command is running or not. If this state is idle, then the first call will change it to reserved, which means that no other movement command is allowed to run. If the state is reserved, then the run method only waits until the state changes to idle. When a movement command is ready, in other words the robot has reached the goal or the command is interrupted, then the language state is changed to idle again.

4.5.2.1 Reference path calculation

When a movement command is allowed to run, it will first calculate a reference path to the destination. [Segovia et al., 1991] state that the path for a robot in a free environment is usually presented by four methods. The first one uses straight lines and circular arcs. Another method is to use a curve named "Clothoid" (Cornu Spiral) with linear variation according to the curvilinear abscissa. This type of curve gives a continous curvature path but it is difficult to connect two straight lines with an arc of clothoid. The third way is to use polar polynomials and the fourth way is to use bezier's polynomials.

The path calculation for displacement commands in ILMR is simply done using bezier curves with equations (2).

$$
\begin{aligned}
x &= (1-t)^3 x_0 + 3t(1-t)^2 x_1 + 3t^2(1-t)x_2 + t^3 x_3 \\
y &= (1-t)^3 y_0 + 3t(1-t)^2 y_1 + 3t^2(1-t)y_2 + t^3 y_3
\end{aligned}
\tag{2}
$$

where t is time and x_i and y_i are defined in equations (3), where $(x_{start}, y_{start}, h_{start})$ is the start point of the path and $(x_{end}, y_{end}, h_{end})$ is the end point of the path and *trajdist* is the distance between those two points.

$$
\begin{aligned}
x_0 &= x_{start} \\
y_0 &= y_{start} \\
x_1 &= x_{start} + 0.4\, trajdist \cos(h_{start}) \\
y_1 &= y_{start} + 0.4\, trajdist \sin(h_{start}) \\
x_2 &= x_{end} - 0.4\, trajdist \cos(h_{end}) \\
y_2 &= y_{end} - 0.4\, trajdist \sin(h_{end}) \\
x_3 &= x_{end} \\
y_3 &= y_{end}
\end{aligned}
\tag{3}
$$

Finally the maximum curvature for the created path is calculated, and if it is more than the maximum curvature for the robot, then if allowed by the user, the robot moves backwards and turns to the destination. A new path is calculated and its maximum curvature is calculated again, and if it's not admissible, then the robot continues turning until an acceptable path is generated.

4.5.2.2 Position controller

In [Chung et al., 2001] a position controller for a differential drive wheeled mobile robot is introduced. This controller has two separated feedback loops, a velocity feedback loop and a position feedback loop. The proposed algorithm is also designed to compensate for both internal and external errors. This controller has been successfully tested by computer simulation, but the problem is the complexity of the kinematics and the control algorithm required.

In [Lakehal et al., 1995] a fuzzy based path tracking control for a mobile robot is proposed. The inputs of the controller are the position and orientation errors of the robot with respect to the reference path. In this method no mathematical model of the robot is required, and control rules are simple and clear. Human experience is, however, needed to establish the rules.

In [Feng et al., 1994] a model-reference adaptive motion controller for a differential drive mobile robot is described. This controller modifies control parameters in real time to compensate for motion errors. The internal errors are detected by wheel encoders and compensated by a cross-coupling control method. The external errors can be caused by wheel slippage, floor roughness, etc, and are compensated by an adaptive controller. The adaptation algorithm is based on the hyperstability theory to provide good convergence characteristics and stability.

When a path is successfully created in ILMR then a position controller is created and its run-method is called repeatedly. The used position controller has been developed in a way that it can be used with many different kinds of robots. If the curvature in a robot is marked with c, Δc is the maximum rate for c to change and Δt is a period between two cycles of calculations, then at any moment three possible curvatures for a robot can be calculated using equations (4) where c_{now} means the current curvature of the robot.

$$\begin{aligned} c_1 &= c_{now} \\ c_2 &= c_{now} + \Delta c \Delta t \\ c_3 &= c_{now} - \Delta c \Delta t \end{aligned} \tag{4}$$

With these curvature, current position and heading values we can easily calculate where the robot will be after a certain time. Then we can calculate the cumulative position error in regard to a desired trajectory according to equations (5), where *dist* calculates the minimum distance from the desired trajectory points to the calculated trajectory. Index k is the index of the current reference point and n is the number of points inside a certain distance in front of the robot.

$$\begin{aligned} err\_curr &= \sum_{j=k}^{k+n} dist[c_1, x_{ref}(j), y_{ref}(j)] \\ err\_left &= \sum_{j=k}^{k+n} dist[c_2, x_{ref}(j), y_{ref}(j)] \\ err\_right &= \sum_{j=k}^{k+n} dist[c_3, x_{ref}(j), y_{ref}(j)] \end{aligned} \tag{5}$$

Figure 32 shows three possible trajectories when c_1, c_2 and c_3 are used. The desired path is given with little squares and the robots trajectory is given with a line and little dots.

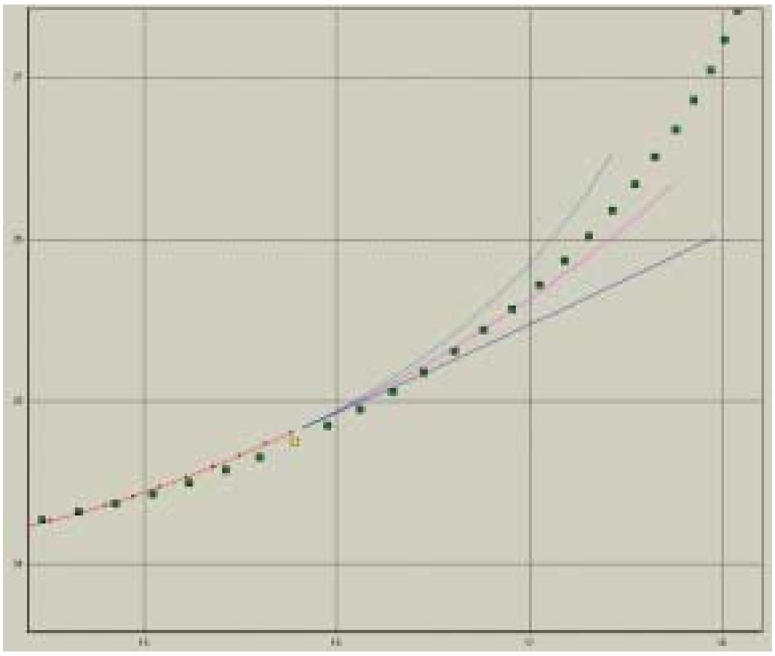


Figure 32. Curvature calculation in the position controller.

When the three error values have been calculated, then the final control curvature can also be calculated according to equations (6a, 6b) in case of err_curr < err_left and err_curr < err_right.

if err_left < err_right

$$c = c_{now} + 0.5\Delta c\Delta t\left(\frac{err\_curr}{err\_curr + err\_left}\right) \tag{6a}$$

if err_right < err_left

$$c = c_{now} - 0.5\Delta c\Delta t\left(\frac{err\_curr}{err\_curr + err\_left}\right) \tag{6b}$$

If err_left is minimum then

$$c = c_{now} + \Delta c\Delta t\left(1 - 0.5\frac{err\_left}{err\_curr + err\_left}\right) \tag{7}$$

and if err_right is minimum then

$$c = c_{now} - \Delta c \Delta t \left(1 - 0.5 \frac{err\_right}{err\_curr + err\_right} \right) \qquad (8)$$

In Figure 32 the checked distance in front of the robot is about two meters. It depends on Δc and the speed of the robot and the curvature difference between the robot and the reference trajectory in front of the robot.

4.5.3 Movement commands

Goahead() moves a robot ahead or back.

Goahead	::= **goahead** "(" value ")"
value	::= number | variable
number	::= <a real number in meters>
variable	::= letter {letter | digit}

Example goahead(-1.45), goahead(dist1)

Turn() command is used to make a robot turn to the left or right.

Turn	::= turn "(" exact | abstract ")"
exact	::= "d=" value ",c=" value
abstract	::= left | right | variable
value	::= <real number in degrees>
variable	::= letter {letter | digit}

Parameters	d=???	*turning angle in degrees*
	c=???	*turning curvature (1/m)*
	left	*turns to the left; if not possible tries to followwall until turning is possible*
	right	*turns to the right; if not possible tries to followwall until turning ispossible*
	variable	*turns to the position stored in the variable*

Example	turn(d=90,c=0.2), turn(left), turn(todoor)
Description	When turn(left) or turn(right) is given, the space for turning is first checked. If there is space, the trajectory is generated and executed. If there is not enough space, then the current distance to the desired direction is maintained and followwall is activated. The followwall continues until turning is possible.

FollowPath() makes a robot to follow a given path.

Followpath	::= **followpath** “(“ filename [,reverse] [,backwards] “)”
filename	::= letter {letter | digit}

Parameters	reverse	*the robot goes backwards*
	Backwards	*the path is followwed from the end to the beginning*

Example	followpath(path_1), followpath(path2,reverse,backwards)
Note	The parameter reverse can be replaced by a speed-command with a negative parameter. The effect is similar.

FollowWall() moves a robot along a wall nearby.

Followwall	::= followwall “(“ mode [“,” distance] “)”
mode	::= left | right | middle
distance	::= <positive real number in meters>

Parameters	left	*the robot follows the wall on the left side*
	right	*the robot follows the wall on the right side*
	middle	*the robot goes in the middle of two walls*
	distance	*sets the distance to the wall. Default distance is defined in the robot model*

Example	followwall(middle), followwall(left,2.5)

Description	The environmental map is used to calculate the angle of the wall on the left or right. The position of the robot is projected to the wall. From this point three meters ahead a new point is calculated, and this point is used to calculate the goal position for the robot at desired distance from the wall. Then a command gotoxy is generated and executed internally, and this is repeated after each two meters until the command is terminated.

FollowTarget() makes a robot to follow a target.

Followtarget	::= **followtarget** "(" target "," distance ")"
target	::= letter { letter | digit }
distance	::= <positive real number>

Parameters	target	an object which is defined in the robot model
	distance	distance to the object in meters

Example	followtarget(user,5)
Description	This command uses two different sensors. A colour camera is used to locate the target that is defined by colour and size. The angle of the found colour-object is used to determine the distance of that object with a laser-scanner. The size of the object is also calculated from the scanner readings. If the size and angle are acceptable, then this target position is set as goal position of the robot. The algorithm is shown in Figure 33.

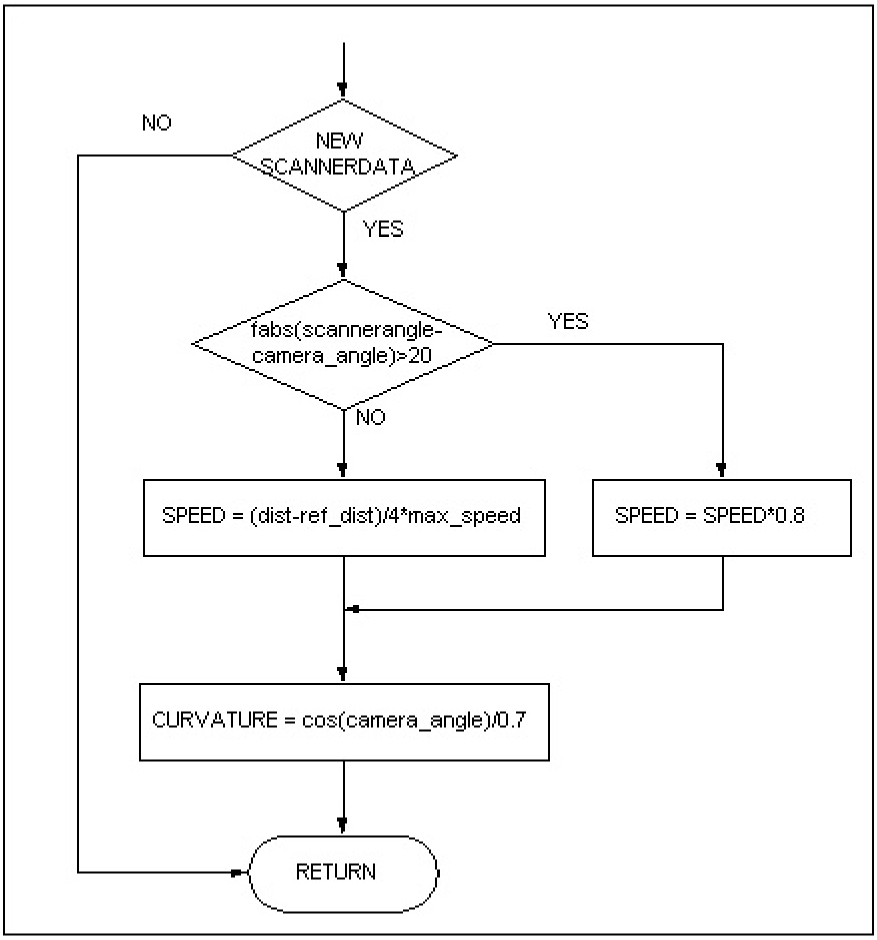

NEW SCANNERDATA
NO
YES
fabs(scannerangle-camera_angle)>20
YES
NO
SPEED = (dist-ref_dist)/4*max_speed
SPEED = SPEED*0.8
CURVATURE = cos(camera_angle)/0.7
RETURN

Figure 33. Algorithm for the command followtarget.

