Aalto University
Automation Technology
Series A: Research Reports No. 35
Espoo, November 2010

ActionPool: A NOVEL DYNAMIC TASK SCHEDULING METHOD FOR SERVICE ROBOTS

Tapio Taipalus

Doctoral dissertation for the degree of Doctor of Science in Techology to be presented with due permission of the Faculty of Electronics, Communications and Automation for public examination and debate in Auditorium AS1 at the Aalto University School of Science and Technology (Espoo, Finland) on the 12th of November 2010 at 12 noon.

Aalto University
School of Science and Technology
Faculty of Electronics, Communications and Automation
Department of Automation and Systems Technology

Distribution:

Aalto University
School of Science and Technology
Faculty of Electronics, Communications and Automation
Department of Automation and Systems Technology
P.O. BOX 11000
FI-00076 AALTO
FINLAND

e-mail: tapio.taipalus@tkk.fi
Tel. +358 9 470 23778
Fax +358 9 470 25142

ISBN 978-952-60-3429-4 (printed)
ISBN 978-952-60-3430-0 (pdf)
ISSN 0783-5477

Aalto University

ABSTRACT OF DOCTORAL DISSERTATION	AALTO UNIVERSITY SCHOOL OF SCIENCE AND TECHNOLOGY P.O. BOX 11000, FI-00076 AALTO http://www.aalto.fi

Author Tapio Taipalus

Name of the dissertation

ActionPool: A Novel Dynamic Task Scheduling Method for Service Robots

Manuscript submitted 14.6.2010	Manuscript revised 28.9.2010
Date of the defence 12.11.2010	
☒ Monograph	☐ Article dissertation (summary + original articles)

Faculty	Faculty of Electronics, Communications and Automation
Department	Department of Automation and Systems Technology
Field of research	Service robotics
Opponent(s)	Doctor of Science Hannu Lehtinen
Supervisor	Professor Aarne Halme
Instructor	Professor Aarne Halme

Abstract

Service robots require the seamless utlisation of several technical disciplines. Most of the required technologies are sufficiently advanced to provide feasible solutions to be used in the designing of service robots. For instance, mechanical engineering, control theory, electronics and electrical engineering aspects of the design have all matured well. On the other hand, it is the perception and artificial intelligence that provide the means for modelling the environment and the knowledge which are lagging behind. The latter two disciples in their current state, greatly limit the complexity of the tasks which can be performed by service robots.

In this thesis, an ActionPool method for representing task knowledge and executing multiple tasks simultaneously with service robots is presented. The method is based on a concept in which the actions that are ready for execution are placed into a pool and from those most suitable for the situation are selected one by one. The number of actions in a pool and the number of tasks are limited only by the available computational resources.

The actions can belong to different tasks, and thus the action pool allows the robot's indivisible resource to be dynamically dealt out for various tasks requiring the resources. In the ActionPool method, the functional parts of the service robot are divided into resources and an action pool is assigned to each one of them. This way, numerous tasks can be executed simultaneously. The ActionPool method allows a natural way of dynamically adding and removing tasks to and from the robot's active execution. The action selection method can direct the perception processes to observe the relevant parts of the environment.

The ActionPool method has been implemented on two different service robot platforms to verify the generic nature of the method. Several tasks have been executed successfully to validate the claims about the qualities of the method. Compared to previous approaches, this work provides a fresh execution- and contingency-centric vantage point to the well studied robot control problem.

Keywords service robot, multi-tasking, knowledge representation	
ISBN (printed) 978-952-60-3429-4	ISSN (printed) 0783-5477
ISBN (pdf) 978-952-60-3430-0	ISSN (pdf)
Language English	Number of pages 170
Publisher Aalto University, Department of Automation and Systems Technology	
Print distribution Aalto University, Department of Automation and Systems Technology	
☒ The dissertation can be read at http://lib.tkk.fi/Diss/2010/isbn9789526034300/	

Aalto-yliopisto

VÄITÖSKIRJAN TIIVISTELMÄ	AALTO-YLIOPISTO TEKNILLINEN KORKEAKOULU PL 11000, 00076 AALTO http://www.aalto.fi

Tekijä Tapio Taipalus

Väitöskirjan nimi

ActionPool: Dynaamista tehtävien ohjausta palveluroboteille

Käsikirjoituksen päivämäärä 14.6.2010	Korjatun käsikirjoituksen päivämäärä 28.9.2010
Väitöstilaisuuden ajankohta 12.11.2010	
☒ Monografia	☐ Yhdistelmäväitöskirja (yhteenveto + erillisartikkelit)

Tiedekunta	Elektroniikan, tietoliikenteen ja automaation tiedekunta
Laitos	Automaatio- ja systeemitekniikan laitos
Tutkimusala	Palvelurobotiikka
Vastaväittäjä(t)	Tekniikan tohtori Hannu Lehtinen
Työn valvoja	Professori Aarne Halme
Työn ohjaaja	Professori Aarne Halme

Tiivistelmä

Palvelurobotit ovat vähitellen siirtymässä tutkimuslaboratorioista ja teollisuusympäristöistä kuluttajille ja samaan ympäristöön ihmisten kanssa. Monella alalla teknologia on jo kypsää todella hyödyllisille ja ihmisten aikaa säästäville palveluroboteille. Konenäkö, sekä havainnointi yleensä, ja tekoäly eivät vielä osaa mallintaa robotin dynaamista ympäristöä kovin hyvin. Monimutkaisempia tehtäviä varten palvelurobotin tulee lisäksi mallintaa omat tehtävänsä.

Tässä työssä esitellään ActionPool metodi, mikä on tapa mallintaa, esittää ja suorittaa monia tehtäviä samanaikaisesti palvelurobotilla. Metodi perustuu robotin jakamiseen loogisiin mekaanisiin kokonaisuuksiin, resursseihin, ja niiden hallintaan. Suoritettavat tehtävät jaetaan toimenpiteisiin robotin eri resursseille.

Jokaisella robotin resurssilla on pooli, mihin suoritukseen valmiit toimenpiteet eri tehtävistä listataan. Jokaisen toimenpiteen tai ympäristössä tapahtuvan merkittävän muutoksen jälkeen valitaan tilanteeseen parhaiten sopiva toimenpide. Tehtävän edetessä siirretään uusia toimenpiteitä pooliin suoritusta varten.

Ympäristöä havainnoidaan rinnakkaisena prosessina ja toimenpiteen valinta ohjaa havaintoja ympäristön merkityksellisiin osiin. Esitetty ActionPool metodi sallii robotin eri osien käytön samanaikaisesti eri tehtävissä. Metodi ei rajoita toimenpiteiden lisäystä tai poistoa poolista, mikä tarkoittaa sitä, että tehtäviä voidaan lisätä tai poistaa robotille dynaamisesti tarpeen mukaan. Tämä mahdollistaa luonnollisen ja joustavan tavan ohjata robotin tehtävien suoritusta.

Esitetty metodi on toteutettu kahdelle eri palvelurobotille. Metodin yleistettävyyden osoittamiseksi tehtiin useita kokeita menestyksekkäästi kumpaakin robottia käyttäen. Tämä työ antaa uuden tehtävänsuoritus- ja poikkeustapauskeskeisen lähestymistavan muihin esitettyihin ratkaisuihin nähden.

Asiasanat palvelurobotti, moniajo, tietämyksen esitys

ISBN (painettu) 978-952-60-3429-4	ISSN (painettu) 0783-5477
ISBN (pdf) 978-952-60-3430-0	ISSN (pdf)
Kieli englanti	Sivumäärä 170

Julkaisija Aalto-yliopisto, Automaatio- ja systeemitekniikan laitos

Painetun väitöskirjan jakelu Aalto-yliopisto, Automaatio- ja systeemitekniikan laitos

☒ Luettavissa verkossa osoitteessa http://lib.tkk.fi/Diss/2010/isbn9789526034300/

Preface

Working on this doctoral thesis has been one of the most challenging yet most rewarding tasks I have ever completed. The journey has been long with numerous ups and downs and along the way I learnt many useful things about life, academic work, researching, and nature of knowledge.

I would like to extend my deepest gratitude to my supervisor, Professor Aarne Halme, for guiding me throughout my doctoral studies and believing in my work and its outcome.

I thank the pre-examiners of this dissertation, Professor Erwin Prasler and Docent Tapio Heikkilä, for their valuable comments and insights. Based on their feedback, I was able to enhance the structure and the overall quality of this work.

I am grateful to Professor Kazuhiro Kosuge from Tohoku Univeristy, Japan, for his support and insightful comments and for allowing me to use the research and experimental facilities in his laboratory for this work.

I sincerely appreciate the guiding role of Docent Mika Vainio, particularly during the last few months prior to the submission of this dissertation. I am grateful for his time and effort in proof reading my thesis time and time again.

Also a big thank you to all the members of the Automation Technology Laboratory for creating an inspiring working environment, sharing all the minute details of Latex and programming tricks, proof reading my work, and generally making the daily work life a lot more pleasant.

A big thank you to my lovely wife, Hanieh Taipalus, for her patience and mental support after those long working days, for her love and understanding and for being there when I needed a shoulder to lean on and an ear to share my thoughts with. Last by not least, thank you to our beautiful daughter for bringing so much joy and laughter into our lives.

Finally, I would like to express my gratitude to the Academy of Finland for financially supporting me during the various stages of this work.

Espoo, October 2010

Tapio Taipalus

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
AP	Action pool
aPlan	Action plan
BDI	Belief Desire Intention
CAD	Computer Aided Design
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DARPA	Defence Advanced Research Projects Agency
DB	Database
DDD	Dirty, Dangerous, or Dull
DOF	Degree Of Freedom
EL	Event Listener
FSA	Finite State Automaton
GCD	Grand Central Dispatch
GIM	Generic Intelligent Machine
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HMI	Human-Machine Interface
HTN	Hierarchical Task Network
IPC	Inter-Process Communication
IR	Infra-Red

MDP	Markov Decision Process
PC	Personal Computer
POMDP	Partially Observable Markov Decision Process
PRS	Procedural Reasoning System
PTU	Pan Tilt (Zoom) Unit
RAM	Random Access Memory
RAP	Reactive Action Packages
SPA	Sense Plan Act
tPlan	Task plan
UN	United Nations
uTask	Micro Task
VFH	Vector Field Histogram
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

Glossary

Action A piece of work without purpose by itself. In the ActionPool context an Action is an indivisible piece of work utilising a resource.

- **Placeholder** ~ A method to keep a Task from advancing while there is some process pending. The placeholder Action can not be selected for execution, but it can be removed to cancel the process that is pending.
- **Monitored** ~ A type of remote Action whose execution is monitored from the original Action pool.
- **Mutual** ~ A type of remote Action to ensure the simultaneous use of the original and remote resource.
- **Remote** ~ An Action initiated from a different Action pool from the one where it is executed.
- **Unmonitored** ~ A type of remote Action whose execution is not monitored from the original Action pool.

Action plan (aPlan) A plan how to do Action.

Action pool (AP) An abstract data-structure and software agent to manage a resource. It holds a list of Actions that are ready to be executed by a resource.

ActionPool A dynamic task cheduling method utilizing Action pools.

Database (DB) A data collection where the information can be stored or retrieved. In the ActionPool context DB holds the robot's world model.

Event Listener A software agent that reacts to a change in some variable exceeding a designated threshold.

Knowledge Formalised information

Perception Agent A software agent that reads raw data from sensors and outputs object observations into the database.

Plan Detailed instructions for how to conduct an activity such as a Task or Action.

Planning A process to generate a plan by predicting the consequences of the operations the plan is constructed from.

Micro Task (uTask) An indivisible piece of work with a robot.

Multi-tasking Doing multiple tasks with different objectives.

- **Concurrent** ~ Multi-tasking simultaneously by the utilising of different resources by the same entity.
- **Parallel** ~ Multi-tasking simultaneously by the utilising of parallel resources by the same entity.
- **Time-sharing** ~ Multi-tasking simultaneously by dividing the usage of a critical resource by turns.

Task A piece of work with a purpose by itself.

Task Configuration A process of constructing a task from a goal, plan, and metadata.

Task plan (tPlan) A plan how to do Task.

Chapter 1

Introduction

1.1 Motivation and Background

Robotics has been fascinating people for a long time. A general machine that can work for us and serve us. A machine that can think for us or with which we can have a conversation. An artificial person with its own thoughts and motives. People's imaginations can go far beyond the level of technology and, in some rare cases, vice versa. Humans tend to give personalities and feelings even to inanimate objects and stochastic processes in everyday life. One can imagine a sick old car that breaks down often or a stubborn golf ball that does not want to go into the tiny hole designed for it.

When we imagine a machine that is purpose-built to think and has feelings and has a resemblance to a human or animal, expectations can get quite high. Unfortunately, technology can not meet these expectations yet. Machine perception is very far from understanding our world and drawing intelligent conclusions. Let us take an extreme example from the film "Planet of the Apes" (1968). The main character travelled in space and ended up on a habitable planet where humans were treated like animals and apes ruled over them. At the end of the film, the main character finds one partially occluded piece of evidence: a part of the "Statue of Liberty" (Figure 1.1) and suddenly viewers can draw very far-reaching conclusions about past events, the current location, and a myriad other things. With a machine with today's technology we would have a hard time recognising the statue, let alone ways to associate and store information for drawing the conclusions.

But, still we believe that it is possible to make some rational decisions with machines and develop a useful generic service robot with today's technology, while bearing in mind the current limitations.