Gotoxy() moves a robot to a desired position.

Gotoxy	::= **gotoxy** "(" x-value "," y-value [",” h-value] [",r"] | variable ")"
x-value	::= <real number in meters>
y-value	::= <real number in meters>
h-value	::= <real number in degrees>
variable	::= letter {letter | digit}

Parameters	r	*with r-parameter the robot is allowed to move first backwards and turn to the destination if it can't follow the internally defined path*

Example gotoxy(3,4,180), gotoxy(myDoor), gotoxy(myDoor,r)

Gotorelxy() is equal to gotoxy(), the only difference is that it uses relative coordinates.

Gotorelxy	::= **gotorelxy** "(" x-value "," y-value ["," h-value] [",r"] | variable ")"
x-value	::= <real number in meters>
y-value	::= <real number in meters>
h-value	::= <real number in degrees>
variable	::= letter {letter | digit}

Parameters	r	*with r-parameter the robot is allowed to move first backwards and turn to the destination if it can't follow the internally defined path*

Example gotorelxy(3,4,180)

4.5.4 Action commands

With action commands the user can initialise some or all actuators, sensors and processes. He can also use sensors and actuators in a simple direct way or in a more intelligent fashion. Both of which are described in detail in the treatment that follows.

Sensor() is a command which can be used in a simple way to take sensor readings from a named sensor.

Sensor	::= **sensor** "(" sensorID ["," define1] ["," define2] ["," forever] ")"
sensorID	::= name | number
name	::= letter {letter | digit }
number	::= <positive integer>
define1	::= direction | integer | variable

define2	::= direction	
direction	::= left | right	

Parameters	sensorID	*Name or number of the sensor. Defined in the robot model*

Examples
```
dist = sensor(scanner) // returns all scanner readings
dist = sensor(scanner,left) // returns the scanner readings on the left
dist = sensor(scanner,1) // returns the scanner reading with index 1
dist = sensor(1,0) // returns the scanner reading with index 0
Mydoor = sensor(scanner,door) // looks for a door using scanner
```

Description In the last example the robot starts to look for a door using a sensor named scanner. Both scanner and door are explained in the robot model. Originally the first field of the variable Mydoor has the value false, and when the door is found, its value is changed to true, and the coordinates of the door are stored to the variable Mydoor. After that the command sensor will terminate. It is also possible to add the parameter left or right after the word door. In that case, the robot will start to look for the door only on the desired side of the robot. It is also possible to add a parameter forever which tells that the command will not terminate when it finds a door. In this case, the variable Mydoor retains the parameters of the latest door found.

FindTarget() is used to locate an object using a camera or a laser scanner. Its course of action is the same as it is with the sensor-command.

Findtarget	::= **findtarget** "(" sensorID "," object ")"
sensorID	::= name | number
name	::= letter {letter | digit }
number	::= <positive integer>
object	::= letter {letter | digit }

Parameters	sensorID	*Name or number of the sensor. Defined in the robot model*
	object	*Name of object. Defined in the robot model*

Example door = findtarget(scanner,door)

Actuator() is a command which is used in direct control of motors, relays and other actuators of a robot.

Actuator	::= **actuator** “(“ actuatorID “,” value “)”
actuatorID	::= name | number
name	::= letter {letter | digit }
number	::= <positive integer>
value	::= on | off | number
number	::= <a real number>

Parameters	actuatorID	*Name or number of the actuator. Defined in the robot model*
	value	*Reference value for the actuator.*

Description The first parameter is a name or a number of an actuator and the second parameter is a reference value for that actuator. The user can define name, number and range for each actuator in *the robot model*. What actually happens, when an user gives, for example, a command actuator(leg1,90), is as follows: First the interpreter checks if a name leg1 is found. Then it checks if the value 90 is within the admissible range for leg1. After that the message is sent to the intermediate process that contains a user-programmed method to handle this request.

Example actuator(brake,on), actuator(leg1,90)

Init() is used to initialize sensors, services and internal variables.

Init	::= **init** “(“ [address] [{“,”address }] “)”
address	::= letter {letter | digit }

Parameters *without any parameter all sensors and services are initialized. Parameter can be a name for a sensor, service or a variable. It is defined in the robot model.*

Example init(), init(navi)

Manipulate() is a command to switch a manipulation task on.

Manipulate	::= manipulate "(" param ")"
param	::= take | leave | getball | giveball

Example manipulate(take)

Description This command sends a message to the process manipulator in a robot. When manipulation is done the process manipulator sends a reply and the command manipulate terminates.

4.5.5 Control flow commands

The commands of the class control flow are used to control the execution of other commands. They include commands to build conditions and loops.

If() is a condition command which is followed by a command block which ends with the endif command. Inside the parentheses there can be a variable or a comparison expression.

If	::= **if** "(" expression ")" {commands} **endif**
expression	::= variable | comparison
comparision	::= variable ("<" | "<=" | "==" | ">" | "=>") (number | variable)
variable	::= letter {letter | digit }
number	::= <a real number>
commands	::= {ILMR command}

Example

```
dist = goahead()
if(dist>5.5)
 break(goahead)
 turn(left)
endif
```

While() – EndWhile is a way to build a loop. Inside the parentheses there can be a variable or a comparison expression.

While-loop	::= **while** “(“ expression “)” {commands} **endwhile**
expression	::= variable | comparison
comparision	::= variable (“<” | “<=” | “==” | “>” | “=>”) (number | variable)
variable	::= letter {letter | digit }
number	::= <a real number>
commands	::= {ILMR command}

Example

```
followwall(middle)
cnt = 0
loop=while(cnt<5)
  door = sensor(scanner,door,left)
  if(door)
    cnt = cnt + 1
  endif
endwhile
wait(loop)
break(followwall)
gotoxy(door)
```

Description In this task the robot moves along a corridor in the middle. It keeps looking for doors on the left side and after the fifth door the robot stops following the wall and goes to the middle of the door opening.

Break() command stops the robot and clears the command list or terminates a desired command.

Break	::= **break** [“(“ name “)”]
name	::= letter {letter | digit }

Example break, break(turn(d=)

Description If a parameter is given, then ILMR tries to locate a command in the list of commands, which contains the text given using the parameter. If a match is found, then the selected command is terminated.

Stop command is used to stop the current operation of a robot. No parameter is needed.

Cont command is used to continue a stopped operation. No parameter is needed.

Wait() will stop the execution of the following commands in the current command block and wait for a desired time interval, which is given in seconds as the only parameter.

Wait	::= **wait** "(" number | variable ")"
number	::= <real number in seconds>
variable	::= letter {letter | digit }

Example	wait(4.5), wait(door)
Note	The parameter can also be a variable and the wait will be active until the first field of the variable is true.

Clear() is used to clear a variable in ILMR.

Clear	::= clear "(" variable ")"
Variable	::= letter {latter | digit }

Example	clear(counter)

Check-Checkend is used to validate the grammar of the desired task.

Example

```
check
goahead(3)
turn(left)
checkend
```

Description	The interpreter checks the grammar of the task in between the commands check and checkend and gives error messages if there are rrors. No executable objects are created.

4.5.6 Variables

Variables and variable arithmetic is available in ILMR. When the interpreter encounters an arithmetic expression for a variable, it first checks if that variable already exists. If it is found, the operation is linked to that variable. If this is not the case, a new variable is created.

A variable in ILMR is in fact a vector which has several fields and each command knows what field is suitable for it. It has also a value-field that accesses values directly and stores them to variables such as Turn = Turn + 1. The command **door = sensor(scanner,door)** puts the coordinate values of the found door into the fields **X** and **Y** of the variable **door** (**door->X** and **door->Y**). When the above-mentioned command or some other command terminates, it also puts a value 1 into the field **status** of the variable (**door->status = 1**). If the following command is **If(door)**, it checks the field **door->status**. The command **GotoXY(door)** uses the fields **door->X** and **door->Y**. The fields of a variable are shown in Table 8.

Table 8. Fields of a variable in ILMR.

VARIABLE **Field**	**Description**
Status	Gets value 1 when the associated command is ready, otherwise 0
X	x-value in meters
Y	y-value in meters
H	Heading value in radians
Value	The result of an aritmetic operation
Dist	The distance travelled of the associated command

The variables in ILMR are global and they are available in all command blocks. The aritmetic operators -,+,/,* are available in ILMR at the moment.

4.5.7 Obstacle avoidance

Obstacle avoidance can be implemented in many ways. [Sitharama Iyengar and Elfes, 1991] presents the use of artificial potential field. In this approach the goal is regarded as an attractive pole and obstacles are repulsive points. The attractive potential can be defined by an equation (9) and the repulsive potential by another equation (10) [Vidal-Calleja et al., 2002]

$$U_a(q) = \tfrac{1}{2}\xi\rho(q, q_g)\,, \tag{9}$$

where ξ is a positive scaling factor and $\rho(q, q_g) = \| q - q_g \|^2$ is a definite function who has a global minimum equal to zero at $q = q_g$.

$$U_r(q) = \begin{cases} \tfrac{1}{2}\eta\left(\dfrac{1}{\rho(q, q_o)} - \dfrac{1}{\rho_o^2}\right)^2 & if\ \rho \le \rho_o^2 \\ 0 & if\ \rho > \rho_o^2 \end{cases} \tag{10}$$

where η is a positive scaling factor. The region of influence of the obstacle is determined by the positive constant ρ_o. One problem with this approach is avoiding a local minimum, where the sum of repulsive and attractive forces is zero, and another problem is the fact that sometimes the path should go along the other side of the obstacle where the maximum potential is.

A fuzzy controller with a learning automaton is used for obstacle avoidance in [Babvey et al., 2002]. The parameters of the input and the output fuzzy membership functions are determined by a learning automaton at each time step based on sparseness of the obstacles. A neural network for obstacle avoidance is used for example in [Diequez et al., 1995].

A variant of the probabilistic roadmap method (lazy-QPRM) is used in [Lanzoni et al., 2002]. First a roadmap is built with q_{start}, q_{goal} and N nodes distributed uniformly. Then shortest path is found between q_{start} and q_{goal}. If a collision occurs, the corresponding nodes and edges are removed. The planner finds a new path, or updates the roadmap with new nodes and edges. The process is repeated until a collision-free path is found. The result paths are long and irregular, and they are smoothed with an iterative relaxation-based method. This PRM-method assumes that the planner has a complete knowledge of the workspace. With mobile robots, that is not true. They can only sense the portion of the workspace that is nearby and not obstructed by obstacles. Another problem is the computing power required. The calculation cycle can take even several seconds.

In [Prassler et al., 2001] the concept of Velocity Obstacle (VO) is introduced and is used in a robotic wheelchair to compute avoidance manoeuvres of people crossing its path. The VO identifies the set of velocities causing a collision with the obstacle at some future time. In case of several moving obstacles, individual VOs are joined together and avoidance velocities are those that are outside of all VOs.

In ILMR, an environmental map is assumed to be available. In test vehicles that were used, it is formed by means of a 2D-laser scanner functioning at 2 Hz. An example of environmental map is shown in Figure 54 in the chapter 5.

The environmental map is in fact a set of (x,y)-points that are updated when a new scanning is available. The points behind the robot are maintained in the map until the distance has increased over 3 meters.

The environmental map is checked within an area, whose length is defined by the equation (11)

and width by the equation (12).

$$l_{obs} = \max(\frac{1}{R_{\max}} + 2 \times v\,,\ 0.5) \tag{11}$$

$$w_{obs} = 1.2 \times robot\_width \tag{12}$$

where $R_{\max}$ is the maximum curvature (1/m) for a robot. The checked area is in fact a rectangle only when the robot is moving straight. Otherwise the reactangle is stretched according to the curvature of the robot. The checking here means that the coordinate of each point is checked and, if one or more of them is inside the defined area, an obstacle is found.

The path around the obstacle is calcucated in a simple way to search enough space on either side of the obstacle.

The principle of obstacle avoidance in ILMR is shown in Figure 34. The dashed line in Figure 34 is the defined reference path and the dotted square is the area to be checked for obstacles. In figure A, no obstacles are found and the robot goes along the reference path. In figure B, an obstacle is found, and in figure C, the suitable path around the obstacle is calculated. In figure D, the robot leaves the refrence path to go around the obstacle. When obstacle avoidance is going on in figures E and F, the robot continuously checks if the way back to the reference path is free. In figure G, the way back to the reference path is free, but the robot must go on until the rear of the robot has passed the obstacle in figure H.

The state diagram of obstacle avoidance is shown in Figure 35. The state AVOIDING includes the path calculation around the obstacles and the statement OBS_CONTROL=1 says that the normal position control of ILMR is overdriven by obstacle avoidance. If the obstacle is removed or avoided, then the variable OBS_CONTROL gets a value 0, and the position controller drives the robot back to its original route.

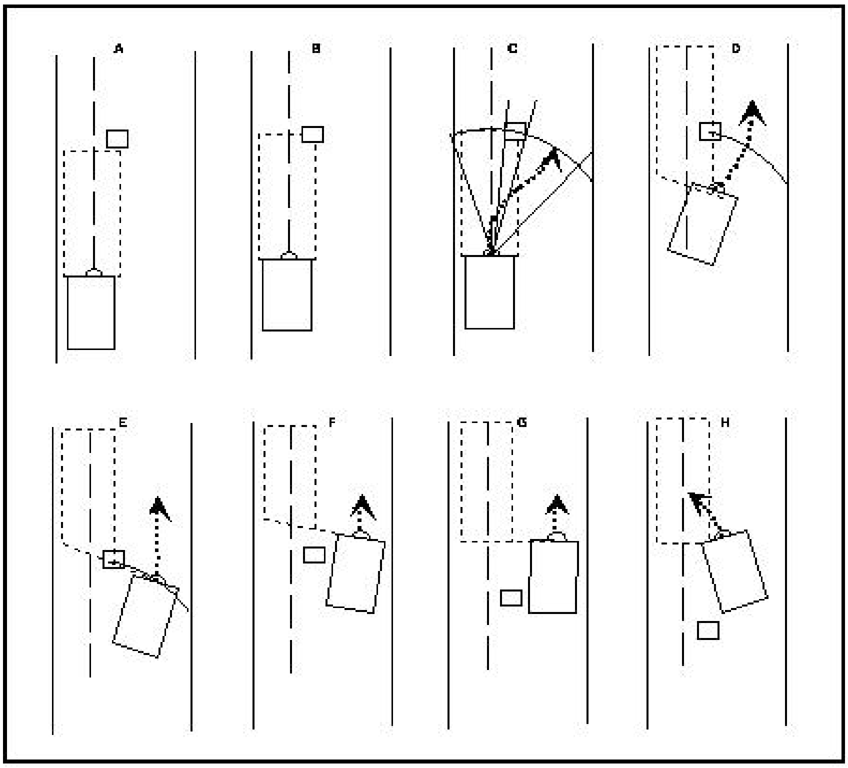


Figure 34. The principle of obstacle avoidance in ILMR.

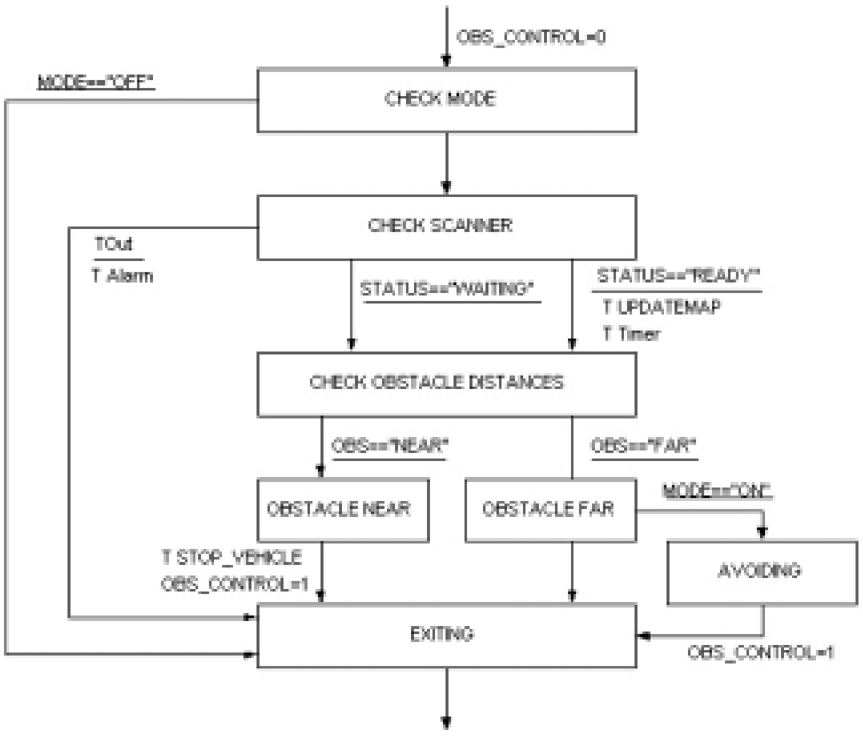


Figure 35. The state diagram of obstacle avoidance in ILMR.

4.6 Use of ILMR with planner

The planner represents the intelligent part of the robot. It is responsible for the decomposition of the intentions of the user into subtasks, and it monitors their execution.

A new command or a mission for a robot is given in spoken English, sign language, by keyboard or by other means. When the planner receives a new command or task, it first checks if a similar task is found in the database, only a part of which is located in local memory, and most of which is on the internet. If a similar task description is found, then the robot’s current status and position is considered and the description is modified accordingly. Then the task is sent in short pieces to ILMR for execution. If the task description is incomplete, then the planner asks the user for supplementary information. If the task description is totally unknown, the planner asks the user to describe it in detail. This new task is then added into a task database as a new skill.

Subtasks, which the planner sends to ILMR can be single ILMR commands or they can be scripts or beforehand learned skills, which are stored in the database of ILMR.

The planner is not yet ready and the development work is going on and the connection to ILMR is not yet fixed but the current plan is as explained above.

5 Experiments with ILMR

The developed intermediate language, ILMR, has been implemented in two mobile robots. A centaur type service robot Workpartner, which has been built at the Automation Technology Laboratory of Helsinki University of Technology [Halme et al., 2001a] was the primary application goal for ILMR. The second robot, MoRo (Mobile Robot) has been built at VTT to act as a testbed for development of autonomous robots. All features and algorithms were first implemented and tested in MoRo, and when they were functional enough, they were also implemented in Workpartner.

5.1 Workpartner

Workpartner is a centaur type service robot with four legs, body and a human-like upper body with two hands and a head. Each leg is equipped with a wheel and both hands with two-finger grippers. The head is equipped with a laser pointer, which is used to measure the distance to a single point, and a colour CCD camera. Appearance of Workpartner can be seen in Figure 36.

The size of Workpartner is such that it is suited to co-operating with humans and can use same tools as humans. The weight of Workpartner is about 230 kg and payload is about 60 kg. The operation time is 4–5 hours with 2 litres of petrol. The maximum curvature in movement is 0.25, which means that Workpartner can turn around in an area with a diameter of 8 m when driving with wheels.

Workpartner has been designed for autonomous tasks such as cleaning of outside areas, small agricultural works, guard, guidance and so on.

The movement of Workpartner can occur in three different ways. On even environment the legs can be locked and Workpartner acts as an ordinary vehicle with wheels. On uneven or otherwise difficult environment such as thick snow the robot slides its legs along the ground. This movement mode is called rolking [Halme et al., 2001b]. With the active leg-system the robot can also step over small obstacles and even climb the stairs. Turning is done using the articulation joint of the body.

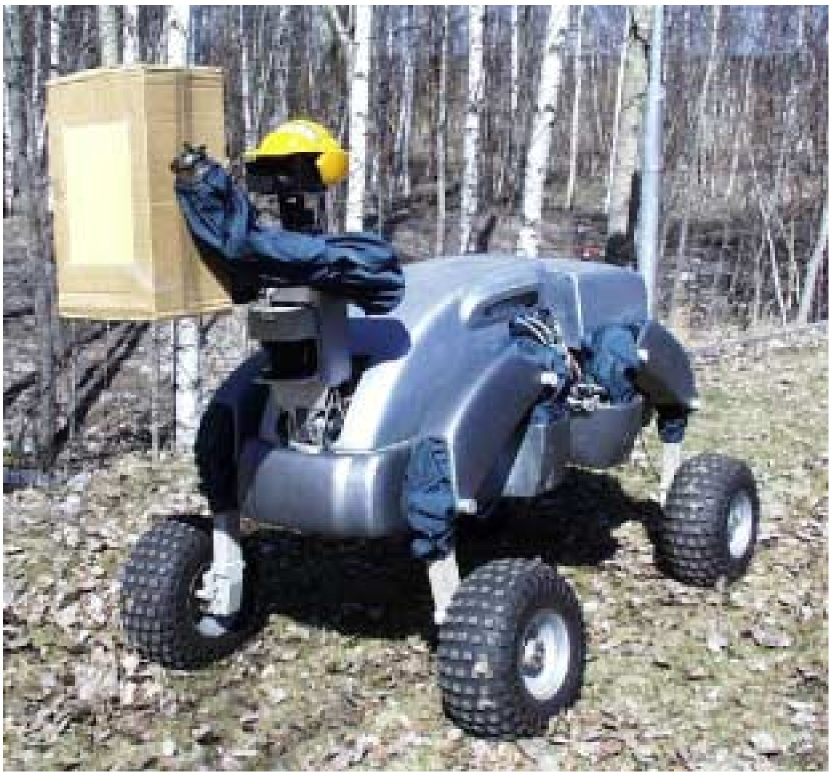


Figure 36. Centaur type robot Workpartner.

5.1.1 Hardware and software architecture of Workpartner

Each leg and the articulation joint are controlled by means of a micro-controller unit. The robot is also equipped with a PC104-size main computer running the QNX operating system. The CAN bus is used to transfer information between subsystems. The navigation subsystem is also equipped with a separate PC104-size computer and communication between the main PC and the navigation PC is arranged through a twisted-pair Ethernet cable. The CAN-based control system of Workpartner is shown in Figure 37.

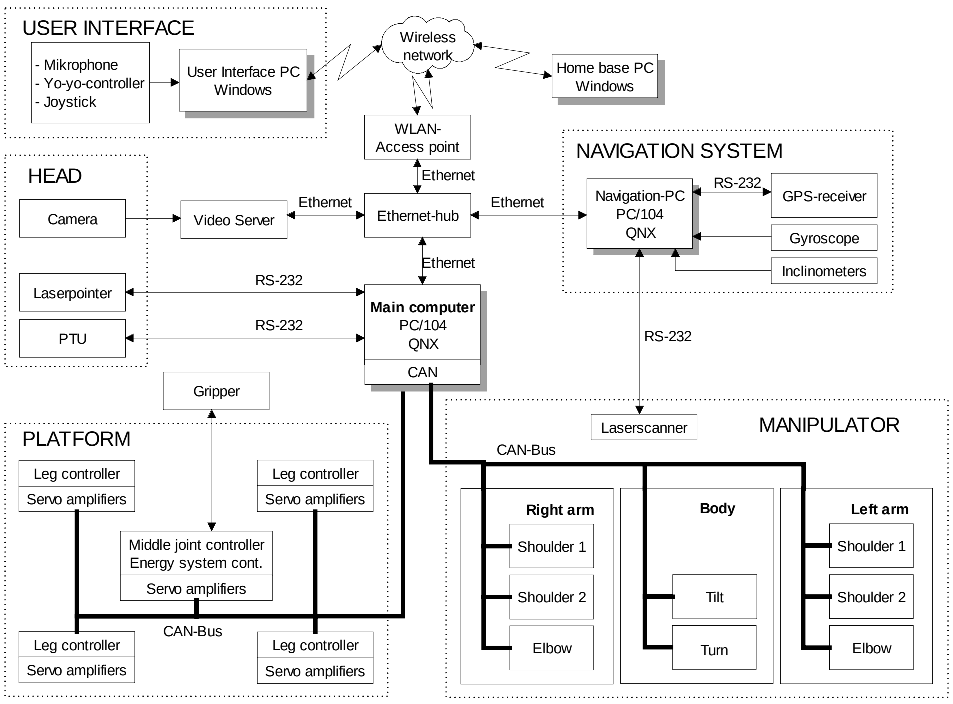


Figure 37. CAN-based control system of Workpartner.

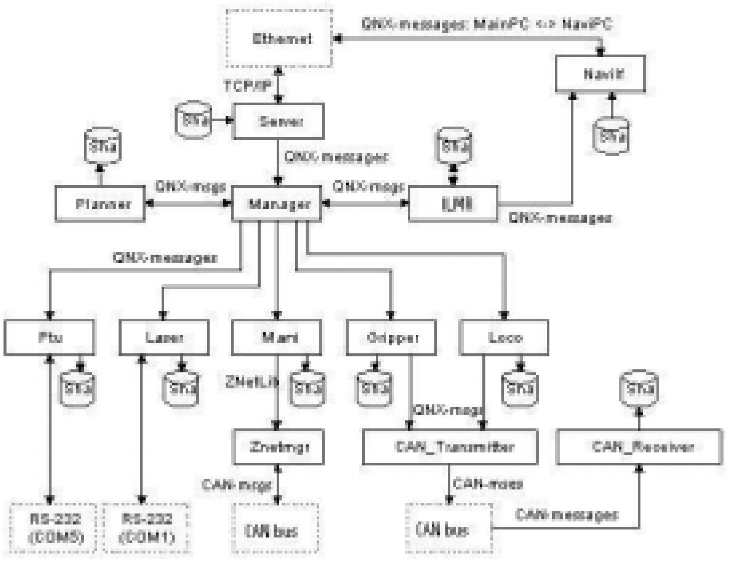


Figure 38. Software architecture of Workpartner.