The usage of service robots and the market for them have been strong and they are expected to grow further in the future [1], [2]. In 2008 the sales of service robots for professional use came to 11.2 billion US dollars and 7.2 million units were sold for personal and private use world-wide. This trend is also recognised by many governments. There is, additionally, the problem of an aging population, for which robotics and home automation are considered a solution by many gov-

Figure 1.1: *Screen shot from the film "Planet of the Apes"*

ernments. Japan is investing in robotics research through university and research institute funding. Furthermore, large Japanese enterprises, such as Honda, Sony, Panasonic, Toyota, or Mitsubishi, are invesing heavily in robotics. For example, Panasonic expects sales of 100 billion yen (890 million Euros) in robotics by 2015 [3]. The Korean government launched a huge service robot research programme in 2009 worth 1 trillion Won (660 million Euros) along with the Robot Land theme park and investments in the private sector [1]. In the USA, the government-funded research has more of a militarily inclined aim, with obviously large but somewhat undisclosed budgets. Their focus spans from unmanned air, ground, and sea wehicles (which are already in use) to research led by DARPA (Defense Advanced Research Projects Agency) into further automated unmanned vehicles [4] and robotised medical evacuation and trauma treatment [5], among others.

Automation has revolutionised the production industry during recent decades. Service robotics could be seen as this automation technology's revolution affecting the service-providing industry too. Service robots have great potential to improve the productivity of a whole nation so that the human capital working in the service sector would instead be freed to do something else; one hopes that this would be something more rewarding, productive, and inspiring. Automation has not only lowered costs in the production industry but has also made the quality of products more predictable and stable. Sometimes, the quality has even surpassed manual manufacturing. Without a doubt, these effects can also be seen in the service industry, too. For example, when did you last buy an airline ticket over the counter from a clerk? Travel agencies can offer better deals for the customer faster and independently of the office location, with a virtual agent browsing through offers from different airlines via the Internet. The only question is when something similar will be done with physical agents.

At the beginning, multi-tasking service robots would be very expensive, as always when a new technology is introduced. This was, and still is, to some extent, the case in the production industry. Because of the high cost, only parties with enough capital can utilise the technology commercially and prosper from it. That can be seen as an advantage or disadvantage of automation, depending on your po-

litical stance. There is definitely one bad side to the development of automation. People freed by the technology cannot always find something else to do, leading to a rise in unemployment. However, in the home environment the service robot has the potential to take over the household chores and release unpaid human capital, which is always welcomed.

It is expected that with the advancement of the required technology and also mass production the cost of a service robot will be reduced drastically. A service robot of the future would be affordable enough to be a personal possession similar to today's cars, of course, with the assumption that the robot is able to perform functions that can provide more freedom and free time for its owner, as a car does.

1.2 Problem statement

This research work started from the question of how robots could learn complex tasks and improve their task execution in time. During the research it became clear that there was no sufficiently abstract and flexible way to represent the task knowledge. The representation has to be abstract in order to minimise the search space for any kind of learning or optimisation algorithm.

Thus, in this research, a concise way to represent a task for a service robot is studied. The control of a service robot is a very complex endeavour and a considerable part of the effort goes into defining and refining how the tasks should be performed. So, the main problem tackled in this work could be formulated into the following question: How can a task be represented to a service robot in a concise way and how can they be executed simultaneously?

To answer this question, three fundamental aspects [6] should be defined before the solution or method is proposed: 1) the available hardware; 2) the task to be performed, and 3) the operating environment. The case considered in this research is a single mobile manipulator kind of service robot. The robot executes multiple varying tasks initiated by a human operator. The operational environment is unstructured, dynamic, and also occupied by humans. The main focus of this work is in the representation of the task knowledge, since it affects the usability of the robot so strongly. But the methodology for the execution of the task is so intertwined with the representation that it cannot be omitted.

1.3 Contributions

This research is in the field of the control of service robots. In this work the so-called ActionPool method was created. The two main contributions of the ActionPool method are as follows.

1. A new way of abstracting the functions to execute a task

   In this work the task is divided into elements called actions. An action is considered as a unit operation for the robot on a particular resource. In the previous works, either the granularity of the unit operation has been much smaller or the size has been completely arbitrary. The representation also has an event listener component that describes the operation sequenced by events that are uncontrollable for the robot or in the case of an exception. In this work a task-centric bottom-up approach from the execution point of view has been taken, i.e. what the robot should do to execute a task successfully and how to describe that in the plan. In the previous works the approach has been a plan- or planning-centric top-down approach, i.e. what would be the most convenient way to manage the plan or create the plan automatically?

2. A method is developed to execute multiple tasks concurrently while utilising different resources of the robot.

   Traditionally, a particular single task has dominated the usage of the robot. The development has concentrated on conducting tasks one by one. If the tasks are mixed, that is known beforehand and the interactions between the tasks are carefully planned before anything is done. The work presented in this thesis divides the functional parts of the robot into interdependent resources instead of considering the robot as a single resource. The division of tasks into actions allows the dynamic sharing of a resource between different tasks and thus no planning is required.

To the best of the author's knowledge, there is no other control method for a service robot that allows the dynamic adding or removing of tasks for execution which has been demonstrated to work on a real robot in a dynamic environment.

The proposed representation allows the incremental configuration of a task. Tasks can be developed independently but executed at the same time. The representation is on an abstract level and thus tasks can be interchangeable between different kinds of robots. Additionally, the task representation becomes very compact and natural for humans. Furthermore, the compact representation makes the search space smaller for various learning algorithms.

1.4 Outline of the Thesis

In this chapter, an introduction to the research field was provided, along with a motivation and problem statement. In the next chapter a more elaborate and focused

explanation of the research field is given. The description of the state of the art is divided into two chapters. First, the different ways to execute a task with a service robot are shown in Chapter 3. In Chapter 4, the state of the art regarding different ways to represent knowledge about the task are considered.

Chapter 5 provides details of the ActionPool method followed by a chapter explaining the implementation of the ActionPool method. The experimental validation of the ActionPool method is shown in Chapter 7. In Chapter 8, the results of the experiments are analysed further and compared to existing methods. Finally, in the last chapter the significance of this work is discussed, along with future work and other considerations related to this work and its applicability.

1.5 Author's Contribution within the Research Groups

The work started with studying the basics of task execution with service robots within the large research group in the WorkPartner project [7, 8]. The project lasted from 1998 until the first quarter of 2006 and the author was privileged to participate in it from 2005.

The major part of the work presented here was done only by the author on a researcher exchange funded by the Finnish Academy (under decision numbers 115898 and 121006). The exchange was a two-year period between 2006 and 2008 at Tohoku University, Sendai, Japan. The System Robotics Laboratory, led by Professor Kazuhiro Kosuge in Japan, provided the research facilities and the hardware to work with. Some algorithms from the Player [9] project were utilised.

Since mid-2008 the author has continued to refine the method in the Generic Intelligent Machines (GIM) research group. The GIM group provided valuable support for the software development and algorithms. Dr. Jari Saarinen provided the localisation algorithms, Antti Maula gave support to the interfacing of the hardware, and Sami Terho and Dr. Teppo Pirttioja shared their insights about the perception and world modelling.

1.6 Declaration of Previous Work

Some early work with the robot platforms used here is presented in conference papers [10, 11]. The basis of the micro task representation utilised in this work to describe the robot-dependent part of the task was presented in a doctoral dissertation [12], which was partly derived from the intermediate language for mobile robots (ILMR) presented in another doctoral dissertation [13]. Some principles of this work were published in [14].

Chapter 2

Introduction to Task Execution and Taxonomy with Service Robots

In this chapter more light is shed on the complex problem field tackled by this work. This is done by defining key terms used in this thesis along, with the explanation of some basic concepts and relations. There are many conflicting and overlapping terms used in robotics because the inspiration for solutions has been coming from diverse fields in science. Table C illustrates this fact. This chapter tries to rationalise the terms and provide crisp and unambiguous meanings for them.

2.1 Service Robots

A service is defined by Webster's New American Dictionary (1968) as: "1. The Performance or work done for another for hire. 2. Assistance." The Dictionary of Economics [15] defines a service as an intangible good often consumed at the same time as it is provided. A notable feature of a service is that it has two parts: a provider and a receiver. The receiver or consumer of the service in this work is referred to as a user of the service robot. The service provider is then a service robot. This is notably different from the case where a service is provided by a group of robots or by the combined efforts of a human and a machine.

A machine that has: 1) functions controlled by 2) sensory feedback and 3) actuators to affect the state of its environment is referred to as a *robot* in this work. Naturally, a service robot would be such a machine executing functions that would serve the user as [16] defines.

Service robots can be classified into single-task and multipurpose service robots. Single-task robots are designed and often optimised to provide only one kind of service. Multipurpose robots can provide many different services and for this to be possible some compromises have to be made in the design phase. A common assumption is that for a successful multipurpose service robot at least three general functions are vital: 1) perception and human-machine interaction (HMI); 2) mobility, and 3) manipulation [17]. For manipulation, this typically means a robot arm with the ability to manoeuvre the end effector to a desired position and orientation

in a sufficient workspace. For perception typical requirements are the ability to recognise and localise objects, obstacles, and humans, as well as the robot's own pose, in relation to some world model co-ordinate system.

If a tool is defined as an apparatus that can make people work more efficiently, a service robot could then be understood as a sophisticated tool for the user to do different kinds of tasks. The aim of this work is to partly increase the autonomy of this tool. Autonomy can be measured as the amount of supervisor intervention needed in the work process. A fully autonomous robot could assess the situation and provide the right services when needed without being asked to do so. It also makes sure that it is operational when needed. As mentioned in the introduction, that would be very difficult. To be useful, it is enough when the level of autonomy is high enough for the robot to provide a greater service than it requires work to operate.

There are other exceptions to justify the use of the service robots, even with very low autonomy level. First is the famous triple-D of the automation. With a robot you could do something that would be Dirty, Dangerous or Dull(DDD) otherwise [18]. Another case is tele-presence, where one can circumvent the classical problem of being in two places at the same time. In tele-presence, the robot roams around, observes and sometimes even interacts with the environment under manual control from a remote location. In an optimal case all sensory stimulus is transferred from the robot to the remote location so that the operator feels like being in the same location than the robot.

There are other exceptions to justify the use of service robots, even with a very low level of autonomy. The first is the famous triple-D of automation. With a robot you could do something that would otherwise be Dirty, Dangerous, or Dull (DDD) [18]. Another case is tele-presence, where one can circumvent the classical problem of being in two places at the same time. In tele-presence, the robot roams around, observes, and sometimes even interacts with the environment under manual control from a remote location. In an optimal case all sensory stimuli are transferred from the robot to the remote location, so that the operator feels as if they are in the same location as the robot.

2.2 Control of Service Robots

From the viewpoint of classical control theory, any feedback control problem, such as the control of a service robot, is constituted from a model similar to Figure 2.1. This diagram shows the flow of information or influence as arrows and the processing of information or influence as blocks.

The input to the system can be either a single value or, in a more complex case, a vector. For example, in the case of an oven as a plant we can have some temperature as a desired state, which is filtered into a certain voltage level to be a reference. The current temperature is measured and in the data acquisition process it gets filtered into a voltage level. The voltage level represents the state of the plant to the controller. The controller decides the correct control input for the actuator.

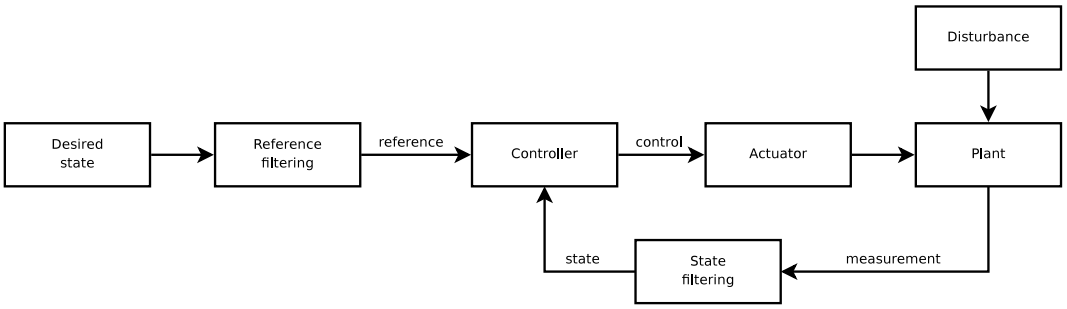


Figure 2.1: *Component block diagram of elementary feedback control*

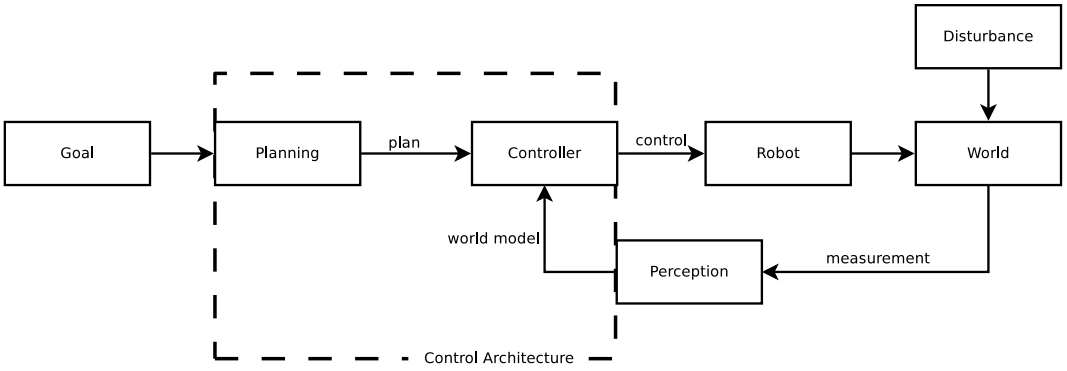


Figure 2.2: *Component block diagram of elementary robot control*

In this case the controller could be just a comparator controlling the current going to the heating element acting as the actuator. Finally, there would be a disturbance, such as a worried cook opening and closing the oven door.