The software architecture of Workpartner is shown in Figure 5.3. The figure shows all tasks running in Workpartner and ILMR is only one task among others. The communication is handled both synchronously by QNX-messages and asynchronously by shared memory (Sha). The reason why ILMR does not seem to occupy a central place in the figure is the fact that other software was already developed to some extent until it was discovered that an intermediate language was needed to make management of the whole system easier.

5.1.2 Sensors of Workpartner

Workpartner is a complicated robot with several subsystems and sensors. The main sensors are explained in the following and more detailed description of sensor system can be seen in [Selkäinaho, 2002].

The wheel revolutions of Workpartner are measured by means of Hall sensors and joint angles are measured by potentiometers. Inclinations are measured using gravity-based inclinometers.

Navigation is based on 2D-laser range finder Sick LMS 291. Successive laser range maps are matched when there are not any known landmarks available. Sometimes the robot also recognizes landmarks as vertical cylinder objects or vertical planes. Using the Hough transform, the robot can estimate its heading position with a 0,2° accuracy relative to the vertical plane. The robot's position can be estimated with the accuracy of 1 cm when two vertical planes are available. With vertical cylinders the corresponding values are 0,5° and 0,1 m. The navigation subsystem gives new position information at a frequency of 2 Hz. More detailed description of the navigation system can be seen in [Selkäinaho, 2002].

The robot head is equipped with a colour camera, which is used to look for objects around the robot and a laser pointer, which can be used to measure distance to a certain point in the camera image.

In its rear part Workpartner also has four ultrasonic sensors to measure distances backwards.

The control architecture of Workpartner is hybrid. It contains features from reactive control and from hierarchical or planning control.

In Figure 39 the robot model file ROBO.INI is shown for WorkPartner. The first tag [SENSORSANDACTUATORS] is followed by a list of sensor and actuator names. After each name there is an ***integer*** that can be used instead of a name as a command parameter in any ILMR command. The next value can be ***double*** and it gives the minimum value of a sensor reading, and the next one gives the maximum value. The last value gives the number of readings associated with that sensor. The second tag [OBJECTS] is followed by two named objects user and side_edge, and both of them are followed by two parameters. Object name will be replaced by parameters when used in ILMR. After tag [IDENTIFIERS] a user can define identifiers that will be replaced by the following parameters when used in ILMR. Tag [PARAMETERS] is followed by some default values for internal variables in ILMR. All of them have however a default value that is used if the file ROBO.INI is missing or corrupted.

```
[SENSORSANDACTUATORS]
gyro 0 -180 180 1
scanner 1 0 30000 361
laser 2 0 30000 1
ptu 3 0 100 2
[OBJECTS]
user yellow 30
side_edge 200 30
[COLORS]
yellow 18
[IDENTIFIERS]
off 0
automatic 1
scanner 2
gps 3
odo 4
rolking 4
wheels 5
parking 6
[PARAMETERS]
max_time_to_search 10
default_speed 0.2
max_speed 0.6
pos_interval 0.5
omax_curvature 0.4
curinc 0.8
robot_length 2.2
robot_width 1.0
followwall_d 1.0
bs_security 1.0
[END]
```

Figure 39. The robot model file ROBO.INI in WorkPartner.

5.2 MoRo

MoRo is a traditional automatically guided vehicle (AGV). Its size is 120 cm x 60 cm, and it can turn around in an area with a diameter of 1.5 m. The steering of the robot is done by means of the front wheel only, which is turnable and is also equipped with the driving motor of the robot. In the rear, the robot has two wheels supported by the same axle. All wheels are equipped with pulse sensors to measure the revolutions of the wheels and, in addition, the front wheel is equipped with a potentiometer to measure the turning angle. Below the body, there is an array of inductive sensors that are used to detect the round metal plates when driving over designated positions. The deviation of the robot can be measured with the accuracy of 1cm using inductive sensors. MoRo contains also a 2D-laser range finder Sick LMS 291 that is used for navigation, obstacle avoidance, and as a security sensor/switch. An optical fibre gyroscope is used to measure the heading angle, but it can also be measured with a good accuracy using only differential odometry.

MoRo is also equipped with a PC104-size main computer running the QNX operating system. It has also a cheap radio-unit, which can be used to communicate with an external PC.

The appearance of MoRo can be seen in Figure 40 and the control architecture of MoRo is shown in Figure 41.

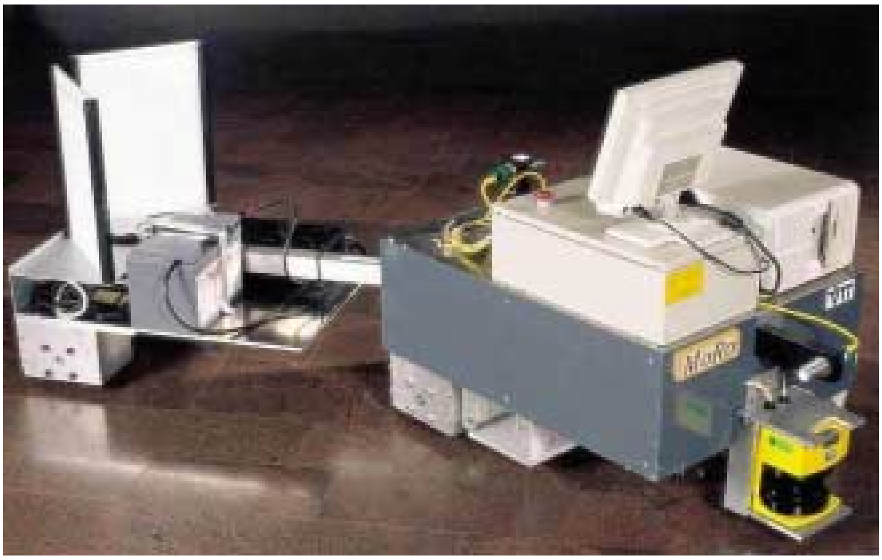


Figure 40. Mobile robot MoRo.

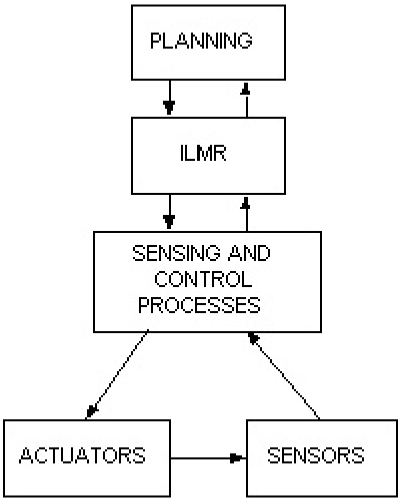


Figure 41. Control architecture of MoRo.

5.3 Experiment site

The experiments with Workpartner were carried out around the Computer Science Building of Helsinki University of Technology. This test environment is shown in

Figure 42. The experiments with MoRo were carried out at an office environment at the VTT building, and this environment is shown in Figure 43.

Figure 42. Test site for Workpartner.

Figure 43. Test site for MoRo.

5.4 Simulators

During the development work a huge number of experiments have been carried out when developing each feature and every single command for ILMR. A simulator environment has also been developed for Windows NT with Borland C++ Builder. It was used to develop the algorithm for the position controller in ILMR. This simulator has a model for both test vehicles, Workpartner and MoRo. These models have been built in an iterative way, in which a trajectory was driven with a robot and a current version of the position controller. Then the same trajectory was driven with the simulator, and parameters were manipulated in such a way that results were equal. The simulator contained also a genetic algorithm to search optimal values for parameters. The main window of the simulator is shown in Figure 44.

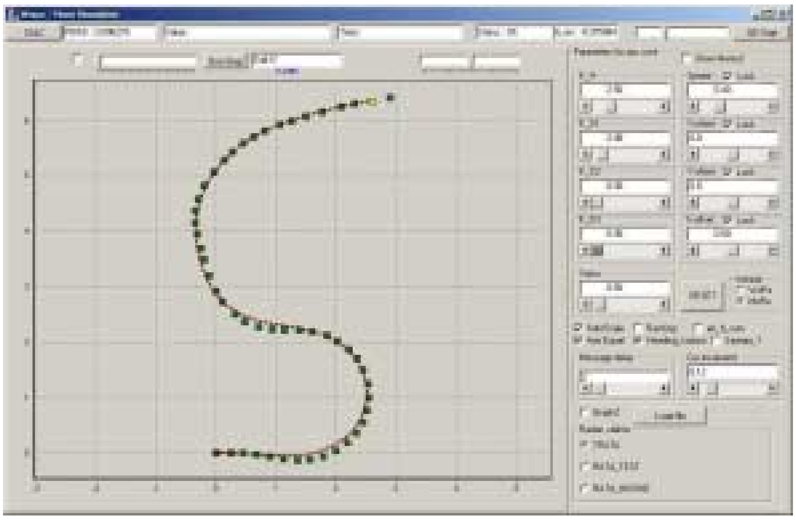
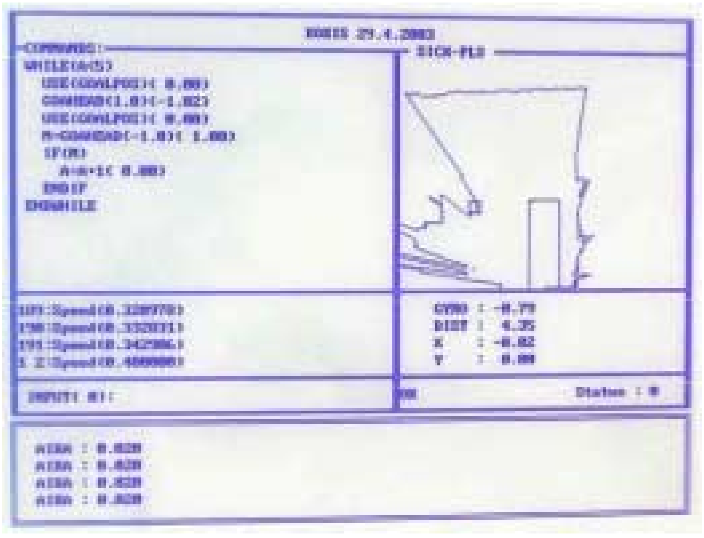


Figure 44. Windows NT-based simulator to be used for the development of the position controller.

Another simulator has also been developed for QNX-environment. The whole language, ILMR, with all its features is available in this simulator and it has been the main tool when developing commands and features for ILMR. The same processes are running in the simulator and MoRo. The only difference is that movements and sensor readings are simulated. The same task could anayway be given for both the simulator and MoRo and when the task execution was successful in the simulator it could also be tested with MoRo. A snapshot of this simulator is shown in Figure 45.



```
                                        BORIS 29.4.2003
COMMANDS:                                            SICK-PLS
WHILE(A<5)
  USE(GOALPOS)( 0.00)
  GOAHEAD(1.0)(-1.02)
  USE(GOALPOS)( 0.00)
  M-GOAHEAD(-1.0)( 1.00)
  IF(M)
    A=A+1( 0.00)
  ENDIF
ENDWHILE

[illegible]:Speed(0.328770)                    GYRO : -0.79
[illegible]:Speed(0.332031)                    DIST :  4.35
[illegible]:Speed(0.342986)                    X    : -0.02
[illegible]:Speed(0.400000)                    Y    :  0.00

INPUT( 0):                                 [illegible]        Status : 0

  AIKA : 0.020
  AIKA : 0.020
  AIKA : 0.020
  AIKA : 0.020
```

Figure 45. QNX-simulator for ILMR.

5.5 Experiment plan

All designed commands and features of ILMR have been verified first with the simulator and then with two different robots. The experiment plan is shown in Table 9.

Table 9. Experiment plan.