In the case of a service robot the system is far more complex but can still be described with these elements, as can any system with feedback control. In Figure 2.2 the same diagram is shown with terms usually used in robotics. In robotics there is also a common term called "control architecture", indicated by a dashed line in the figure. Sometimes it includes the planning part and in some cases it excludes it. It is important to notice that some arrows were also included in the control architecture. The controller has two inputs, the plan (or task) to be done and the state of the world, in one way or the other. In a simple case these inputs are just single numbers but in the service robot case these inputs need to convey a huge amount of information. This leads to the fact that the information needs to be compressed for efficient processing in order to keep up with the speed of the world around the robot. Now we can conclude that to define a control architecture we need to define: 1) how the plan is represented; 2) how the world is represented, and 3) how the right control for the actuators is deduced.

An interesting common feature in the feedback control of complex systems is that several cascaded controllers exist in which the output from one controller is the reference for the next one. The case of service robot control is not different. So the control architecture for the service robot is actually a stack of cascaded con-

trollers or even cascaded control architectures. A famous paper [19] describes three cascaded controllers or architectures. The hierarchical stacking of the layers is challenged with the theory of heterarchical organisation, sometimes called agent-based control. But in their applications you can still find layers below and above the agents, and thus agents forming a layer in the hierarchy themselves. That is to say, in the feedback control of service robot there is not just a controller in series but a combination of controllers in series and parallel.

The feedback loop presented in Figures 2.1 and 2.2 is needed to keep the system stable. From classical control theory we know that too long a delay in the loop tends to make the system unstable. The delay is constituted of the time spent gathering the measurements and solving the suitable control, along with the effectiveness of the actuator to change the state. What is too long a delay then depends on the speed or time constant of the plant or world around the robot. And when service robot control is considered, several control loops with different delays are actually present. They each still use the same sensors to observe the world, but luckily, typically, as the level of abstraction gets higher, the system being controlled gets slower.

The control of a service robot is a complex problem and, like any complex problem, an essential part of the solution is its suitable division into subproblems. This is commonly done by dividing the control into hierarchical layers. The breakdown of the problem continues with dividing each layer into components. With this kind of division it is always essential to define well how and what the layers and components communicate with each other. Figure 2.3 attempts to illustrate the division. The NASREM architecture [20] proposed six layers with different levels of abstraction. [19] suggested a division into three layers based on the world model the components use. The first one utilises just the instantaneous sensor readings, the second one has some memory of the world model, and the third one also predicts the world model. Volpe argued in [21] that only two layers should be used for the sake of hiding information from higher-level layers.

What is communicated down in the hierarchy of layers or control architectures is the plan or task representation. Thus, in the case of several layers, several representations are needed at different levels of the control. The combination of components used in one or more layers is also a well-known and widely studied problem, and so several solutions have arisen and they are commonly called robotic middleware [9, 22, 23, 24, 25].

2.3 Task

The Oxford Advanced Learner's Dictionary (2001) defines the word 'task' as "a piece of work that somebody has to do, especially a hard or unpleasant one". The general description of a task is further clarified here as a process that has some purpose or goal in the operating environment. Robots operate in a real *physical environment* and their tasks are, then, real tangible tasks and not just abstract operations inside a computer.

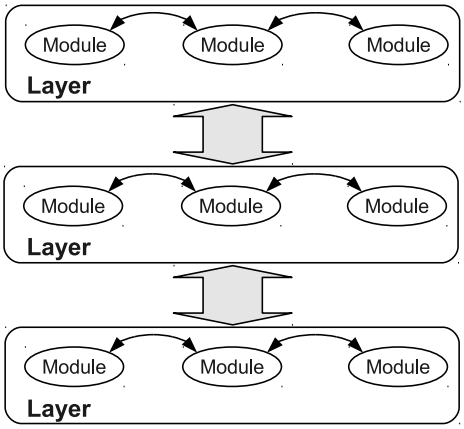


Figure 2.3: *Layers and components of a robot control architecture*

When the task for a service robot is being configured, the environment is typically clear from the context and does not need specific definition. The purpose and *goal* of the task are relatively simple to describe to a human but for a machine they are then much more difficult to define. That is actually a branch in a science called ontology. When the task is initiated and its execution is started, the intended execution process, the *plan*, needs to be described. There are also relevant *metadata* related to the task. It is important to define when and how the task is started and sometimes when it is considered to be done. Most commonly this is done by defining the pre- and post-conditions of the task. Other metadata could be the importance, estimated execution time, success rate of the task, or who configured or initiated the task. Sometimes it would be interesting to know if the task is a part of some greater entity and, if it is, which part of it. The metadata about the task can be used to arbitrate the right task to do on the basis of some assigned goals and the current situation. This is often called knowledge in the control of robots.

Many different types of tasks can be identified on the basis of their execution process and goal, as shown in Table 2.1.

Table 2.1: Task Types

Type of task	**Example**
Continuous	Follow a person
Do once	Unlock the door, clear the yard
Maintain state	Keep the door open, keep the yard clear
Manipulation	Open the door, Pick up litter X from the yard
Information delivery	Tell person X that the door is open
Observation	Check if the door is open

These tasks have differences in terms of the level of abstraction, dynamics, and the feedback needed. But a common feature is that a task has some purpose or goal in itself. The task does not define exactly how it should be achieved. It may have a plan of how to do it but the situation can change and the plan becomes invalid. Tasks can be defined on different scales. Picking up a piece of litter from the ground or removing litter from the whole city are both tasks but on different scales or levels of abstraction.

A term related to the task and used many times in robotics is *mission*. The term 'mission' is not used extensively in this work, but typically it describes a very abstract task with possible multiple goals. The term 'goal' in the literature is sometimes analogous to a task in the sense that it can define the purpose of the task. In this work we adopt a definition that the goal is just part of the task configuration. The goal can change during the task, depending on the environment and parameters of the task, while the purpose of the task can stay the same. The definition of a goal typically leaves the way to achieve the goal completely open. Different tasks can have the same goal and different plans may achieve the same goal.

In many cases the most challenging part of the task execution is confirming when it is done, i.e. when the goal is achieved. For example, a simple 'goto' command can have only a certain level of accuracy because of the uncertain pose and limitations of the hardware and underlying position controller. As another example, when an audible message is delivered, to make sure it was received and understood, should the robot try to evaluate the facial expressions and direction of the gaze of the person? Or should there be a defined way to acknowledge the message, which in turn would not be intuitive and effortless robot usage?

Technically, many tasks can be done without explicit information about the final end result. A task could be executed in an open-loop control and in some cases coercion can be utilised, for example, delivering an audible message multiple times, loudly and close enough to the person receiving the message. However, it is vital for many functions, such as unsupervised learning, to observe the end result of the task. If there is no way for the robot to evaluate the performance of the task, it cannot learn from non-existent past experience.

The right performance level of the task is an essential feature to consider when configuring and executing a task. The optimal performance level is a very vague expression, despite the intuitive clear sound of it. Task performance can be pruned into measurable numbers relatively easily on the basis of time, reliability, or energy used, among other metrics. These measures often contradict each other and even their relevance can change according to the time and situation. So it is important to find designs and execution parameters that are "good enough" under a wide variety of execution conditions, especially when the conditions are typically unobservable to the robot.

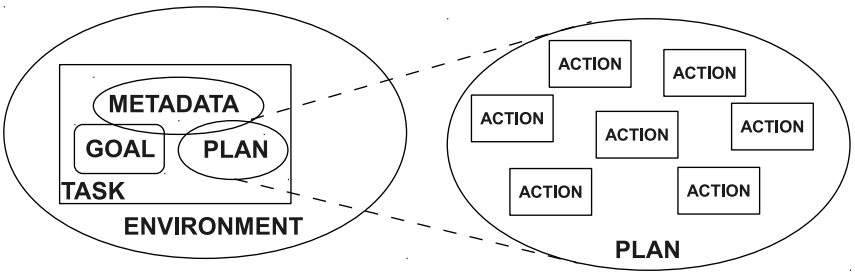


Figure 2.4: *Relations of the terms used*

Nested Tasks

As mentioned, tasks can be on different levels of abstraction. A more abstract task can be constituted from less abstract tasks and this relation makes them *sub-tasks* of a task with a wider scope. The abstraction can work in two directions. In the top-down approach a task can be divided into sub-tasks and the most commonly used formal way of doing this is the hierarchical task network (HTN) [26, 27] method. Alternatively, a less commonly used and, arguably, a more natural bottom-up way is finding common combinations of sub-tasks and merging them into one task.

When the task cannot be divided into smaller sub-tasks, an *atomic operation* or *action* can be perfomed. An atomic operation can be defined as an operation that can be controlled only by starting and stopping it. An atomic operation is also subject to the viewpoint. If we break the task down into the smallest bits and follow them down with our viewpoint, we can consider how a single electron is causing a change in an electrical field and creating moment in an electric motor. The solution for the viewpoint of selection is the definition of the layers of abstraction. That fixes the viewpoint on some specific level. The division into sub-tasks is actually the *plan* of the task. Figure 2.4 illustrates the relations between the terms.

2.4 Plan

The *plan* is a frequently used term in robotics. It defines how something is intended to be done. The plan can be defined using actions, tasks, goals, or any entity describing some sort of activity. The plan also has some rules to define in which order and combinations the activities are executed. A plan for picking up a piece of litter from the ground can be a sequence to stretch the arm, grasp the litter, and lift the arm up with the litter. On the other hand, a plan to remove litter from the whole city can contain the task of picking up a piece of litter from ground. The plan does not have a purpose in itself like the task has. The result of a plan is dependent on the current situation when the plan is executed.

2.4.1 Planning

The right plan for the situation can be achieved in two ways. 1) It can be created on the basis of the situation with a process of *planning*. 2) You can have a set of plans that are already planned and choose the best one for the situation. Typically, planning is a very expensive, slow, and difficult process. Thus these two methods are a kind of trade-off between speed and applicability.

There is a hybrid method that humans are believed to follow. A coarse and abstract plan is selected from a set of ready-made plans that best match the situation and then the details are planned and defined as the execution of the plan advances. This requires very high-level symbolic abstraction of the environment and plan in order to find the best-matching coarse plan.

The inputs for the planning process are the current situation, the goal, and the available operations. The planning process then outputs a sequence of operations to achieve the goal from the current situation. In automated planning it is required to express which goals a particular operation can achieve or the planner has to have the ability to predict it. Then the search for a suitable sequence can start from the current situation and/or from the goal, first by finding operations suitable for the current situation or achieving the desired goal, then by estimating the following situation after or before the selected operations. These steps are repeated until the sequence between the goal and the current situation is found. Of course it might be that such a sequence cannot be found.

Now, different ways of representing the operations and their relations can make this planning problem easier and faster. The most inconvenient fact is that the output of one operation or the activities of other agents in the same environment cannot be guaranteed. There are many techniques to accommodate this. 1) The planning can be done more frequently than the environment is changing. 2) The validity of the plan is constantly monitored and planning can be done again when the plan becomes obsolete. 3) The alternative operating sequences are considered and accommodated into the plan. 4) If the environment cannot be observed in detail, blindly believe that nothing will go wrong. If the plan is made beforehand, only options 3 and 4 can be used.

One technique is to divide the goal into sub-goals that are perhaps easier to achieve. Again, this requires advanced understanding of the problem. When working on a layered architecture, the atomic operation on one layer has to have a plan for execution on the next one. So the planning has to be done on all levels of abstraction, either in an automated manner or manually, and either in situ or beforehand. Furthermore, the plan has to be ready on the lowest level before anything can be done. In principle, in the case of a robot the very lowest-level plans are fixed beforehand by soldering them into the circuit boards and bolting them into degrees of freedom.

The plan defines the process of execution of the task, but the planning process does not define all the other information in the task. The process of defining all the required information in the task is called *configuring* the task.

2.5 Multi-tasking

As already mentioned, there are multipurpose robots that can execute different kinds of tasks. If the tasks can be under execution at the same time instant, it is called *multi-tasking*. Actually, for multi-tasking it is not necessary to do different tasks involving different objects; the task can be same but the object must be different or the object can be same but just the tasks different. The last case is the most difficult, because the tasks are likely to interfere with each other. The division into different types of multi-tasking used in this work is defined in the following subsections.

The capability for multi-tasking is a very important feature. A proof of that is its wide use in the world of nature. For example, a bird can eat and watch out for predators at the same time. It can use its head to peck grains from the ground and to glance around, or it can fly and look for a good place to land at the same time.

2.5.1 Time-sharing

Doing any kind of task requires the use of some kind of resource, be it computer-processing power, energy, space, or time. When different tasks need to use the same resource and it cannot be shared, they have to use it in turns. This is so-called *time-sharing* multi-tasking. There are several ways to coordinate access to resources. A major division is into co-operative time-sharing and pre-emptive time-sharing

In co-operative time-sharing the task voluntarily releases the resource after using it for a while. This is prone to problems with, intentionally or not, selfish tasks that can occupy the resource excessively. Pre-emptive time-sharing limits the time that each task can use the resource and ensures a share of time for each task. Thus the task can be suspended in the middle of its execution in order to share the resource, but tasks are more independent of each other.

2.5.2 Concurrent

If different tasks use different resources, there are no conflicts for the resource. Thus the tasks can effectively be executed at the same time and this can be called *concurrent* multi-tasking. A classical example is a human chewing gum and walking at the same time.

2.5.3 Parallel

If several instances of the same resource exist, the tasks can be divided into elements to be executed on any of these resources. This is called *parallel* multitasking. An example could be a human picking up litter with both hands. The left hand can be used to answer a mobile phone while the right hand continues picking and the left hand can join the litter-picking task after the call finishes.

2.6 Task Knowledge Representation

Task knowledge is considered in this work as all required information related to the task in order to execute the task. What is required depends on the infrastructure in which the task is executed. The task knowledge or domain knowledge of Matarič [28] is very important for the efficient utilisation of service robots. Thus, the way in which the task knowledge is represented becomes a crucial problem. The process of configuring and planning what the robot should do is a challenge in itself. Therefore, the syntax of the representation should support the design process. In practice this means that the elementary atomic operations of the robot should be abstracted to functions on the same level as human thinking if a human is doing the planning or wants to understand the plan. So the humans can concentrate on the real problem of how the task should be performed.

When configuring a task, it is very difficult to get everything right from the beginning. It is common to add new features to existing plans and it is just good practice to start from simple things and make sure that they work before doing something more complex. Thus the task configuration is an iterative process. The initial plan is refined and new features are incrementally added at each iteration. And in the case of a more complex system, with multi-tasking operation, individual tasks should be able to be developed independently.

One viewpoint on the task is the data it contains. The task can be divided into a static part and a variable part. For example, in a common 'goto' task the variable parts are e.g. the goal pose, the margins within which the goal pose should be achieved, and the maximum speed or acceleration for the motion or deadline when the goal has to be met. On a general level, this variable part is common for all representations within the myriad different ways to represent a task.

The static part of the representation depends on the underlying control architecture of the robot. The task is like a computer program and the representation is its source code. Each control architecture has its own instruction set for elementary building blocks of the source code. Typically, you can combine these instructions to form a higher-level abstract function or instruction. All in all, the static part is different from robot to robot.

Service robots can be very different from each other, but they need to do the same kinds of things. For example, the mobility of the robot can be based on wheels, tracks, legs, or wings, but still it performs the same function, manipulating the robot's pose in the world coordinates. So on some level the task knowledge could be transferred between different robots and enhance the usability of the robot. For this the different levels of abstraction in the representation come in handy. Learning in robotics has recently gained a lot of attention as a research topic. Task knowledge representation is a vital part of the task learning process, because that is exactly what the learning process works on. Typically, it tries to optimise the representation on the basis of the minimisation of some cost function or maximising the reward function.

The task knowledge is separated from the control logic. It defines the archi-

tecture's control logic of how to execute a task. Distinctive requirements for the representation of the task knowledge in this work are:

1. The task representation can be serialised to be saved or sent over a communication link, i.e. it is not an emergent behaviour of the system.
2. The task can be configured independently from the robot and the representation can be used to transfer it into the control architecture for the execution, i.e. the plan is separate from the execution process.
3. The representation of one task can be transferred to another instance of the same control architecture for execution, i.e. it is generic and robot-independent.

2.7 Non-Functional Requirements of Service Robot Control Architectures

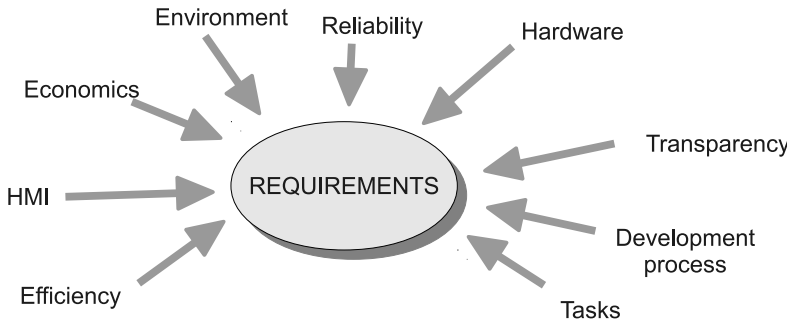


Figure 2.5: *Constraints for service robot task execution causing the requirements*

There are many factors that affect the use of a service robot and thus its control architecture. There are requirements imposed by the hardware, environment, economics, users, developers, tasks, etc (Figure 2.5). In this section the non-functional requirements identified by the author of this work and other authors are presented. At the end all the quality characteristics are combined and analysed. The correspondence between different authors is presented in a compilation table (Table 2.2).

The actual goals or non-functional feature requirements for service robots receive surprisingly scant consideration and are rarely well-motivated in the literature. It is challenging to define a legitimate base to consider these requirements and list them unambiguously and it requires extensive research in itself [29]. That is left for future work and instead a consensus of other authors' opinions is gathered up. This is not intended to be a comprehensive and exhaustive list of requirements. The purpose of this listing is to create some basis for the analysis of this work. So this list is referred back to in Chapter 8.

First, a list of the most basic general requirements for the control architecture, in the author's opinion:

- the possibility of controlling the execution by suspending, resuming, and cancelling the task execution
- responsive to changes in the environment and in the robot
- stable with a variable amount of uncertainties in the environment

When the efficiency and economics of the service robot are considered, the author likes to raise the following requirements above others:

- multiple tasks under execution at the same time
- different resources of the robot in use at the same time

For easy human-machine interaction (HMI) with the system the author promotes the following features:

- add and remove tasks at will
- set priorities for the tasks
- monitor tasks' execution state

2.7.1 Non-Functional Requirements in Literature

Ingrand [30] and Alami [31] define the following requirements for an autonomous planetary rover system:

- programmability
- autonomy and adaptability
- reactivity
- consistent behaviour
- robustness
- extensibility

Pattie Maes defines in [32] the following characteristics for action selection behaviour for an autonomous agent, such as a planetary rover:

- goal- or multi-goal-driven actions
- the selected action is relevant for the current world state
- stick to a particular goal
- plans to handle interacting and conflicting goals
- robust even if parts break down
- reactive and fast

In [33] Laffey et al. expect the following classes of features from a real-time expert system:

- efficient integration of numeric-symbolic computing

- continuous operation
- focus-of-attention mechanism
- interrupt-handling facility
- optimal environment utilization
- predictability
- temporal reasoning facility
- truth maintenance facility

In [34] Rodney Brooks identified a number of requirements of a control system for an intelligent autonomous mobile robot:

- multiple goals
- multiple sensors
- robustness
- additivity

In [35] the following agent design problems are indicated by Atkin et al.:

- processing sensor information
- reacting to a changing environment
- integrating reactive and cognitive processes to achieve an abstract goal
- interleaving planning and execution
- distributed control
- allow code reuse within and across domains
- using computational resources efficiently

Evan Drumwright and Victor Ng-Thow-Hing define the following design goals for a task matrix [36] framework to describe tasks for a humanoid robot:

- simple task programs can be treated as primitive components to perform more complex tasks and behaviours.
- the task matrix is independent of any particular approach to goal planning or task sequence execution.
- on-line additions to the matrix are allowed to facilitate continual learning or the upgrading of skills
- the task matrix should promote robot independence.

In paper [37] Christopher Parlitz et al. define the following indispensable key issues for a robot framework:

- modularity
- platform independence
- inter-process communication
- resource management
- configuration of the systems
- robustness
- tools for testing and debugging
- performance analysis

2.7.2 Combined Feature Requirements

In the Table 2.2 the features identified by other authors are combined. Some of the requirements grouped together are not exactly identical but close enough to be considered same in principle. The table also indicates classification of the requirements into runtime qualities and development time qualities. In the following, the list of distinguishable requirements are defined and motivated in detail. Some general requirements such as autonomy, integrity or maintainability are considered as unions of the requirements listed here:

In Table 2.2 the features identified by other authors are combined. Some of the requirements grouped together are not exactly identical but are close enough to be considered the same in principle. The table also indicates the classification of the requirements into runtime qualities and development time qualities. Below, a list of distinguishable requirements have been defined and elaborated on in detail. Some general requirements, such as autonomy, integrity, or maintainability, are considered to be unions of the requirements listed here.

1. **Programmability** Programmability The robot should be able to achieve multiple different tasks described at an abstract level. The way to configure the tasks has to be simple enough for a novice user without background experience to utilise it. The way to programme the task has to be expressive enough to enable complex tasks to be performed.

2. **Adaptability** Adaptability The robot should select relevant actions for the current state of the world. It should be possible to refine or modify the task and its behaviour according to the current goal and execution context. The changes and non-uniformity of the environment should not only be recognised but exploited.

3. **Reactivity** Reactivity Reactivity to an outside stimulus has been considered as a vital function for a robot for decades [34].

4. **Predictability** Predictability The reactions of the robot to the events must be guided by the objectives of its tasks. If the robot decides to pursue some task, it should stick to it and not spend all its resources switching between tasks. Some tasks are more important or urgent than others. If one task is prioritised, judgment can be subjective to the operator; its execution should be more likely than that of the others.

5. **Robustness** Robustness All the information that the robot gets is plagued with uncertainty. No sensors are perfectly accurate and the world tends to change faster than the sensor readings are refined into observations about the world. This variation in the operational environment has to be incorporated for reliable functionality [38]. Sometimes parts of the robot may even break down. It would be good to have redundancy in the robot's functions or at least the execution should continue with the parts that are still working.

6. **Extensibility** The integration of new functions and configuration of new tasks should be easy. This includes the configuration of the task just by re-organising and adding parts from other tasks. Learning capabilities are important to consider here: the architecture should make learning possible.

7. **Multi-tasking** Making and operating a robot capable of manipulating the state of the world, which is shared with humans, requires a considerable amount of material, energy, and space. It is economically justified only in very few cases to have only one robot for one task. The robot should be able to do many different kinds of tasks. It would be even more beneficial if the robot could perform multiple tasks at the same time. Many tasks include waiting periods and parts of the tasks are done in locations far apart from each other. With rational task scheduling for multi-tasking execution, the operation can be made more efficient.

8. **Resource management** Different parts, sensors, and actuators, of the robot are relatively costly. When executing one task at a time, often only one part of the robot is used. Again, when multi-tasking, with careful scheduling, better utilisation of all robot parts could be achieved.

9. **Attention control** Perception functions are typically computationally the most expensive operations of a service robot. The robot has to direct its attention, i.e. sensors and perception functions, on the basis of the tasks at hand. Only the relevant parts of the environment should be observed.

10. **World model** The robot has to have an abstract representation of some parts of its operational environment. This is required to interact and communicate with the robot, to accommodate complex tasks, and even to enhance the sensory readings.

11. **Sensory information** Even relatively simple sensors can generate considerable amounts of data. The data are rarely usable by themselves. They have to be filtered from noise and pertinent information has to be extracted.

12. **Interleaving planning and execution** Planning competes with perception for the title of the computationally most expensive operation. There is typically more room for adaptation in planning than in perception. The planning horizon, level of detail, and level of branching and pruning of bad-looking options can be adjusted. The planning and execution processes have to run in parallel and the result of the planning should be joined smoothly to the execution. Safe robot operation should not be compromised by the computational resources required for planning.

13. **Modularity** Many feature requirements mentioned above demand parallel processing. The control of the service robot requires the division of the system into more manageably sized units. The computing system has to be distributed over many processors to guarantee responsiveness and accommodate

the computational load. All this is best done by dividing the software into modules. It is also beneficial for the hardware to be modular [39].

14. **Platform independence** If the architecture can be independent of the platform, the advances made and tasks created in one robot could be transferred to another robot. The development of the system can be divided into many independent parallel units.

15. **Independent representation** Because of the complexity of our world, in our daily life many situations appear in a seemingly stochastic pattern. Some tasks cannot be scheduled or anticipated but have to be set for execution or cancellation ad hoc. For this the tasks have to be independent of each other. It is an unsustainable situation if, while one task is being developed, other tasks have to be explicitly considered.

16. **Execution control** Execution control This is an intuitive feature. For any system physically operating in a shared environment, there has to be a possibility of stopping the system if something goes wrong. This feature requirement can also be found in many regulations regarding autonomous work machines and the like [40]. Furthermore, suspending the execution instead of just cancelling it is more of a usability feature.

17. **Performance analysis** For developing new tasks, to monitor the performance, and to diagnose problem situations, it is vital to know what the robot is trying to do and has already done.

Table 2.2: Combined feature requirements from different authors (R: runtime qualities D: development time qualities)

#	type	**Alami, Ingrad** [30, 31]	**Maes** [32]	**Laffey** [33]	**Brooks** [34]	**Atkin** [35]	**Evan** [36]	**Parlitz** [37]	**Author**
1	D	Programmability		Efficient integration of numeric-symbolic computing			Task construction	Configuration of the system	
2	R	Autonomy and adaptability	Selected action is relevant for current world state	Optimal environment utilisation		Integrating reactive and cognitive processes to achieve an abstract goal			
3	R	Reactivity	Reactive and fast	Interrupt-handling facility		Reacting to changing environment			Responsive to changes
4	R	Consistent behaviour	Stick to a particular goal	Predictability					Set priorities for the tasks
5	R	Robustness	Robust even if parts break down	Continuous operation	Robustness			Robustness	Stable with variable amount of uncertainties
6	D	Extensibility			Additivity		Skill upgrading		
7	R		Goal- or multi-goal-driven actions		Multiple goals				Multiple tasks under execution at the same time
8	R		Plans to handle interacting and conflicting goals	Temporal reasoning facility	(Multiple sensors, multiple goals)			Resource management	Different resources of the robot in use at the same time
9	R			Focus-of-attention mechanism		Using computational resources efficiently			
10	R			Truth maintenance facility					
11	R				Multiple sensors	Processing sensor information			
12	R					Interleaving planning and execution			
13	D					Distributed control		Modularity, IPC	
14	D					Allow code reuse within and across domains	Robot-independent	Platform independence	
15	R/D						Independent representation		Add and remove tasks at will
16	R							Tools for testing and debugging	Possibility of controlling the execution
17	R/D							Performance analysis	Monitor execution state of tasks

Chapter 3

State of the Art of Task Execution Principles

In this chapter the different ways to fuse the perception of the current state of the world and the desired task into motion commands to the robot's actuators are shown. These are commonly called control architectures or control paradigms; sometimes the term "control schema" is used. From what is shown in Figure 3.1 (reprint of Figure 2.2) this chapter mostly handles the "controller" part and a little bit of the "world model". As mentioned in 2, the successful control of a service robot requires a stack of cascaded controllers with different characteristics. The different characteristics are discussed here.

Control architectures have been studied for decades now and similar classifications to the ones used here could be found in many robotics textbooks [41, 38, 42, 28]. The division here is performed from the point of view of how the plan is handled. Robotics, as a science, approaches the problem in a practical experimental way. The closely related questions about the nature of knowledge and automated decision making are studied from a more theoretical aspect in the science of Artificial Intelligence (AI). AI also has a long history behind it and many textbooks cover the basic findings [26, 43, 44].