TEST 1	**Movement command test**
Purpose	To test the performance of a movement command. The command TURN is selected for a representative of movement commands. The behaviour of turn command with detailed parameter is the same as it is with other movement commands.
Parameters	D=-90, c=0.4 and 0.6 for MoRo and c=0.25 for Workpartner
Speed	0.2…0.8 m/s for MoRo and 0.4 m/s for Workpartner
Number of tests	18 (15 with MoRo and 3 with Workpartner)
TEST 2	**A command with an abstract parameter**
Purpose	To test the performance of a command with an abstract parameter. The command Turn(left) was used in this test.
Parameters	Left
Speed	0.3 m/s
Number of tests	5 (With MoRo)
TEST 3	**Followwall test**
Purpose	To test the performance of a followwall command. This command is behaviour-based and its behaviour was tested here.
Parameters	Left,1
Speed	0.3 m/s
Number of tests	3 (With simulator)
TEST 4	**Obstacle avoidance**
Purpose	To test the performance of obstacle avoidance behaviour.
Commands	Obsavoid(on) and Goahead(5)
Speed	0.3 m/s
Number of tests	3 (With simulator)

TEST 5	**Short task**
Purpose	To test the linking of commands together
Commands	List of commands shown in Figure 52
Speed	0.3 m/s
Number of tests	3 (With MoRo)
TEST 6	**Loop test**
Purpose	To test the performance of a loop command while. A sequence of several movement and other commands was carried out three times.
Commands	List of commands shown in Figure 56
Speed	0.3 m/s
Number of tests	1 (With Workpartner)
TEST 7	**A task with abstract parameters**
Purpose	To test how a way out of the laboratory to a certain doorway is described with natural language and with ILMR and to show how the robot carries out the given task.
Commands	List of commands shown in Figure 58
Speed	0.3 m/s
Number of tests	1 (With MoRo)

All movements of the robots with ILMR, except for **followtarget, followwall** and **obstacle avoidance**, are based on a planned trajectory. Thus the behaviour of all movement commands is similar and it is enough to show the performance with one command. This is done for the **turn-**command separately with both robots with the test 1.

The command **turn** with an abstract parameter **left** has been tested with the test 2. The command **turn(left)** behaves differently than with detailed parameters. If there is not room enough on the left the robot keeps the distance to the left and moves forward until there is room enough to turn. Then the suitable path to the left is generated and the robot follows it.

The behaviour of the command **followwall** has been tested with the test 3 and the obstacle avoidance with the test 4 and then a list of movement commands, a script, has been tested to show how movement commands can smoothly be linked together with the test 5. Also looping command of ILMR have been tested with simple tasks with the test 6 and finally a more complicated task with abstract parameters has been tested with test 7.

The command **followtarget** has not been reported separately, but it was successfully used in many other tests to teach a robot a new path.

All experiments have been done with the final language, though some experiments with Workpartner during the development work are also presented to show how ILMR has been used in different tasks. Some of tests have been carried out with the qnx-simulator, that was running the same processes as MoRo, but the movements and sensor readings were simulated.

The successful performance is estimated by two criterias: deviations from the desired paths when it is possible and fluent advance.

5.6 Results of experiments

5.6.1 Tests 1 and 2: Turn

The parameters for the command turn can be given in two ways : numerically and abstractively. In this test the robot was asked to turn left 90 degrees with different curvatures and speeds. For MoRo the curvature was 0.4 and 0.6 and for Workpartner it was 0.25. The speed was in range 0.2–0.8m/s. The result trajectories are shown in Figure 46 and the results in Table 10. The results with right turns are equal and they are not shown here. The error is biggest with Workpartner and with higher speed. The reason for that is the fact that the given curvature was too near to the maximum curvature of Workpartner and the reference curvature was clipped as can be seen from Figure 48. This result requires some modification to the code of ILMR.

In the test 2 the robot was asked to turn left with the command **turn(left)** and the result trajectories are shown in Figure 47 and results in Table 11. In this test, the robot MoRo was driving along a corridor and when the corner was detected, in other words there was enough room on the left, the trajectory was generated and the robot followed it. When the command turn(left) was given, the robot started to follow the wall on the left and looked for space to turn. Table 11 and Figure 47 tells that the performance of one trial was different than the others. The reason could have been an erronous data from the scanner. Otherwise the performance is good.

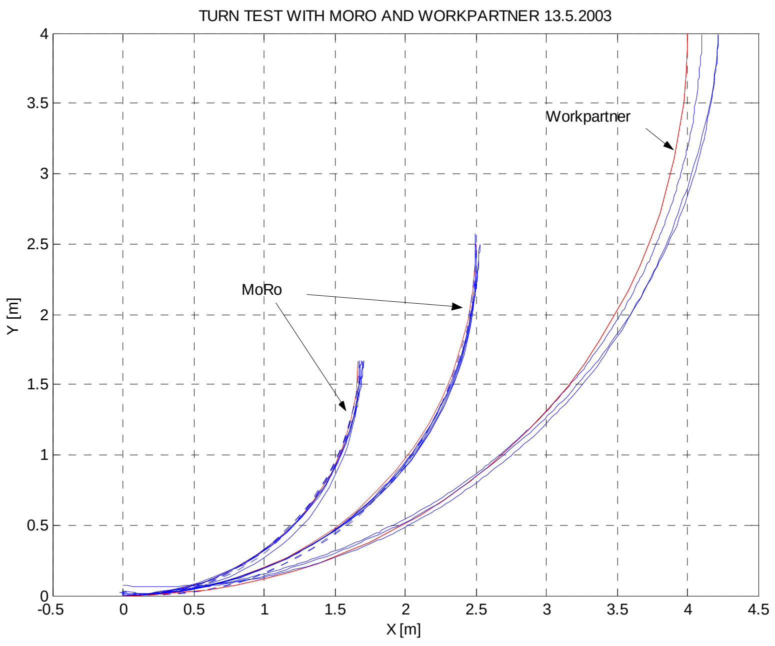


Figure 46. Trajectories of the turn test with both robots.

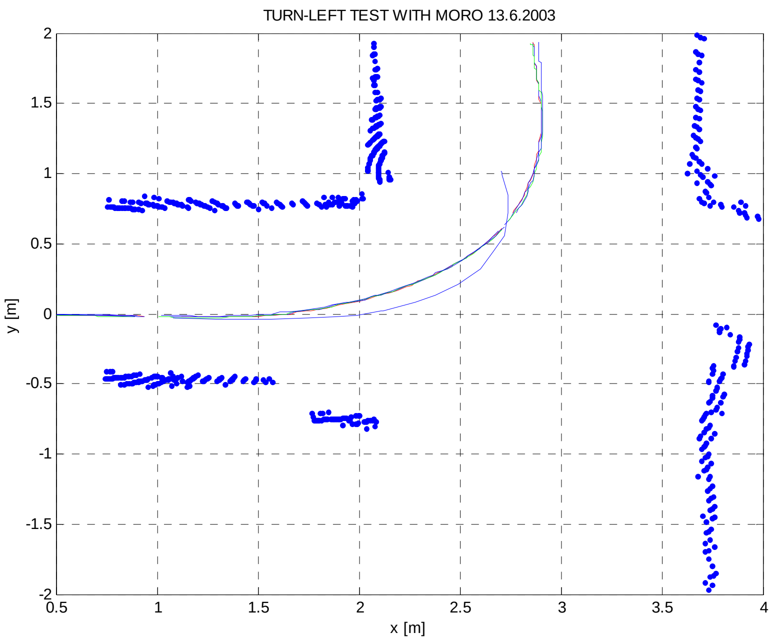


Figure 47. Turn left test.

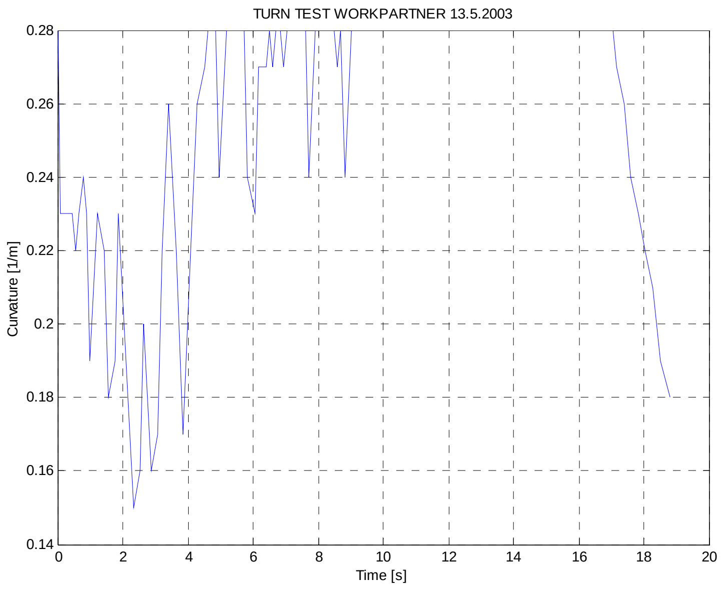


Figure 48. The clipping of the curvature output of the position controller in a test with Workpartner.

Table 10. Maximum deviations from the path in test 2 with means and standard deviations.

TEST 1: TURN(d=90,c=x.x)			
Number of trials	Maximum deviation from the path	Mean of maximum deviations	Std. deviation of maximum deviations
8 (c=0.60, v=0.2-0.8m/s)	0.08 m	0.0225 m	0.0242 m
7 (c=0.40, v=0.2-0.8m/s)	0.04 m	0.025 m	0.0115 m
3 (c=0.25, v=0.2-0.4m/s)	0.22 m	0.183 m	0.0635 m

Table 11. Maximum deviations from the path in test 3.

TEST 2: TURN(LEFT)		
Number of trials	Maximum deviation on x-axis	Maximum deviation on y-axis
4	0.04 m	0.01 m
1	Deviant function	

5.6.2 Test 3: followwall

In this test the robot was asked to follow the wall on the left side at the distance 1.0m. The used command was **followwall(left,1.0)** and the recorded paths are shown in Figure 49. The algorithm is such that door-openings are ignored and thus the corner was detected late.

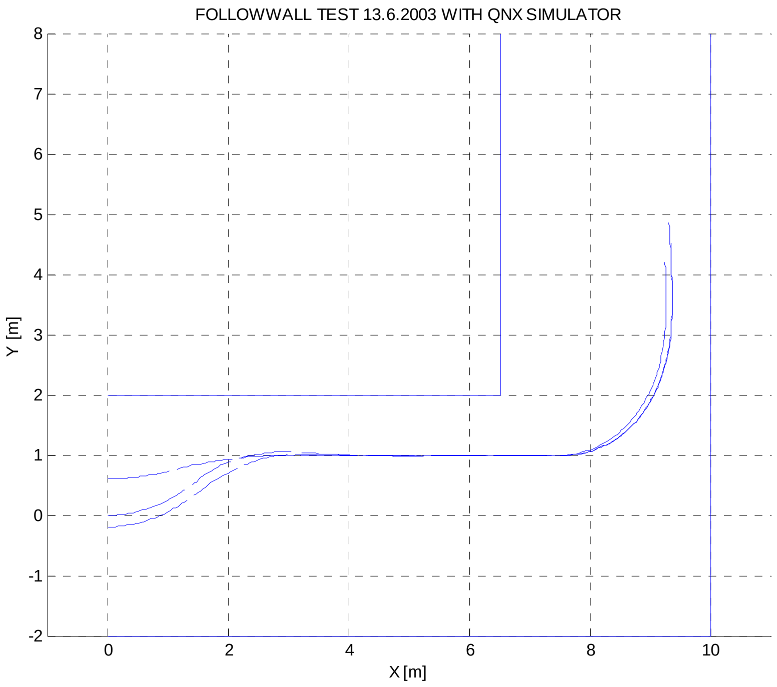


Figure 49. The trajectories of the followwall test with qnx-simulator.

Three tests were carried out with different initial distances to the wall and to the desired distance. The speed of the robot in this test was 0.3 m/s and the desired distance was reached within three meters. These tests were carried out with the qnx-simulator and the performance of the followwall behaviour with a real robot can be seen in test 7. In practise the performance is the same for both the simulator and the robot MoRo. The results are shown in Table 12.

Table 12. Maximum deviations from the desired distance to the wall in test 3.

TEST 3: followwall(left,1)		
Number of trial	Deviation from the desired distance	Excess of the goal distance
1	0.4 m	0.02 m
2	1.0 m	0.05 m
3	1.2 m	0.03 m

5.6.3 Test 4: obstacle avoidance

Obstacle avoidance is a behaviour that is activated when the robot is moving and an obstacle is detected. It was tested successfully with MoRo several times. Here in the Figure 50 is a result of an obstacle avoidance test with the qnx-simulator for ILMR. The behaviour of the algorithm for obstacle avoidance is the same in the simulator and in real robots and it will work same way.