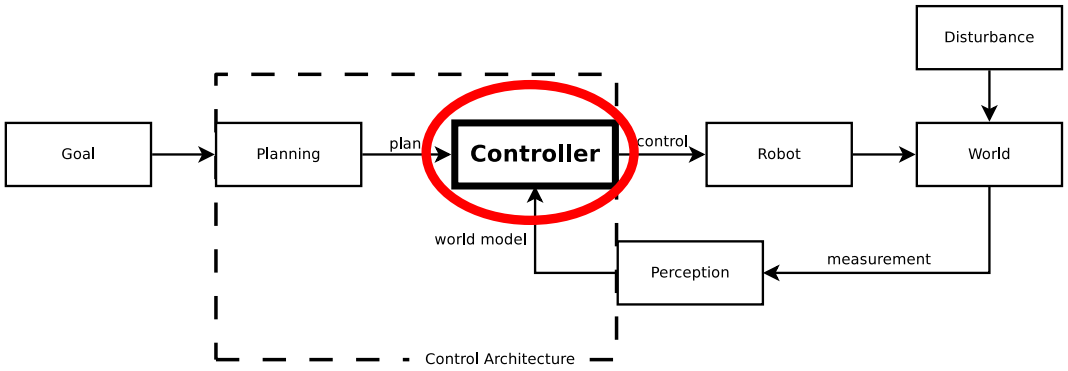


Figure 3.1: *Task execution in elementary robot control*

As it is a scientific field, many surveys related to the topic have been done. One form of these surveys is the state of the art in doctoral dissertations [45, 46], just like the one presented here. Some surveys can be found in conference and journal publications [47, 48, 49, 50, 51]. The same structures for the principles of controlling robots can be found in all of them. So it is not surprising to find standardisation efforts in the field, too [52].

3.1 Tele-operation

Tele-operation is a functionality where the robot is controlled from a remote location, relying only on telemetric information and not on direct human perception of the situation. One theory for developing an autonomous functionality is to start from full tele-operation and to gradually automate parts of the tasks and functions until full autonomy is achieved [53]. One example is Sandvik Tamrock's AutoMine [54], where front loaders operate mostly autonomously and only a fraction of the work cycle is still operated by humans. This is a good approach from the point of view of abstracting the task. Easy, obvious, and repeated procedures can be pruned out with automated execution and only the relevant information is brought to the human user.

The automation and abstraction process leads to very application-specific solutions that can be difficult to transfer into different tasks, or even the physical construction of the automated system can be too application-specific to do anything else. The key problem is that the whole system would be relying on phenomenal human abilities in perception that will not be able to be reproduced with machines in the foreseeable future. It is not enough just to automate all the basic functions in the work cycle. Some form of responsive automated interrupt handling is required too.

3.2 Deliberative

The deliberative Sense-Plan-Act (SPA) model has a long history. It is the first control scheme derived from the field of artificial intelligence (AI). It could also be seen as an extension of traditional feedback control theory. In SPA the control process has three distinct phases (Figure 3.2) in one control cycle [20, 55].

1. The current state of the world is modelled in detail or the model is updated.

2. The next action or a sequence of motions of the robot is carefully searched for from among a myriad options. The search is based on the world model and predictions made from it.

3. The action is executed.

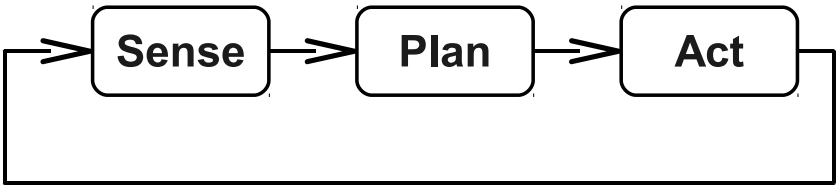


Figure 3.2: *Deliberative control scheme*

Unfortunately, during the prime time of this scheme (the 1970s and 1980s), the computing power was very limited and one control cycle could take minutes to execute. This was a growing problem for practical implementations for the real world, even in static environments. If the reaction times for humans are compared, they are in the order of one second or fractions of a second. For the robot to survive in our dynamic environment, the reaction time should be along the same scale. The lack of computing resources could also be a reason for this scheme. Only a pre-calculated sequence of movements could be controlled by the computers of those days and that is what the control scheme had to provide.

The SPA model works as long as the world behaves as predicted. When, and, it is hoped, if an unexpected change in the world is detected, re-planning is needed. The original SPA scheme assumed that changes in the world between the control cycles would be so small that they could be neglected. Any change would be included in the next cycle. An expensive planning operation is required to be performed during every cycle. With today's computers this scheme could be more feasible. But the world model would also be more detailed with today's sensors and the search process over all possible scenarios would be more difficult. Maybe, after all, today's computing resources could be utilised better with another scheme.

3.3 Reactive

The reactive and, especially, the subsumption model was the answer to the slowness of the SPA scheme [34, 56, 32]. In the reactive model the control is based on separate modules that react to a stimulus from the environment by exerting a command based on the stimulus. It skips the whole planning process altogether (Figure 3.3). As a consequence the perception process is lighter, too, because the world model does not need the information for predictions.

Each module in a reactive system handles the sensor information independently. This means that the information from the sensors has to be broadcast inside the system somehow. This is usually easy, but the other end of the process is more challenging. All the commands produced by the modules have to be somehow combined together. There is only one robot to command, after all, and it cannot go

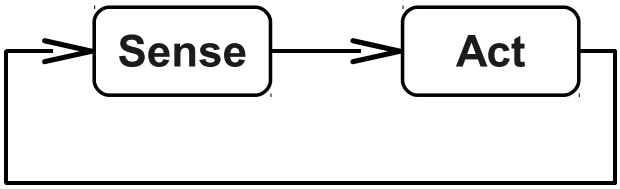


Figure 3.3: *Reactive control scheme*

left and right at the same time. Generally, commands are combined together, either by fusing them together, for example with a vector sum, or by arbitrating between commands, for example, on the basis of priority.

There are many advantages to having the control process performed by separate modules: 1) the development and testing of different modules can be done independently. The control problem can thus be divided into smaller subproblems and the system can be built incrementally bit by bit; 2) the system would be robust against the failures of single modules. The other modules than the failed ones would still be giving commands that were sensible in some manner; 3) the system can be implemented on a distributed computing system. Separate processing units can handle separate modules in parallel and the execution time can again be improved.

The command or output from one module does not have to be a command directly to the robot's actuators. It can be an input for some other module. Furthermore, some modules can inhibit or activate other modules. In this way a more complex or higher-level response from the system could be achieved. If a network of modules is constructed smartly, even the command combination problem can be omitted. Sometimes an emergent behaviour can arise from this kind of network without being explicitly programmed.

Still, a more complex task requires some plans. The task can have different stages in task execution that are very difficult to sequence just by constructing a smart network of reactions. Sometimes task execution requires interaction with abstract concepts that cannot be directly observed, have been observed before, or it is known will be observed in the future. But with simple tasks, the reactive paradigm works like a charm.

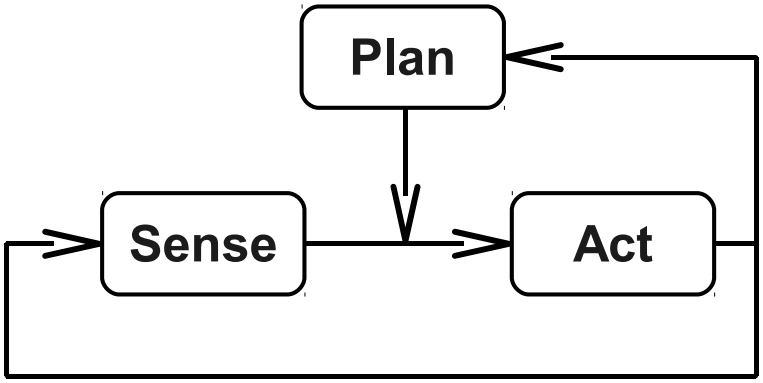


Figure 3.4: *Hybrid control scheme*

3.4 Hybrid

The hybrid architecture combines the two paradigms described above and utilises the good features of both [19, 57, 58, 59]. Working instantaneously or on a short time horizon, the reactive scheme provides fast responses for the robot. The deliberative scheme works on a longer time horizon and makes sure that the things that the robot does are beneficial in respect to the goals that the robot has. The two different time horizons can be seen as two feedback loops in Figure 3.4. By planning ahead it can avoid problematic situations beforehand and be proactive.

The combination of these two paradigms is not so straightforward. The output of the deliberative part is a plan, typically as a sequence of actions, and the input of the reactive part is measurements from the environment. In the hybrid scheme these parts are layered and a sequencing layer is placed between them. The job of the sequencing layer is to interpret the plan into reactive networks or network parameters. Furthermore, it has to change the parameters or the network for the reactive layer at the right time, according to the plan.

This forms three cascaded controllers. In a compressed form, the deliberative top layer takes in goals and outputs plans. The sequencing layer takes in the plans and selects the right reactive controller configuration for the time on the basis of the plan. The reactive layer just takes measurements and outputs the control for the actuators.

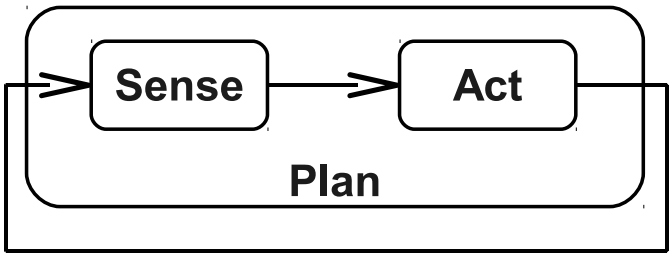


Figure 3.5: *Behaviour-based control scheme*

3.5 Behaviour-Based Control

The planning is really hard thing to do. As mentioned in the introduction, estimating the future events in our dynamic environment is a challenge. One way to divert the difficult planning part is the behavioral based control paradigm [60, 42, 61], where the planning is integrated into the interactions of the behaviors (Figure 3.5). Behaviors can be considered as model for animals and humans alike to control their activities. Depending on the internal state of the agent and stimulus from the environment some behavior can be activated or deactivated. Complex functionalities can be achieved with large enough number of behaviors. The study of this is called ethology in biology or behaviorism in psychology and in robotics it is just behavior based control.

The planning is a really hard thing to do. As mentioned in the introduction, estimating the future events in our dynamic environment is a challenge. One way to divert the difficult planning part is the behaviour-based control paradigm [42, 60, 61], where the planning is integrated into the interactions of the behaviours (Figure 3.5). Behaviours can be considered as models for animals and humans alike to control their activities. Depending on the internal state of the agent and the stimulus from the environment, some behaviour can be activated or deactivated. Complex functionalities can be achieved with a large enough number of behaviours. The study of this is called ethology in biology or behaviourism in psychology and in robotics it is just behaviour -based control.

Behaviour is conceptually close to reaction. They run in parallel and take inputs and produce outputs on the basis of their own subjective intentions. The outputs are fused, just like with reactions. Compared to reactions, instead of using just the direct stimulus from the environment a behaviour can utilise some abstract concepts, such as memory or the internal state of the robot. The behaviour itself can also have internal states and change its output accordingly, even if the input from the environment is constant.

For complex tasks temporal sequencing is required. One way to do this is to combine the behaviours into a network where the input can come from the environment or from the other behaviours and the output goes to actuators or other be-

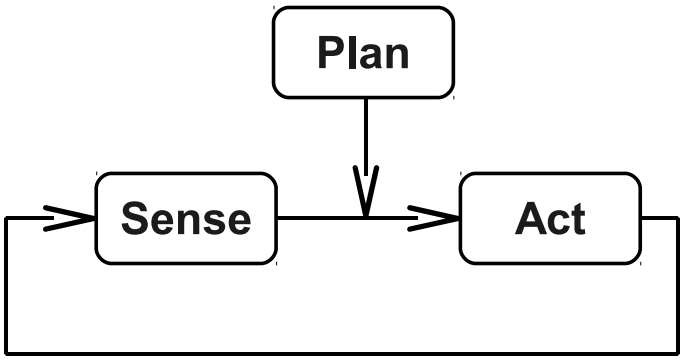


Figure 3.6: *Knowledge-based system*

haviours. The network of behaviours can produce this by behaviours with activation conditions from preceding behaviours. A task constructed from this kind of network can become too complex to handle and design by hand. [62] proposes a solution for this by introducing a "hierarchical abstract behaviour architecture". It divides the behaviours into primitive behaviours, whose output also goes to actuators, and abstract behaviours, whose output goes only to other behaviours. A network made up of abstract behaviours could be combined into one even more abstract behaviour. In this way a hierarchical composition of the behaviours could be created, hence the name.

Another way to utilise behaviours is to define their interaction and to sequence them with a finite state automaton (FSA). FSA and state diagrams are discussed more in the next chapter, but the basic difference from an interconnected network is the fact that only one discrete state is active at a time. The state can be an individual behaviour, updated weights and connections in a network, or a completely separate state-specific network.

Control based on behaviours is still a complex and somehow rigid system. If the objectives for the system change, it can be quite an endeavour to adjust the control to the new objectives. Generally, a behaviour-based system is better suited to a completely autonomous system than a system with external commands telling it what it should be doing.

3.6 Knowledge-Based Systems

Knowledge-based systems (KBS) also arise from the studies of artificial intelligence. They do away with the difficult planning by doing it offline. The plans or knowledge about what to do in different situations are gathered beforehand. Typically, the information is gathered from an existing expert on the domain and not explicitly planned at all. Thus a certain rule-based form of KBS is called *expert systems*. Figure 3.6 illustrates KBS a little misleadingly, while trying to stay loyal

to the common way of representing SPA, reactive and hybrid controls. It emphasises that the plans are not created on the basis of the current state of the environment. But, of course, the plan selection does take account of the current state of the environment.

KBSs have been applied in different domains for a couple of decades but their use in robotics and especially in service robot domains has mostly been limited to simulation cases. A good but slightly outdated survey can be found at [33]. The tools, methods, and representations of the expert systems have been developed to be very general and there is room for optimisation for the service robot domain. A couple of knowledge-based systems are introduced below.

3.6.1 Expert System

Generally, an expert system means a system where the knowledge of an expert is gathered. It is gathered into a representation that can be deduced by a machine. The representation should support the collection of the information from several experts. The expert system should solve a problem in a specialised field.

One way to represent the knowledge is to formulate it as logical rules. The rules can then be expressed with so-called fuzzy logic (instead of crisp logial values, the computing is done with variable levels of logical values) [63], which can incorporate some of the environmental uncertainty and contradictory advice from the experts. The rules are then in "if-then" form, such as: "If the goal is left then turn left" or "If something is too close in front then turn right". The outputs of the different rules can be summed and, for example, averaged to get the exact output value. In this way a continuous non-linear mapping from the current state to the desired action can be made.

3.6.2 Procedural Reasoning System

Procedural reasoning systems (PRS) come from the *beliefs-desires-intentions (BDI)* model in agent theory [30, 64, 65]. Beliefs are the information gathered from the environment. Desires are the goals for the system. Intentions are deduced on the basis of the beliefs and desires. The PRS then further specifies an interpreter and a knowledge library component. The knowledge library is a library of plans that can fulfil specified goals. Furthermore, the plans have preconditions for when they can be applied. Figure 3.7 illustrates the PRS components. The interpreter then fuses the information from all the other components and generates the intentions for the system. The outcome of the PRS is not a specific plan but a suggestion of what would be a good thing to do or, in one word, intentions. Again there has to be a cascaded structure, with another controller then generating actuator commands on the basis of these intentions.

PRS is a good model for formally dividing the task control problem into components and defining the interactions between components. The really hard problems are still unaddressed. How can the environment be discretised into beliefs that can

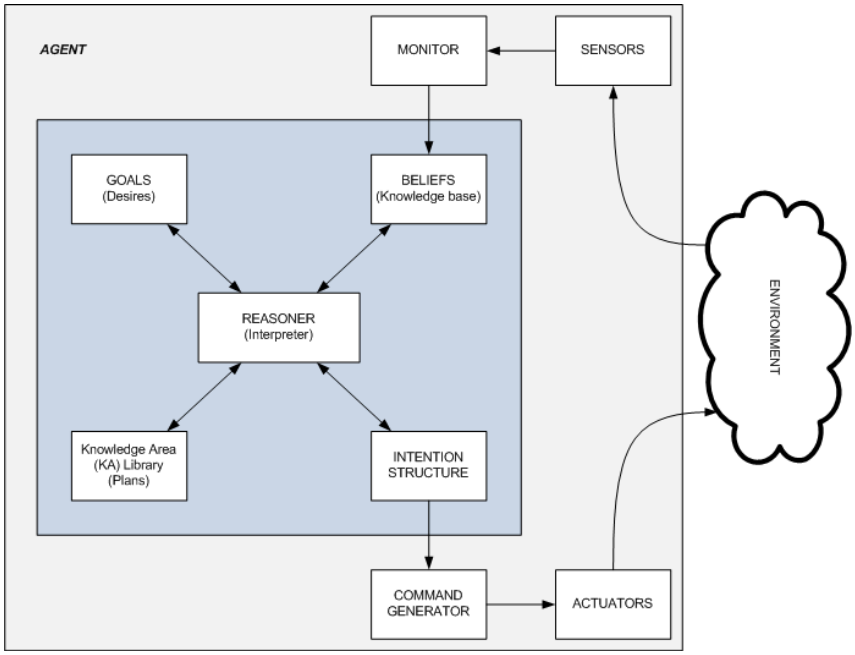


Figure 3.7: *Components of the procedural reasoning system (courtesy of Wikipedia)*

be matched to the preconditions? What should those beliefs be? How can the uncertainty about the consequences of the actions taken be expressed? How can that information be used in the decision making?

3.7 Discussion

All the systems described above could be combined in many ways to achieve a complete control system. Each behaviour in behaviour-based control could have a KBS inside. Or the KBS could define the behaviour-based network for each time instance and the behaviours can have deliberative control architectures inside them. The best selection for the control problem at hand is not based only on the architectural principles; the implementation also plays a crucial role. For the domain of this work, a robot in an dynamic open environment, the common understanding is that it requires reactive control for the lowest level.

Some characteristics of each control method are discussed below, specifically from the viewpoint of the plan and the world model. The discussion is based on the formalism defined in Section 2.2 and illustrated in Figure 3.1.

3.7.1 Plan

All the methods that are presented handle the plan differently and they are summarised here. If it is assumed that the planning for complex tasks is too difficult for today's technology, we can rule out the deliberative and hybrid solutions. The complexity of the problem can rule out the reactive method and, arguably, the behaviour-based method. This leaves the knowledge-based systems as a viable option for the problem-setting of this work.

- Tele-operation

  The plan is in the operator's head. Some parts of the plan can be given to the robotic system, such as way points to a mobile robot. The plan can be presented to the human operator even in vague free-form text or speech and the operator can produce accurate and specific operations on the basis of those.

- Deliberative

  The plan is created from scratch during each control cycle on the basis of the current situation.

- Reactive

  There is no plan, or the plan is embedded into the construction of the reactive components, i.e. it has to be pre-configured into the system in the design phase.

- Hybrid

  The plan is still created from scratch but the reactive layer makes sure that it is safe to operate the robot, and meanwhile the planning takes place. The plan, when ready, then alters the reactive layer on the basis of how the sequencing layer interprets it.

- Behavioural

  Behaviour-based control does not define the concept of the plan. The plan can be embedded into the interactions of the behaviours or the plan can be partly decomposed into smaller subtasks that the behaviours can accomplish. It is said that the behaviour-based system does not need to do planning, but individual behaviours can distribute the planning problem. Anyhow, the plan should also be pre-configured in this approach.

- Knowledge-based

  The plan is part of the knowledge. The other part of the knowledge is to know when to use which plan. The pre-configured plans are in a kind of lookup table and the one most suitable for the current situation is selected.

3.7.2 World model

All the world models that the control schema presented above expect are collected into this section. It is important to notice the differences and the requirements for the computing resources imposed by them.

- Tele-operation

  For tele-operation, the system has to provide enough information from the remote location for the user, so that the user can build the necessary world model inside his head. The representation can be just visualised direct measurement data. Some further filtering and abstraction to symbolic representation can be done to offload the requirements for human attention. For example, for operating a car, it is not necessary to know what the exact oil level is but if it is too low, some indication should be shown.

- Deliberative

  In deliberative control the whole world the robot knows is modelled into a data structure. The most important feature of the model is that it can be simulated to obtain the outcome of different options. It is not just a passive storage of information but also a simulation engine. The simulation should model all the actors and dynamics of the whole environment.

- Reactive

  In the reactive paradigm the world is its own model. The belief is that nothing needs to be stored to represent the world and information about the state of the world can be retrieved on a demand basis. The most notable weakness of this model is that in many cases information needs to be gathered for a long time to compensate for the noise in the measurements.

- Hybrid

  In the hybrid model both of the above-mentioned world models are used. Generally, with distinctively different control architectures there is the problem of data consistency and association. For example, is an object in the world model the same and in the same place on all levels of control? Efficient distribution of the sensory information has to be considered, as does the possible management of directional sensors.

- Behavioural

  Behavioural control does not dictate whether the world model should be distributed or centralised. Each behaviour still needs its own viewing angle to the environment, which argues for a distributed world model, one for each behaviour. On the other hand, from the viewpoint of the computational resources, only the active behaviours need to have an updated world model and actively filter the raw measurement data. That calls for a centralised model because very often a world model would need to be constantly updated for

the sake of accuracy and consistency. In a centralised model, each perception function would have only one instance, which necessarily would not be the case with independent behaviours.

- Knowledge-based

  In the KBS the measurements from the environment should be filtered all the way to (preferably discrete) beliefs. These abstract states of the environment require more processing power than the other models. Not only the PRS, but many other knowledge-based systems too, rely on the world being represented as discrete states or some other kind of extensive perception. It all depends on the way in which knowledge is represented. The arbitration mechanism for the suitable plan requires the world state to be assessed in one way. The plans that are selected can view the world in another way. Naturally, the arbitration mechanism can use a coarser and more abstract world model than the plans.

Chapter 4

State of the Art in Plan Representation for Service Robots

The control architecture or method dictates the syntax for the representation of the plan. Similarly, the control architecture defines the perception process or, at least, how the perceived world is represented in the architecture. Furthermore, several representations are needed inside one architecture because of the cascaded nature of the controllers.

The plan is an integral part of the task knowledge and thus it has to follow the requirements set in Section 2.6. The other parts of the task, the goal and meta-data, are more difficult to define but easier to represent than the plan.

In this chapter several ways to represent the plan for the robot controller are considered (Figure 4.1). The following sections are divided on the basis of the plan representation used, but are not purely just explanations of the representations. As mentioned, the controller and world model have their influence too.

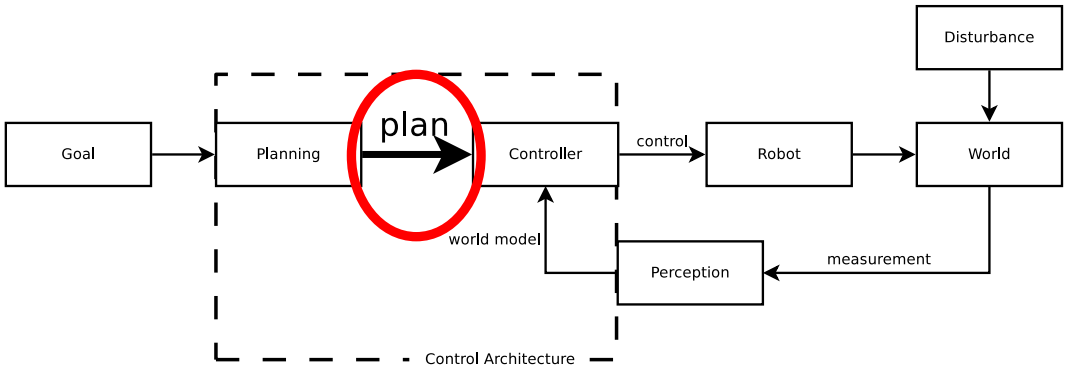


Figure 4.1: *Plan representation in elementary robot control*

4.1 Analogous Systems

There exist multitasking systems and organisations that are analogous to the processes studied in this work. Thus it is justified to ask why these systems are not directly modelled for service robot use. In this section the most apparent concepts are discussed.

4.1.1 Computer Operating System Architectures

The most obvious analogy could be found from computer operating systems dividing the use of the central processing unit (CPU) between many programs. Programs are started, stopped, and even paused and continued independently of CPU scheduling [66]. Additionally, the system responds promptly to interruptions from several peripherals. The computing tasks can even utilise parallel CPUs.

A relevant difference compared to robot task execution lies in the context switch. For the context switching of a computer program, many things need to be done. A program pointer, the location in the memory where the execution process reads the next command from, needs to be stored. The state of the variables used by the program needs to be stored. And, of course, the new program pointer and variables need to be loaded for the new context. But even if the storing and reading has to be done in RAM instead of the CPU cache, this is a fast operation compared to the case of a service robot. In other words, the CPU time can be sliced into very thin time slots for different programs, so fast that for a slower sampling rate perception system, the human eye, the different programs seem to run simultaneously. A task in the real world, especially involving physical material with a significant mass, cannot be changed so fast.

When a program is paused and the current state of the program is stored to memory, it does not matter what kinds of calculations are done before the program is revoked. With robots, the paused task execution cannot start from exactly the same state it was in when the task was paused, if some other task has been executed during the pause. Additionally, CPU resources are often interchangeable from the point of view of the calculation task.

Actually, the closest computer multitasking and resource shearing concept to the ActionPool is the Grand Central Dispatch (GCD) of Apple [67]. In GCD independent parts of calculations are packaged into so-called Blocks. Blocks are organised into queues to define their execution order. A queue can be scheduled to start immediately or triggered by some event, for example a timer. Finally, there is a thread pool symbolising the computing resources, from where a free thread or computing resource is selected from the next block waiting in the queues.

4.1.2 Company Order/Deliver System

Almost any kind of commercial endeavour is a multitasking organisation. Conceptually, it is close to a service robot. There are requests for services or products

coming to the company at a rate which is not directly controlled by the company. The requests can be taken back or priorities can be assigned to them.

The request process in the company context still exhibits differences compared to the service robot. The company's task space, i.e. its selection of services and products, is not directly controlled by the user or customer. The company decides by itself what to offer and the request is based on the selection provided by the company. With a service robot, the user should be able to define what the robot does. There is no description language of how to do it in the company context. Maybe a food recipe for a restaurant might be an exception but it is applicable only in a very narrow field of services and products.

Companies producing custom products or services do not have any formal description languages for their products or services, either. Instead, there is typically an intensive negotiation situation to form an agreement on what the company has to do [68]. And still the agreement does not describe uniquely what needs to be done or how it should be done. This is not an applicable method for robotics.

4.1.3 Military Organisation

A military organisation is a strictly hierarchical structure where the chain of command is one of the key issues [69]. The structure has independently working units where objectives are defined by a body that is higher in the command chain. The organisation can execute multiple tasks concurrently and with different priorities. Usually, the tasks are very well documented, including the way they should be done. Tasks can be started and stopped, as well as paused and continued at will. The military organisation is designed to be a self-supporting system. The organisation has many features that are desirable for a service robot, but there is one fundamental difference. All the resources in the military organisation that can affect the state of the world are distributed and physically independent. The service robot is a single unit whose resources are interdependent.