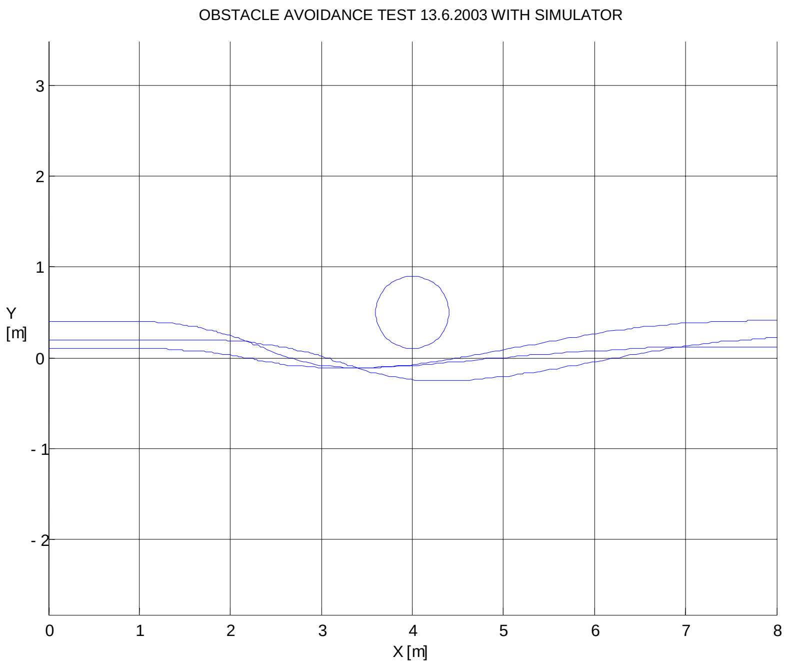


Figure 50. Obstacle avoidance with the qnx-simulator.

The environmental map built by MoRo in a test at the laboratory is shown in Figure 51.

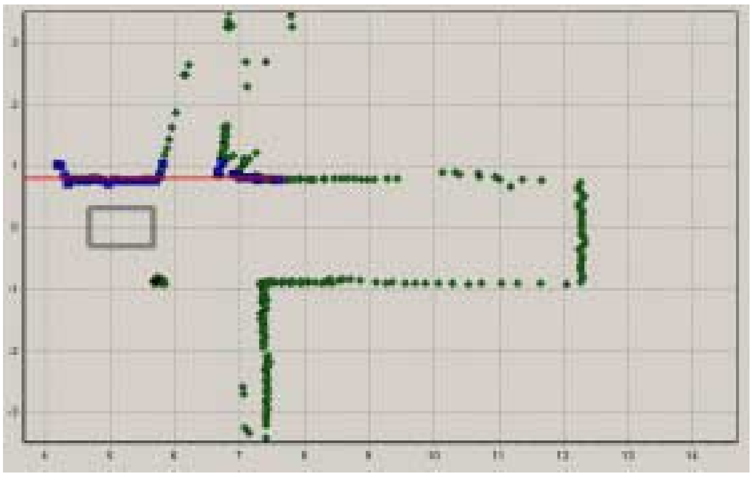


Figure 51. An environmental map of the test vehicle MoRo.

The diamonds in Figure 51 are the points from the walls and they are calculated from the scanner data. The line in figure is the calculated wall which can be used with the followwall command.

5.6.4 Test 5: a short task

In this test the robot was asked to carry out a short task which is shown in Figure 52. In this test several movement commands are linked together. It means, for example, that when the command **goahead(1.2)** is nearly finished (0.5 m left) and the language knows that the operation continues directly with the following movement command **turn(d=90,c=0.5),** the command **goahead(1.2)** is terminated and the left path (0.5m) is linked to the beginning of the **turn(..)** command.

The result trajectories are shown in Figure 53 and the maximum positional error is 0.07m when the robot is going back to the starting point. In Figure 54 the speed variation during the test is shown and Figure 55 shows the growth of the x-coordinate when two commands are linked. It is shown from the recorded data and can be seen from Figure 55 that it takes about 0.1 second to link the remaining part of the previous command to the next command. The performance is good all the time during the test and the behaviour of the robot is fluent all the time. There are no problems with continuity when commands are linked together. The post-condition of the last command matches always with the pre-condition of the next command.

```
speed(0.4)
goahead(1.2)
turn(d=90,c=0.5)
m=goahead(0.7)
wait(m)
speed(-0.4)
use(goalpos)
goahead(-0.7)
turn(d=90,c=0.5)
goahead(-1.2)
```

Figure 52. The description of a short task to a robot in ILMR.

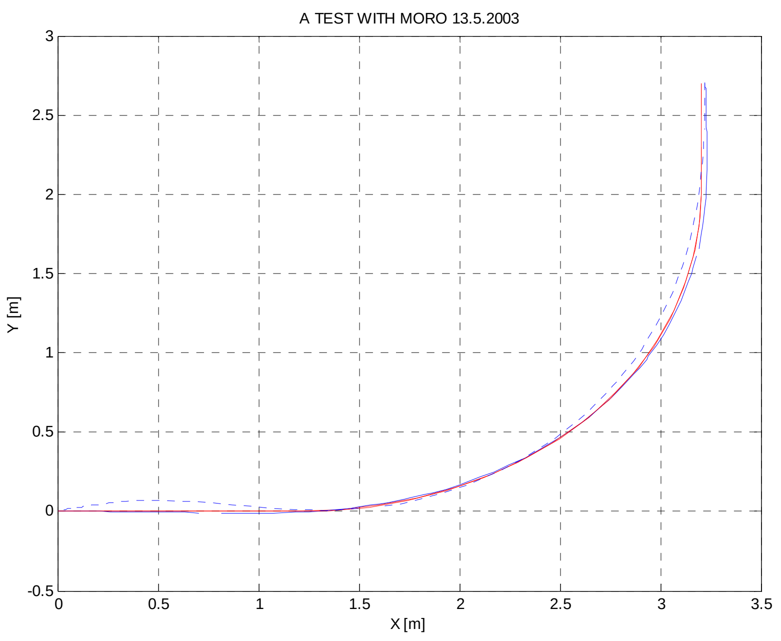


Figure 53. The result trajectories when the robot MoRo has carried out a short task.

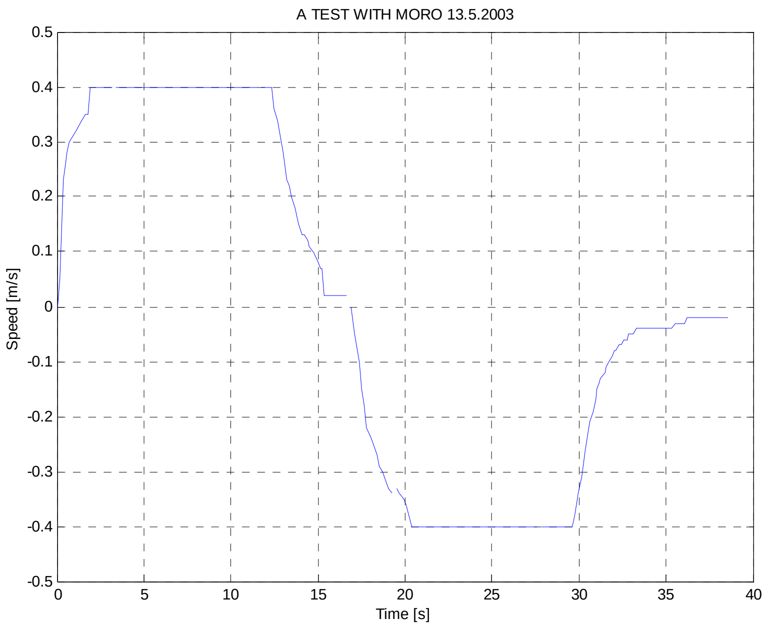


Figure 54. The speed of MoRo when carrying out a task shown in Figure 52.

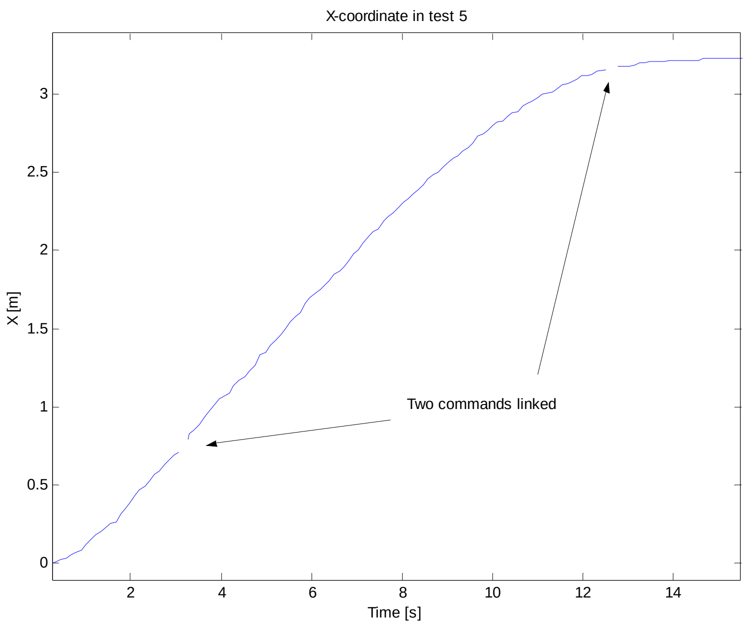


Figure 55. X-coordinate of Moro when carrying out a task defined in Figure 52.

5.6.5 Test 6: loop

In this test the performance of a loop-command **while** was tested. The task used for Workpartner is shown in Figure 56 on the left and definition for subtasks TURN1 and TURN2 are shown on the right. The result trajectory is shown in Figure 57.

```
locate(0,0,0)
wait(1)
cnt=0
while(cnt<2)
 speed(0.4)
 f=followpath(PATH)
 wait(f)
 TURN1
 speed(0.4)

f=followpath(PATH,BACKWARDS)
 if(f)
  cnt=cnt+1
 endif
 wait(f)
 TURN2
endwhile
```

TURN1:

```
speed(-0.4)
t1=turn(d=90,c=0.25)
wait(t1)
```

TURN2:

```
speed(-0.4)
t1=turn(d=-90,c=0.25)
wait(t1)
```

Figure 56. Loop-test with Workpartner.

The word PATH is the name for the trajectory that was taught to Workpartner beforehand. In the loop the taught path was driven backwards and forwards twice.

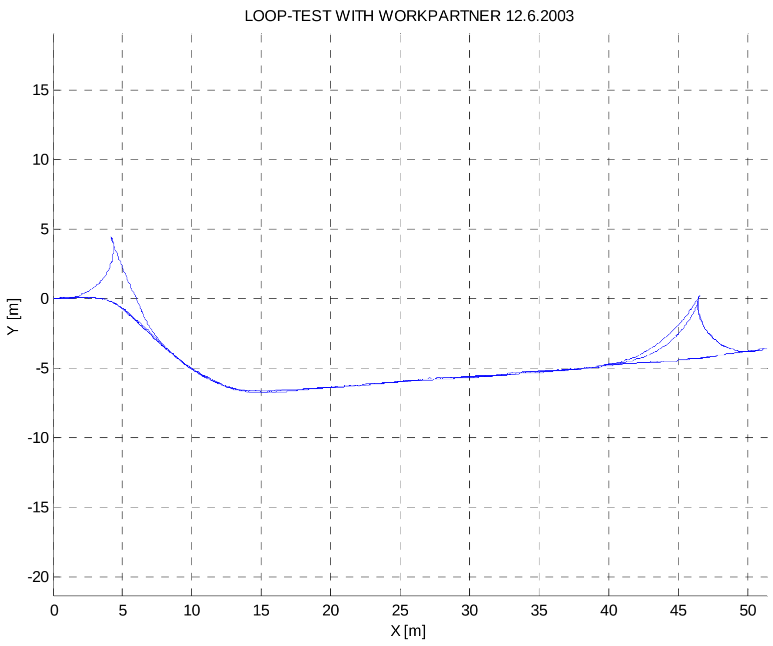


Figure 57. The trajectory of Workpartner when carrying out the task shown in Figure 56.

The performance in this test was good and the total distance Workpartner travelled was more than 200 m with several manoeuvres.

5.6.6 Test 7: a task with an abstract description

The last test with ILMR was a short task that was defined in an abstract way. The task description is shown in Figure 58. On the left is a description that a person could use when describing to another person and on the right is the corresponding description with ILMR commands. It is a task in which a way out of the laboratory is learnt using abstract parameters for commands of ILMR. First the robot turns to the left when there is enough room. If there is not, it keeps the distance to the objects on the left and looks for the room on the left. Next it turns again to the left. When this turning is over, it looks for a door on the right and continues to the door. Then it turns to the left again twice and starts to follow wall on the left. When five meters have been passed, the command followwall is terminated, and the robot starts to look for space to turn again to the left. When this turning is over, the robot goes still ahead one meter.

The result trajectory of this test is shown in Figure 59. The scanner readings on the left and right are recorded, and they are shown in the figure in addition to the robot's position. The task execution was successful with the MoRo robot and the path was learnt and could be driven next time with the command followpath(path-1).

This is an example of a task that can be given using natural language and is easily transferable to the ILMR commands in a high-level planner.