4.1.4 Computer GUI events

Many computer systems with a graphical user interface (GUI) are based on so called event model. They accommodate multiple applications simultaneously and the resources of the human machine interface (HMI) (the display, mouse, and keyboard, for example) are shared between applications [70, 71]. The basic concept is to assign functions to the GUI events and then just sit and wait for the events to occur. The concept relies on the fact that the functions related to the event would be executed so fast that the users would not perceive a noticeable delay and thus the function can be even blocking. With service robots this is not applicable because all actuator-related functions take a noticeable amount of time to execute.

Computer GUI concepts are also adjusted towards user-driven execution. This means that in principle all the actions, or event-function pairs, the system executes are initiated by an event caused by the user. Sometimes events can be combined into

so-called macros for automatic sequential execution and sometimes the programs can generate events themselves, such as an incoming email pop-up. Still, in the GUI concept most of the functions and even macros are initiated by the user and thus make the user's attention a limited resource for the system execution process. Moreover, the system does not define a plan; the plan is in the user's mind and is executed through the user's actions.

4.2 Procedural

Procedural representation builds on a library of elementary functions of the system. By combining these functions, more complex functions can be created. To form a procedure, a sequence of function calls and flow control rules is defined. The basic rules are conditional branching, forking, and merging. The procedural representation can be visualised with a flow chart (Figure 4.2). Consequently, the procedure's sequence of function calls and flow control rules can be broken down back to elementary functions for execution. This is the way in which most of the general-purpose computer programming languages work.

A computer program listing is a good example of a task representation for a computer. With the right compiler or an interpreter, one can translate the listing into executable instructions for the CPU.

Procedural representation can be considered to be the most common type. Any other representation can be reduced into this or any other representation in a way that is already in a procedural form, since they are typically implemented in a procedural programming language. Additionally, single-task systems can be considered to use procedural representation since their "hard wiring" is often programmed directly in some procedural computer programming language, although in that case it does not exactly meet the criteria of a plan representation being independent from the controller.

There are many different plan representations that function in this procedural way [7, 13, 36, 72, 73, 74, 75] but they are notorious to construct and maintain. Procedural representation is very expressive and as such it is a good general task representation for any kind of actor. This comes with the trade-off between complexity and length of representation. It does not take into consideration the unique properties of the service robot case, in order to reduce the size of the instruction set.

Since procedural representation is so widely used in computer programming, methods have been developed to overcome its shortcomings. There are de facto standard ways of documenting, graphical representation [76], tools for work-flow maintenance, and advanced UIs for editors.

4.3 Distributed

In the distributed plan representation the plan is defined as a set of independent components, independent in the sense that they do not require the presence of other

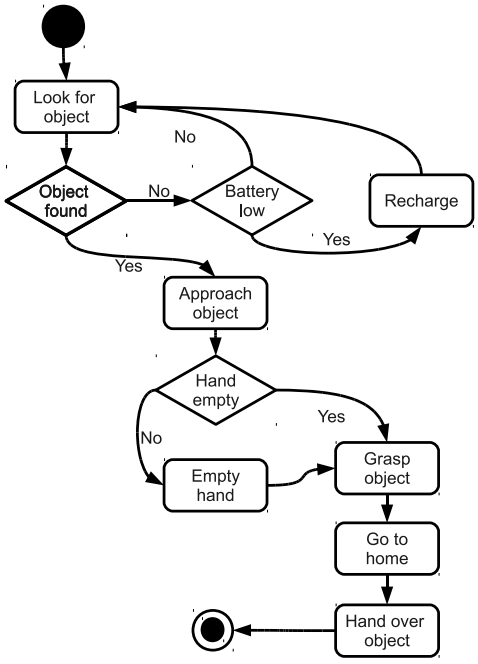


Figure 4.2: *A flow chart description of the procedural programming paradigm*

components and in most cases do not know anything about them. Still, there has to be a mechanism to integrate the output of independent components into the output of one machine. This can be done in a myriad different ways but here they are classified into four types: fused controls, centralised action selection, decentralised action selection, and disconnected controls.

4.3.1 Fused Controls

In fused control the independent processes calculate the control from their own perspective. The actual control for the robot comes from the fusion of all these independent controls (Figure 4.3). The best-known technique for this is fuzzy control [63], although it is not used that much in service robot scenarios. Distributed representation is an ideal way to use parallel processors and it is believed to be close to the biological way of processing information because of the strong parallelism inside the brain. Ideally, it could be independent controllers giving just vectors that are summed to get a reference vector. The most notable early use of this representation was the motor schemes in [77]. In [34] a subsumption was used to control competing controllers and avoid race conditions. Further on, the processing components developed into more complex units called behaviours [42, 78]. This system worked well in simple reflex kinds of survival behaviours, but proved to be difficult to use in more complex scenarios: the manual construction of the plan can be very tedious and difficult.

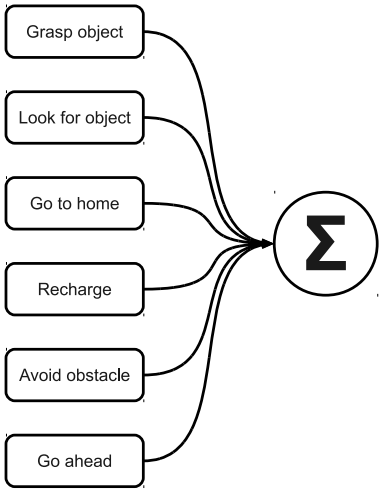


Figure 4.3: *Distributed plan representation using fused controls*

4.3.2 Centralised Action Selection

At some point of a more complex task (i.e. not only navigation), a decision has to made. It is not sufficient to merge the outputs of the components but one of them has to be selected in systems where only one action can be done at a time. The simplest way to perform this action arbitration is to use priorities and the action or component with the highest priority will be selected. A more complicated method would be action voting from among a discrete set of responses [79]. And the complexity goes on in [32], where activation energy was distributed in a system on the basis of the current state of the world and the preconditions of the actions, as well as the assumed effect of the action enabling other actions to take place. The action with the most activation energy was selected.

4.3.3 Agents: Decentralised Action Selection

A kind of predecessor of networks of behaviours (discussed in 4.6) is agent-based control. The robot's functions are considered to be independent software agents that communicate with each other. An agent could be imagined as a more developed and independent behaviour. The agents can negotiate to avoid the race conditions [64] and the whole control fusion element can be omitted. For example, the agents can pass the control privilege token between each other [80].

What is common to both agents and behaviours, at least on the ideal level, is that each agent or behaviour has its own understanding of the state of the world and they

only gather information that is relevant to their own decision making. This could potentially reduce the processing needed for perception but similar agents often process the same information many times. One of the strong points of distributed control is its robustness. When one agent or behaviour is disconnected, the system can still be operational with just that one particular feature lacking.

4.3.4 Disconnected Controls

When the underlying system can be further abstracted, the components do not directly compete for the control of the machine. The components are disconnected from the direct control. The underlying infrastructure will handle the action allocation and scheduling to the resources. This comes with the trade-off that the component does not know when or how its control takes effect. Besides, the component's control signal can be on a very abstracted level.

In classical AI this is done by utilising goals. The goal is not just an expression of the desired outcome, but could also be utilised as an instruction. The boundary condition is that the underlying infrastructure has to be able to interpret that goal. The goal can be broken down into smaller sub-goals and several goals can exist at the same time. It is very convenient to represent the tasks just as a set of goal states for the world, such as "full cup of hot coffee on my desk" and "snow cleared from the driveway", and it would be intelligent if the robot could figure out how to achieve those goals. And PRS tries to do just this [81].

In PRS there is a set of skills or knowledge of how to change the state from one to the other with some preconditions. Then it is just a problem of finding a matching chain of skills to apply in order to get from the current state to the goal state. But the parameterisation of the world into comparable state presentation is a great challenge.

Another classical AI technique called a *partial-order plan* is marginally or partially distributed. It is defined by a set of actions, ordering constraints, and causal links [26]. On the basis of the ordering constraints and the causal links, several actions are issued at the same time but their execution order is defined by the underlying infrastructure. The *Markov Decision Process (MDP)* defines a plan or policy as a set of state-action pairs. The *policy* defines an action to be taken in a certain state or belief state and these state-action pairs can be seen as distributed components. The effect or latency of the action does not impact on the policy as such. One drawback of MDP for service robotics use is that it assumes the system to be a Markov process with discrete unique states, which can be difficult to formulate. Even with *partially observable MDP (POMDP)*, with probabilistic multi-state hypothesis, the action is based on only one state [82], although technically the policy can be applied even when there are multiple simultaneous states.

Many robot control techniques deploy distributed components as a part of the plan. In [59] the *RAP (Reactive Action Packages)* plan spawns monitoring processes or system-level *patches* that can create and terminate a monitoring process. The approach in [83] utilises a set of relations (*signal* or *forward*) and a directed acyclic

graph (DAG) of actions (*tasks*) as a plan. The DAG is a functional representation but it is just half of the plan. The relations work as distributed components to propagate events and interact with the DAG.

4.4 State Diagram

The state diagram, in its purest form, is a system described with a set of discrete states and conditioned transitions between the states. In most cases a specific initial state and final states are used. The system can be presented as a graph such as in Figure 4.4. The renowned problem with the state diagram is the state and transition explosion. The complexity of the state diagram grows faster than the complexity of the problem. The work by [84] addressed these problems by introducing especially the superstate and orthogonal state. With the superstate a common transition for a group of states can be expressed, reducing the number of transitions. With orthogonal states two simultaneous discrete states can be expressed and the number of states reduced. With the enhancements, the state diagram starts to approach characteristics of a flow-chart. To highlight this distinction, the enhancements are not considered here.

The state diagram can be used to describe a state machine. But here we are discussing the plan representation and the distinction between it and the control mechanism per se has to be recognised. The state diagram and its derivatives are an intuitive and simple way of representing plan logic and they are used in robotics [85, 86]. The state diagram representation can even be abstracted into a state in a higher level state diagram. The state diagram can be formulated into the mathematically convenient form of states and connections, for example for automated processing [45].

Even though the state diagram is very suitable for some cases, such as describing the task of automatic vacuum cleaning, it has serious drawbacks for more complex tasks. In the state diagram it is assumed that the whole system is only in one of the states at one time. This is a valid assumption if the robot's only functionality is mobility, as in the vacuum cleaning case. But with a more complex case of mobile manipulators, there are parts that can change their state while the robot's mobile base stays in one state. In other words, the state diagram occupies all the resources of the robot, whether they are needed or not. Furthermore, the robot can have several distinct internal states, such as "the battery is low" or "the location is unknown". Of course, simultaneous distinctive states could be merged into a single state representing their combination. Inevitably, this would lead to the state explosion.

Another drawback of the state diagram becomes apparent when multiple tasks need to be performed. Plans for the tasks have to be executed one after another, because there is no defined interaction mechanism between two different plans. For parallel task execution tasks have to be combined together in the plan definition phase and multiple tasks have to be merged into one giant task representation. That is difficult for the development because multiple developers cannot merge their work

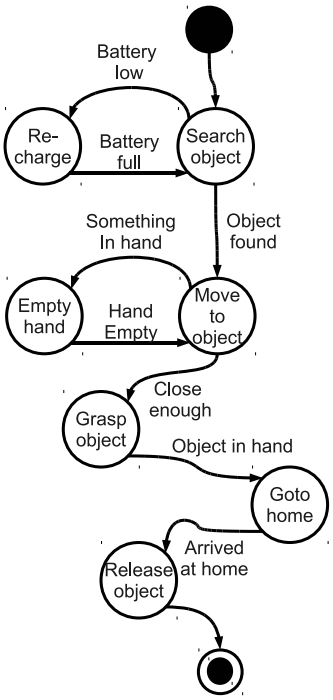


Figure 4.4: *An example of a state diagram describing a fetch task*

easily. [87] offers some help for this by introducing the so-called common state concept, where different state diagrams can be merged together by combining them from a common state point. In this way the developer has to specifically define the merging points and the merging is limited to those points and the mechanism does not allow priority adjustments.

Maybe the worst drawback of the state diagram in service robot use is that it does not allow variable passing. The states and their transitions themselves hold all the relevant information. In other words, by virtue of its general form, the state diagram defines only the static part of the plan or task. To overcome this restriction, the state diagram has been augmented in practical solutions [34, 84].

4.5 Tree structure

A tree structure is one type of graph that can be used to represent a plan. The tree graph is constructed from nodes that are connected with edges. The tree graph forms a tree-like structure with one root node to start from and edges dividing into child nodes and finally ending up in so-called leaf nodes that are connected only to their parent node. Compared to the tree analogy from nature, the tree graph is often presented upside down, with its root at the top and leaves at the bottom.

The tree structure is used mainly in describing the plan decomposition based

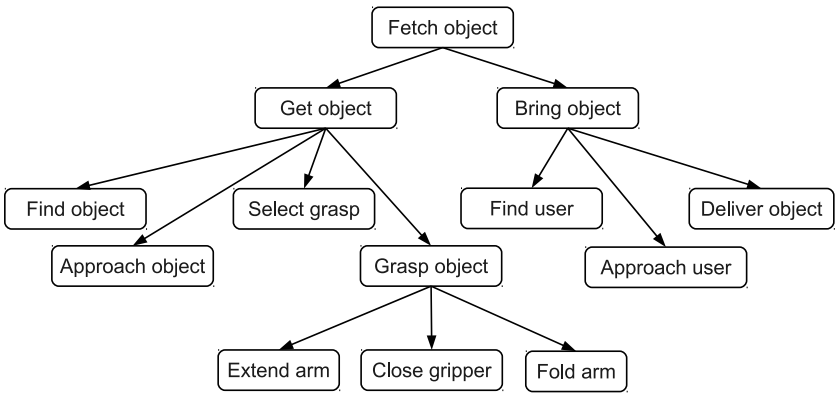


Figure 4.5: *Tree graph describing the Hierarchical Task Network of fetch task*

on the *hierarchical task network (HTN)* concept. The HTN divides the task into sub tasks until the division reaches the executable actions (the leaves in Figure 4.5). The plan can be divided into HTN automatically [88, 89] or manually [90, 91]. HTN representation has been used to document the program flow of task execution [92, 93, 94].