- **Take first two turns to the left**
- **And then look for the door on the right**
- **Go to the door**
- **Take again two turns to the left**
- **And follow the wall on the left about 5 meters**
- **Turn to the left**
- **And move forward 1 meter**

```
init()
obsavoid(on)
record(path_1)
turn(left)
t=turn(left)
wait(t)
door=sensor(scanner,door,right)
wait(door)
gotoxy(door)
turn(left)
t=turn(left)
dist=followwall(left)
if(dist>5)
  break(followwall)
  t=turn(left)
  goahead(1)
endif
record(close)
```

Figure 58. The same task descripton of test 7 with natural language and ILMR.

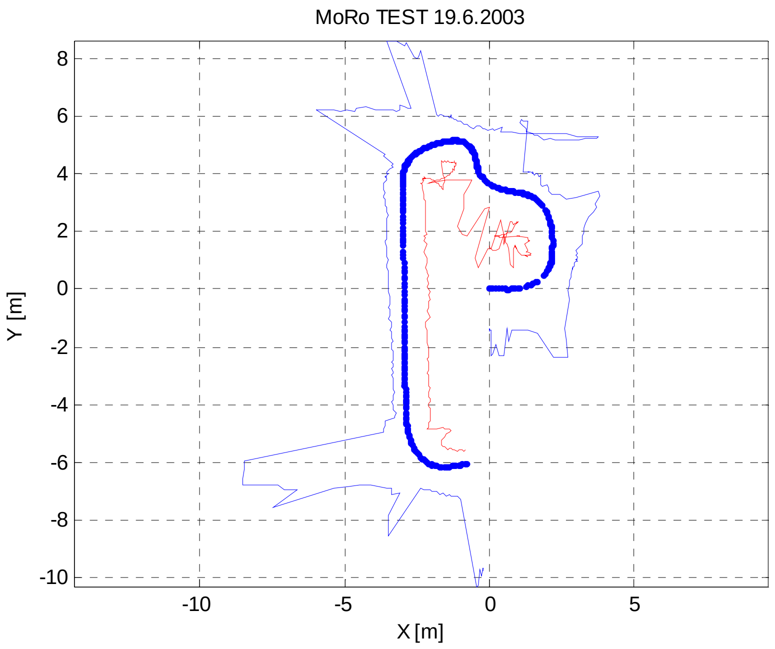


Figure 59. The result trajectory of the task shown in Figure 58. Also walls and other obstacles shown by the scanner are drawn in the figure.

5.6.7 Other tests

In [Sievilä, 2003] the robot Workpartner had a mission to search for drilling holes using a camera. When a hole was found, the robot moved to the hole and measured its profile. Next it searched for another hole and measured it and this was repeated until the last hole was measured. The movements in this mission was done using ILMR and another software utilized the commands of ILMR. It gave means for an application developer to utilize easily the subsystems of the robot.

During the summer 2002 an interesting test was successfully carried out with Workpartner. First a path was taught to Workpartner with commands **record(path)** and **followtarget(user,4),** and then it automatically drove to the working area with the command **followpath(path)**. It stopped and looked for a box with a command **findtarget(scanner,box)**. Then it drove to the box with the command **gotoxy(x,y),** and took the box using the trigger command **manipulate(take)**, which started the manipulation process in another unit of Workpartner. After that Workpartner drove away and laid down the box with the trigger command **manipulate(leave)**.

6 Conclusions

An intermediate language for mobile robots (ILMR) has been presented in this thesis. It makes it easier to design new tasks for a robot. ILMR is intended for service and field robots, and it acts as an intermediate link from user, an intelligent planner or a human-robot interface to a robot's actions and behaviours. The main principle in the development work has been simplicity and ease of use. Neither deep knowledge of robotics nor good programming skills are required when using ILMR. While easy to use, ILMR offers all the required features that are needed to control today's specialised service and field robots. These features contain sequential and concurrent task execution and response to exceptions.

In the future, robots will be used also in homes, and ordinary people should be able to give tasks for robots to perform. This should be done descriptively using natural language as in describing tasks to another person.

ILMR is intended to be used with higher-level abstract languages, such as sign language or even natural spoken language. An action in ILMR can be given coarsely, i.e. in an abstract way, or in detail. Due to this coarseness, ILMR is suitable to be used with higher-level abstract languages and the set of elementary commands supports directly the use of natural language, and therefore the co-operation between ILMR and the high-level planner in a robot is easy.

The user can also use ILMR commands directly to control a robot or construct subtasks as scripts or lists of ILMR commands. Simple tasks for mobile robots can be easily given in ILMR and even more complicated tasks can be easily designed descriptively using direct ILMR commands.

ILMR has been implemented in two different kinds of robots, and its use and performance has been studied with simulators and actual robots in a wide variety of tests. The structure and operation of ILMR has proved to be useful and several tasks have been carried out successfully using both test robots.

In the future, the set of elementary commands will be expanded to also allow manipulator commands and coordinated movements between the body and the manipulators of the robot. The use of variables will also be made more fluent in the future and some new control structures will also be added.

The abstract use of commands will also be developed in the future to ensure a still more fluent connection from natural language to robot control. This includes the parameterization of scripts.

The intelligent monitoring of the status of the robot will also be developed in the future to reject any given command that could somehow harm or damage the robot.

References

Albus, J.S., McCain, H.G. and Lumia, R. [1987]. NASREM: Standard Reference Model for Telerobot Control. NASA 1987 Goddard Conference on Space Applications of AI and Robotics.

Asimov, I. [1989]. The Complete Robot. Great Britain, Glasgow, 1989. 682 p.

Babvey, S., Momtahan, O. and Meybodi, M.R. [2002]. A Fuzzy-Based Intelligent Navigation System for Mobile Robots. Proceedings of the 3rd International Symposium on Robotics and Automation, Mexico, 2002. Pp. 186–191.

Brooks, R.A. [1986]. A robust layered control system for a mobile robot. IEEE Journal of Robotics and Automation RA-2.

Brooks, R.A. [1991]. Integrated systems based on behaviors. SIGART Bulletin, 2(4), pp. 46–50.

Billard, A. and Hayes, G. [1997]. Robot’s first steps, Robot’s first words. Proceedings of the GALA97 conference, Groningen, Assembly on Language Acquisition, Edingburg, April 4–6, 1997.

Chen, B. and Xu, Y. [1988]. ZDRL: Motion-Oriented Robot Programming Language. Proceedings of the 1988 IEEE International Conference on Systems, Man and Cybernetic, Volume 2. August 1988. Pp. 1226–1229.

Choi, B. and Chen Y. [2002]. Humanoid Motion Description Language. Proceedings of the Second International Workshop on Epigenetic Robotics, Edinburgh, Scotland, August 10–11, 2002. 4 p.

Chung, Y., Park, C. and Harashima, F. [2001]. A Position Control Differential Drive Wheeled Mobile Robot. IEEE Transactions on Industrial Eelectronics, Volume 48, Issue 4, August 2001, pp. 853–863.

Dallaway, J.L. and Jackson, R.D. [1994]. The User Interface for Interactive Robotic Workstations. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems ’94, Volume 3, September 1994. Pp. 1682–1686.

Diequez, A.R., Raimúndez, C., Sanz, R., Lopez, J. and Delgado, E. [1995]. An Intelligent Supervisory Model for Path Planning and Guidance of Mobile Robots in Non-structured Environments. Proceedings of the 2nd IFAC Conference on Intelligent Autonomous Vehicles 95, Espoo, Finland, 12–14 June, 1995. Pp. 81–86.

Drews, P. and Fromm, P. [1997]. A Natural Language Processing Approach for Mobile Service Robot Control. Proceedings of the 23rd International Conference on Industrial Electronics, Control and Instrumentation (IECON 97), Vol 3, November 1997. Pp. 1275–1277.

Elmaghraby, A.S. [1988]. A Robot Control Language. Proceedings of the Southeast Conference '88, IEEE, April 11–13, 1988. Pp. 413–416.

Evans, J. [1994]. Helpmate: An Autonomous Mobile Robot Courier for Hospitals. Proceedings of the International Conference on Intelligent Robots and Systems, 1994. Pp. 1695–1700.

Fakta 2001 [1981]. 8. osa, WSOY, Porvoo, 1981.

Fakta 2001 [1985]. 13. osa, WSOY, Porvoo, 1985.

Feng, L., Koren, Y. and Borenstein, J. [1994]. A model-reference adaptive motion controller for a differential-drive robot. Proceedings of the IEEE International Conference on Robotics and Automation. 8–13 May 1994, Volume 4. Pp. 3091–3096.

Firby, R.J. [1987]. An investigation into reactive planning in complex domains. In Sixth National Conference on Artificial Intelligence, Seattle, WA, July 1987. AAAI.

Firby, R.J. [1989]. Adaptive execution in complex dynamic worlds. Technical Report YALEU/CSD/RR #672, Computer Science Department, Yaly University, January 1989.

Firby, R.J. [1994]. Task Networks for Controlling Continuous Processes. In Proceedings of the Second International Conference on AI Planning Systems, 1994. Pp. 49–54.

Fleureu, J.L., Le Rest, E. and Marce, L. [1995]. PILOT: A Language for Planning Mission. Proceedings of the 2nd IFAC Conference on Intelligent Autonomous Vehicles 95, Espoo, Finland, June 12–14, 1995. Pp. 386–391.

Freund, E., Lüdemann-Ravit, B, Stern, O. and Koch, T. [2001]. Creating the Architecture of a Translator Framework for Robot Programming Languages. Proceedings of the 2001 IEEE International Conference on Robotics & Automation, Seoul, Korea, May 21–26, 2001. Pp. 187–192.

Fraser, R.J.C. and Harris, C.J. [1991]. Infrastructure for Real-time Robotic Control: Aspects of Robot Command Language. IEEE Colloquim on Intelligent Control, 19 February 1991.

Gat, E. [1997]. ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents. Proceedings of the IEEE Aerospace Conference, 1997.

Grosskreutz, H. and Lakemeyer, G. [2000]. Towards more realistic logic-based robot controllers in the GOLOG framework. Korrekturabzug, Künstliche Intelligenz, Heft 4/00, arenDTaP Verlag, Bremen. Pp. 11–15. ISSN 0933-1875

Halme, A. [2003]. Lecture of prof. Halme on Service and Field Robotics. http://www.automation.hut.fi/edu/as84145/luennot.html, Retrieved 5.9.2003.

Halme, A, Leppänen, I., Ylönen, S. and Kettunen, I [2001a]. Workpartner – centaur like service robot for outdoor applications. Proceedings of the 3rd International Conference on Field and Service Robotics, June 11–13, 2001, Finland. Pp. 217–223.

Halme, A., Leppänen, I., Montonen, M. and Ylönen, S. [2001b]. Robot motion by simultaneously wheel and leg propulsion. Proceedings of the 4th International Conference on Climbing and Walking Robots, September 24–26, 2001. Pp. 1013–1019.

Ingrand, F.F., Chatila, R., Alami, R. and Robert, R. [1996]. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. Proceedings of the 1996 IEEE International Conference on Robotics and Automation, Minneapolis, 1996.

Kamada, T. and Oikawa, K. [1998]. AMADEUS: A mobile, Autonomous Decentralized Utility System for Indoor Transportation. Proceedings of the 1998 IEEE International Conference on Robotics & Automation, Leuven, Belgium, May 1998. Pp. 2229–2236.

Kanayama, Y. [1989]. Locomotion Functions for a Robot Language. Proceedings of the International Workshop on Intelligent Robots and Systems ’89, IEEE/RSJ, September 4–6, Japan, 1989. Pp. 542–549.

Kauppi, I., Blom, M., and Lehtinen, H. [2001]. Motion Control Language for Mobile Robots. Proceedings of the 3rd International Conference on Field and Service Robotics FSR2001, June 11–13, Finland, 2001. Pp. 43–48.

Kawamura, K., Peters, R.A., Johnson, C., Nilas, P. and Thongchai, S. [2001]. Supervisory Control of Mobile Robots using Sensory EgoSphere. Proceedings of 2001 IEEE Enternational Symposium on Computational Intelligence in Robotics and Automation, July 29 – August 1, Canada, 1994. Pp. 523–529.

Kirchner, W.H. and Towne, W.F. [1994]. The Sensory Basis of the Honeybee’s Dance Language. Scientific American, June 1994. 7 p.

Kuhnhen, M. [2002], Entertainmet robotics – Are we being taken for a ride? Proceedings of the 33rd International Symposium on Robotics, Stockholm, Sweden, October 2002, CD-rom.

Kulyukin, V. and Steele, A. [2002]. Introduction and Action in the Three-Tiered Robot Architecture. Proceedings of the 3rd International Symposium on Robotics and Automation, Toluca, Edo. Méx., México, September 2002. Pp. 578–583.

Lakehal, B., Amirat, Y. and Pontnau, J. [1995]. Fuzzy steering control of a mobile robot. International IEEE/IAS Conference on Industrial Automation and Control: Emerging Technologies, 22–27 May 1995. Pp. 383–386.

Lanzoni, C, Sánches, A. and Zapata, R. [2002]. A Single-Query Motion Planner for Car-like Nonholonomic Mobile Robots. Proceedings of the 3rd International Symposium on Robotics and Automation, Toluca, Edo. Méx., México, September 2002. Pp. 267–274.

Lauria, S., Bugmann, G., Kyriacou, T., Bos, J., and Klein E. [2001]. Training Personal Robots Using Natural Language Instruction. Intelligent Systems, IEEE, Volume 16, Issue 5, Sep/Oct 2001. Pp. 38–45.

Lauria, S., Bugmann, G., Kyriacou, T., Bos, J. and Klein, E. [2002]. Converting Natural Language Route Instructions into Robot Executable Procedures. Proceedings of the 2002 IEEE International Workshop on Robot and Human Interactive Communication, Berlin, September 25–27, 2002. Pp. 223–228.

Lehtinen, H., Kaarmila, P., Blom, M. and Kauppi, I. [2000]. Mobile Robots Evolving In Industrial Applications. Proceedings of the International Symposium on Robotics (ISR 2000), Montreal, May 2000. Pp. 96–101.

Leifer, L., Van der Loos, M. and Lees, D. [1991]. Visual Language Programming: for robot command • control in unstructured environments. Proceedings of the Fifth International Conference on Advanced Robotics (91 ICAR), Vol 1., June 1991. Pp. 31–36.

Levesque, H.J., Reiter, R., Lesperance, Y. and Scherl, R.B. [1997]. GOLOG: A Logic Programming Language for Dynamic Domains. Journal of Logic Programming, 31(1–3), pp. 59–83.

Lopes, L.S. and Teixteira, A.J.S. [2000]. Human Robot Interaction through Spoken Language Dialogue. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE CS Press, Los Alamitos, California, 2000.

Makatchev, M. and Tso, S.K. [2000]. Human-Robot Interface Using Agents Communicating in a XML-Based Markup Language. Proceedings of the IEEE International Workshop on Robot and Human Interactive Communication ROMAN 2000, Osaka, Japan, September 27–29, 2000. Pp. 270–275.

Miller, D.J. and Lennox, R.C. [1990]. An Object-Oriented Environment for Robot System Architectures. Proceedings of the IEEE International Conference on Robotics and Automation, Cincinnati, OH, August 13–16, 1990. Pp. 14–23.

Mäkelä, H. [2001]. Overview of LHD navigation witout artificial beacons. Robotics and Autonomous Systems 36 (2001), pp. 21–35.

Nilsson, N.J. [1984]. Shakey the Robot. SRI AI Center, Technical Report 323, April, 1984.

North, S. and Hermans, P. [2001]. XML Trainer Kit. IT Press, 2000.

Nishiyama, H., Ohwada, H. and Mizoguchi, F. [1998]. A Multiagent Robot Language for Communication and Concurrency Control. Proceedings of the International Conference on Multiagent Systems, 3–7 July 1998. Pp. 206–213.

Paoletti, J.C and Marce, J. [1990]. A monitoring language for telerobotics applications. First International Symposium on Measurement and Controllingf Robotics, ISMRC'90, ROBEX,90, NASA, Houston, Texas, 1990.

Paul, R.P. [1977]. WAVE: A Model-Based Language for Manipulator Control. The Industrial Robot, Vol. 4, No. 1, 1977, pp. 10–17.

Pembeci, I., and Hager, G. [2001]. A Comparative Review of Robot Programming Languages, August 14, 2001.
Available from http://citeseer.nj.nec.com/555282.html, Retrieved 10.9.2003.

Prassler, E., Scholz. J and Fiorini, P. [2001]. A Robotic Wheelchair for Crowded Public Environments. IEEE Robotics & Automation Magazine, Vol. 8, Issue 1, March 2001, pp. 38–45.

Prassler, E., Stroulia, E. and Strobel, M. [1997]. Office Waste Cleanup: An Application For Service Robots. Proceedings of the the 1997 IEEEE International Conference on Robotics and Automation, Albuquerque, New Mexico, April 1997. Pp. 1863–1868.

Segovia, A., Rombaut, M., Preciado, A. and Meizel, D. [1991]. Comparative study of the different methods of path generation for a robot in a free environment. International Conference on Advanved Robotics, 'Robots in Unstructured Environments', 91 ICAR., Vol. 2, 1991. Pp. 1667–1670.

Selkäinaho, J. [2002]. Adaptive autonomous navigation of mobile robots in unknown environments. Helsinki University of Technology, Automation Technology Laboratory, Series A: Research Reports No. 24, December 2002. Dr. Thesis. 88 p.

Sievilä, J. [2003]. Palvelurobotin suorittama etsintä- ja tunnistustehtävä. Teknillinen Korkeakoulu, Automaatio- ja systeemitekniikan osasto. Diplomityö. 60 p.

Sitharama Iyengar, S. and Elfes, A. [1991]. Autonomous Mobile Robots: Perception, Mapping and Navigation. IEEE Computer Society Press, Los Alamos California, 1991. Pp. 428–436.

Simmons, R. and Apfelbaum, D. [1998]. A Task Description Language for Robot Control. Proceedings of the Conference on Intelligent Robotics and Systems, Vancouver, October 1998.

Stroulia, E. and Goel, A. [1999]. Evaluating PSMs in Evolutionary Design: The Autognostic experiments. International Journal of Human Computer Studies, 51, pp. 825–847.

Thrun, S., Bennewitz, M., Burgard, W., Cremers, A., Dellaerr, F., Fox, D., Hahnel, D., Rosenberg, C., Schulte, J. and Schulz, D. [1999] Minerva: A second-generation museum tour-guide robot. In Proceedings of the IEEE International Conference on Robotics Automation (ICRA). Pp. 1999–4005.

Tiedon maailma 2000 [1999]. Otava, Hongkong. 476 p.

Turchin, V. [1997]. Language. *In*: Heylighen, F., Joslyn, C. and Turchin, V. (editors). *Principia Cybernetica Web* (Principia Cybernetica, Brussels), URL:http://pesmc1.vub.ac.be/LANG.html.

Unece and IFR [2002], World Robotics 2002 – Statistics, market analysis, forecasts, case studies and pfofitability of robot investment. United Nations, (ISBN 9211010470, ISSN 10201076).

Vidal-Calleja, T.A., Velasco-Villa, M. and Aranda-Bricaire, E. [2002]. Real-Time Obstacle Avoidance for Trailer-Like Systems. Proceedings of the 3rd International Symposium on Robotics and Automation, Mexico, 2002. Pp. 131–136.

Voudouris, C., Chernett, P., Wang, C.J. and Callaghan, V.L. [1995]. Hierarchical behavioural control for autonomous vehicles. In Proceedings of the 2nd IFAC conference on Intelligent Autonomous Vehicles 95, Espoo, Finland. Pp. 267–272.

Yanco, H. [1992]. Towards an adaptable robot language. In Intelligent Applications of Artificial Intelligence to Real World Robots: Papers from the 1992 Fall Symposium, AAAI Technical Report FS-92-02, Boston, MA, October 1992. Pp. 191–194.

Yong, L.S., Yang, W.H. and Ang, M. [1998]. Robot Task Execution with Telepresense Using Virtual Reality Technology. Proceedings of the International Conference on Mechatronic Technology, Taiwan, 1998.

WWW-reference 1, http://www.automower.com/, Retrieved 9.9.2003

WWW-reference 2, http://trilobite.fi.electrolux.com, Retrieved 9.9.2003

WWW-reference 3, http://www.eu.aibo.com/, Retrieved 9.9.2003

WWW-reference 4, http://dmoz.org/Computers/Programming/Languages/, Retrieved 12.9.2003

WWW-reference 5, ABB Flexible Automation. RAPID Reference Manual. S-72168 Västerås, Sweden. Art. No. 3HAC 7783-1. http://rab.ict.pwr.wroc.pl/irb1400/datasys_rev1.pdf, 19.5.2003. 583 pages.

WWW-reference 6, http://www.informatic.uni-bonn.de/~rhino/docs/rpl-manula/node2.html, Retrieved 10.9.2003

WWW-reference 7, http://www.iredes.org/, Retrieved 12.9.2003

WWW-reference 8, http://www.cs.utexas.edu/users/qr/robotics/splat/, Retrieved 12.9.2003

WWW-reference 9, http://www.sas.be/famous /famous.htm, Retrieved 12.9.2003

Published by

VTT

Series title, number and report code of publication

VTT Publications 510
VTT–PUBS–510

Author(s)
Kauppi, Ilkka

Title

Intermediate Language for Mobile Robots
A link between the high-level planner and low-level services in robots

Abstract
The development of service and field robotics has been rapid during the last few decades. New versatile and affordable sensors are now available, and very importantly, computing power has increased very fast. Several intelligent features for robots have been presented. They include the use of artificial intelligence (AI), laser range finders, speech recognition, and image processing. This all has meant that robots can be seen more frequently in ordinary environments, or even in homes.

Most development work has concentrated on a single or a few sophisticated features in development projects, but even work to design control structures for different levels in robot control has been done. Several languages for industrial and mobile robots have been introduced since the first robot language WAVE was developed in 1973. Tasks can be given to robots in these languages, but their use is difficult and requires special skills of users.

In the future, robots will also be used in homes, and ordinary people should be able to give tasks for robots to perform. This should be done descriptively using natural language as in describing tasks to another person.

In this work an intermediate language for mobile robots (ILMR) has been presented. It makes it easier to design a new task for a robot. ILMR is intended for service and field robots and it acts as an intermediate link from user, an intelligent planner or a human-robot interface to a robot's actions and behaviours. The main principle in development work has been simplicity and ease of use. Neither any deep knowledge of robotics nor good programming skills are required when using ILMR. While easy to use, ILMR offers all the required features that are needed to control today's specialised service and field robots. These features contain sequential and concurrent task execution and response to exceptions. ILMR also makes it easier to manage the development of complicated software projects of service robots by creating easy-to-use interfaces to all of several subsystems in robots.

It is possible for users to use ILMR to give direct commands or tasks to a robot, but it is intended to be used with higher-level abstract languages, such as sign language or even natural spoken language through a high level planner. An action in ILMR can be given coarsely, i.e. in an abstract way, or in detail. Due to this coarseness, ILMR is suitable to be used with higher-level abstract languages and the set of elementary commands supports directly the use of natural language. With ILMR no complicated models of robots and the world are needed. Only a few measureable parameters for robots are needed and a simple map of the environment is maintained.

ILMR has been implemented in two different kinds of robots, and its use and performance has been studied with simulators and actual robots in a wide variety of tests. The structure and operation of ILMR has proved to be useful and several tasks have been carried out successfully using both test robots.

Keywords
mobile robot languages, industrial robot languages, autonomous service robots, autonomous field robots, natural language control, field robotics, high-level languages

Activity unit
VTT Industrial Systems, Otakaari 7 B, P.O.Box 13022, FIN–02044 VTT, Finland

ISBN 951–38–6251–8 (soft back ed.) 951–38–6252–6 (URL:http://www.vtt.fi/inf/pdf/)			Project number
Date November 2003	Language English	Pages 143 p.	Price C
Name of project		Commissioned by	
Series title and ISSN VTT Publications 1235–0621 (soft back ed.) 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)		Sold by VTT Information Service P.O.Box 2000, FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374	

VTT PUBLICATIONS 510 Intermediate Language for Mobile Robots. A link between the high-level planner and ... Ilkka Kauppi

In the future, robots will be used in homes, and ordinary people should be able to give tasks for robots to perform. This should be done descriptively using natural language as in describing tasks to another person. In this work an intermediate language for mobile robots (ILMR) has been presented. It makes it easier to design a new task for a robot. ILMR is intended for service and field robots and it acts as an intermediate link from user, an intelligent planner or a human-robot interface to a robot's actions and behaviours. The main principle in development work has been simplicity and ease of use. Neither any deep knowledge of robotics nor good programming skills are required when using ILMR. While easy to use, ILMR offers all the required features that are needed to control today's specialised service and field robots. These features contain sequential and concurrent task execution and response to exceptions. ILMR also makes it easier to manage the development of complicated software projects of service robots by creating easy-to-use interfaces to all of several subsystems in robots.

Tätä julkaisua myy	Denna publikation säljs av	This publication is available from
VTT TIETOPALVELU PL 2000 02044 VTT Puh. (09) 456 4404 Faksi (09) 456 4374	VTT INFORMATIONSTJÄNST PB 2000 02044 VTT Tel. (09) 456 4404 Fax (09) 456 4374	VTT INFORMATION SERVICE P.O.Box 2000 FIN–02044 VTT, Finland Phone internat. +358 9 456 4404 Fax +358 9 456 4374

ISBN 951–38–6251–8 (soft back ed.)
ISSN 1235–0621 (soft back ed.)

ISBN 951–38–6252–6 (URL: http://www.vtt.fi/inf/pdf/)
ISSN 1455–0849 (URL: http://www.vtt.fi/inf/pdf/)