The tree structure can also model the deduction of an action from the current situation. Each node is a question about the environment and each branch is a candidate for the answer. The leaves are then the final actions again[95]. A *binary decision diagram (BDD)* [27] can be used for similar mapping from a state to one of the available actions.

In the tree structure it is assumed that the execution is not interrupted. When the robot is operating in a dynamic environment, interruptions are unavoidable. No mechanism is presented for re-evaluating the situation before one of the operations is finished. The tree representation grows unnecessarily large in the case of loop-type procedures, where parts of the tree have to be copied instead of backward links being provided. In the event of the parallel execution of multiple tasks, there is no defined mechanism for combining tree structures.

4.6 Functional

Increasing numbers of robots are controlled by using some middleware components [9, 22, 23, 24, 25]. Middleware is used to construct a hardware abstraction layer (HAL) and build components that provide measurement data and accept control signals. Furthermore, they may provide a signalling channel for higher-level control modules and components for some basic functionalities. For example, components can be added to filter the measurement data into the robot's position and provide that information to other components (Figure 4.6). HAL and middleware components

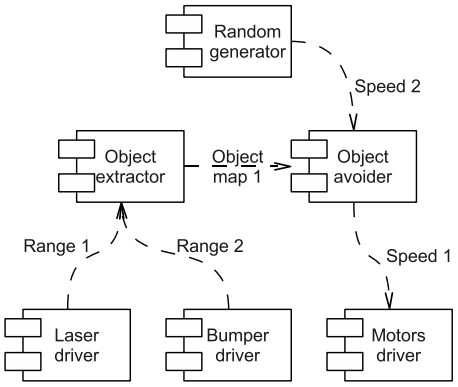


Figure 4.6: *Component diagram of middleware components to achieve a random walk behaviour*

provide a so-called open architecture, where the floor is open for the organisation and activation of the components according to any of the various control methods presented in Chapter 3. Middleware is typically used underneath the "real" control architecture.

Just by constructing and connecting components in some novel way, a task execution performance can emerge from the system. This resembles the declarative functional programming paradigm (such as Erlang or Haskell), where there are no specific state variables and the execution is just based on the inputs and outputs of the functions. When using a procedural programming paradigm (in C or Python) you are in a certain state of the procedure. The functional paradigm is based on data flow, while the procedural paradigm is centered on the control flow. LabVIEW [96] and Simulink [97] also utilise this functional paradigm.

This is a kind of special case of distributed representation. The difference is that there are no competing control signals and no need to evaluate the appropriate control from many options. That makes the functional representation more rigid than the distributed one.

When the plan is composed just from these components, the system is typically a single-task robot. Expansion to a multi-tasking robot would elevate the component network to an unmanageable size. One way in which this is a very nice plan representation is that typically the organisation of components is serialised into one configuration file. Middleware uses the file in the start-up of the system. When using this kind of file, changing to a different configuration can be difficult without restarting the system.

In most middleware the components can be started and stopped independently and dynamically. In this way the plan can be changed on the fly, but then again, the mechanism to change the plan has to be represented somehow. In Saphira [98] it is done with the procedural Colbert [75] language translated into a state diagram. RAP [59] uses goals and the plan manager in [83] uses conditional rules to activate and deactivate the components.

4.7 Trajectory

In particular, when movements are planned for the robot's base or the manipulator, the resulting plan is typically represented as a trajectory. The trajectory can be defined in several ways. It can be an ordered set of points or way points in the operational environment or configuration space of the robot. It can be a continuous function of time in the same space or it could be a set of points or a continuous function in the robot's joint space. Then the next point in the set or the value of the function in the given time instant is used as a goal state for the robot [99, 100]. In classical control theory [101] you have a myriad controllers to pursue the goal.

This is a method that works well in closed systems. For example, a factory environment can be structured into a closed system, where you can control, observe, and maybe even predict the state of the system. But in our problem setting we have an open system. The complete state of the world cannot be controlled, observed, or predicted very much and even if it could, it would be an impractically large vector to work with.

One way to produce these motion sequences is to move the robot manually while the robot logs the pattern. Another model is that the human moves and the robot records, for example, the motion of the fingertip and creates a log of that. This is called programming by demonstration [102, 46]. By following the log, the robot can reproduce the motion. Plans can be constructed from a set of these logs. In order to be able to execute different trajectories one after another, the final pose of the first trajectory has to be the first pose of the second trajectory. This is achieved by defining a home position from which all trajectories begin and finish. Naturally, alternatively some limited set of fixed starting and finishing points can be used, but the number has to be kept low to facilitate log or plan interoperability.

This representation has been used for a long time in closed environments and with non-mobile robots such as a robot arm in a factory environment. Humans also use a similar approach in some closed domains. For example, in the game of badminton, a player tries to be in the middle of the court in the "home" position, from where the player moves to hit the shuttlecock and returns to the "home" position after doing so (ideally, at least). Logs are computationally light since they can be produced offline and that must be one of the reasons for their early adoption in industrial use.

As mentioned, planned trajectories are best suited for static or predictable environments. Furthermore, when the platform for the robot is mobile, the same arm motion cannot be performed in every location of the environment. In a dynamic environment with humans, motion patterns have to be adopted to the current situation. With the increase in computing power, these trajectories can be calculated on a demand basis. But in a dynamic environment the trajectory becomes obsolete. There are two common ways to overcome this problem: global re-planning or local altering of the trajectory or path. Global re-planning is a computationally expensive process and a local alteration, plan repair, is preferred. In a local alteration just the proximity of the robot is considered in the state vector for the controller.

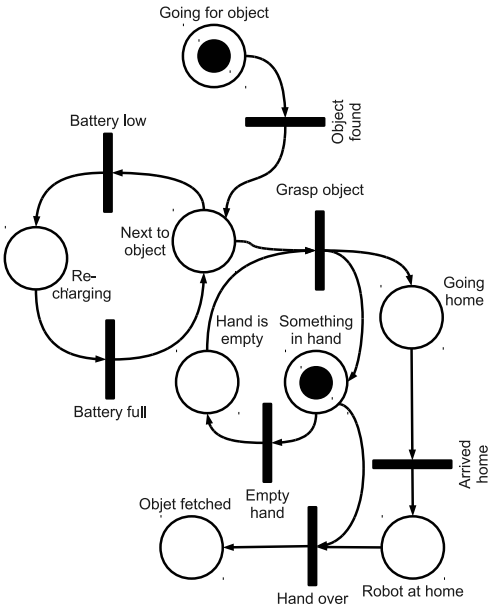


Figure 4.7: *Example of Petri-net describing a fetch task*

4.8 Petri-net

Petri-net is a graph model used to represent a discrete event system (Figure 4.7). A service robot operating in a dynamic environment is very much an event-based system so the usage of a Petri-net sounds like a natural choice. Additionally, a Petri-net can easily be formulated into a formal expression and manipulated mathematically [103]. The Petri-net was developed by the young Carl Adam Petri for the purpose of describing chemical processes. It has been used to describe a task behaviour of a robot, for example in [104, 105].

A Petri-net is constructed from places and transitions. To traverse a Petri-net, tokens are used to represent the current state of the system. When there is a token in every place with an arrow leading to a particular transition, the transition can fire. Similarly, after the transition has fired, the tokens are removed from the places leading to the transition and placed in places with an arrow coming from the transition. There are variations of Petri-nets allowing multiple tokens in one place and allowing multiple arrows or weighted arrows going between a transition and a place.

The drawbacks of the Petri-net are similar to those of the state diagram, even though it is more expressive than a state diagram. Tasks cannot easily be merged and variables cannot be passed. The lack of abstraction in the Petri-net causes problems from the usability point of view because the network can grow to be unmanageably large.

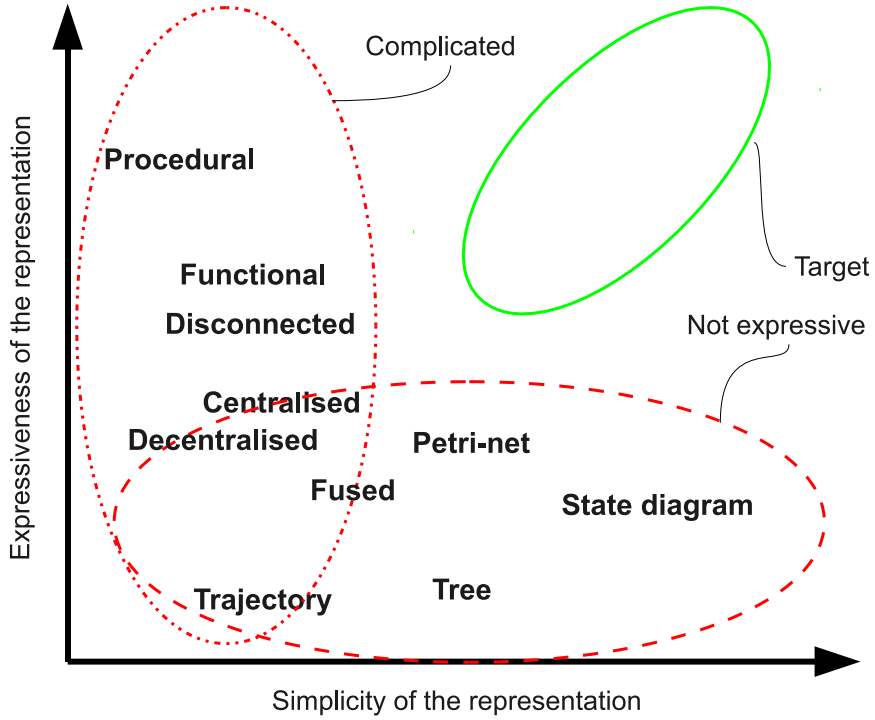


Figure 4.8: *Comparison of plan representations in simplicity and expressiveness* The figure illustrates only the simplicity versus expressiveness paradigm. The mapping of the different representations are based on author's subjective evaluation from the practicality point of view.

4.9 Conclusions

Figure 4.8 maps the presented plan representations on a simplicity versus expressiveness map. These are the trade-offs for the representations. It is difficult for a representation to be expressive and simple at the same time. The representations are either complicated or not very expressive. Another important matter is the practicality issue. Tasks need to be developed incrementally and independently of each other. This is not a challenge only for the plan but also for the system executing the plan.

Despite the myriad task representations described above, none of them *alone* can be considered a good way to represent a plan. Parallelism, abstraction, temporal constraints, or ordering constraints, and, generally, the requirements mentioned in Section 2.7 can not be met with just one formalism of plan representation. Representations has to be mixed. For example, by defining a state diagram in two stages, first as a regular state diagram defining ordering constraints and then tying down the variables in the functional fashion. Still, the parallelism is not handled. The mixing and merging of the basic representations aims to produce the simple and expressive representation. Another interesting example is the IEC standard 61131-3 [106], in

Table 4.1: Summary chart of different plan representations

Plan representation	Implementation	Controller
Procedural	Set of functions, their ordering and flow control components	Instruction interpreter or compiler
Distributed:	Set of independent components	
Fused controls		Fusion of component outputs
Centralised selection		Arbitration of component outputs
Decentralised selection		Components negotiate output
Disconnected		Scheduling of component outputs
State diagram	Set of states and transition conditions	State machine
Tree structure	Dependencies in a tree graph	Operations or decisions travel through the tree
Functional	Set components and their typed connections	Emergent function from the interaction of connections
Trajectory	Desired state vector listings	Classical feed back control
Petri-net	Petri-net diagram or set of places, set of transitions and set of arcs	Petri-net interpreter

which the programming of the operations in the industrial setting is done utilizing five different representations.

A popular way to merge different kinds of plan representations for service robots is to use them in different layers. With the layers, the simple representation can be isolated from the complex expressive representation. For example, in some established control architectures: Saphira [107] uses functional representation on the lower level and a procedural language [75] compiled into a state diagram on top of that. The classical AuRA [61] architecture utilises distributed representation with fused controls (motor schemes) combined with a state diagram on top of it and inside the components (schemes). Similarly, 3T [108] has skills organised in functional representation as the bottom layer. The skills are commanded by the procedural RAP [59, 109] language spiced with some distributed disconnected control, as discussed earlier.

A summary of the different plan representations is provided in Table 4.1.

Chapter 5

Representation and Control of Task in ActionPool

Like most of the research today, this has a relatively long track record. The design and motion control of service robots and work machines, the tasks to be conducted, and the user interfaces have been studied for around two decades within the research group. The group has now been formed into the Finnish Centre of Excellence in Generic Intelligent Machines (GIM) Research. Currently, the research still revolves around the same topics, but the hardware is changing and more complex and abstract parts of the service robot scenario are being studied.

When the task execution was investigated further, a common feature became evident. Almost in every task, the robot was first driven into some location, where it did something, and then it was driven to another location to do something else. All the tasks and phases of the tasks started with the same command, "go-to somewhere". This became a cornerstone of the abstraction. For the task, a unit of operation was formed from a location, together with something to be done in that location. The location is, of course, relevant to that something to be done. The location does not have to be an absolute co-ordinate in space. It can just be a reference to an object with a location. The unit of operation was named an Action in this context and the something to be done was named aPlan (Action plan). The plan for how to do the Task itself is formed of Actions and called tPlan (Task plan), in order not to be confused to the aPlan. The tPlan of the Task together with metadata as a header is called simply a Task.

Some terms specified and used in the ActionPool are words with some general meanings. To identify that the word is actually the name of a term and does not refer to the word's general meaning, the first letter of the term is capitalized. Figure 5.1 illuminates the relations of these terms.

In principle, the aPlan could be implemented utilising any of the representations mentioned in Chapter 4. An expressive procedural representation is favoured to compliment the simplicity of the tPlan representation. The aPlan is on the hardware abstraction level, while the tPlan is on the level where the parts of the robot are commanded by means of single operations.

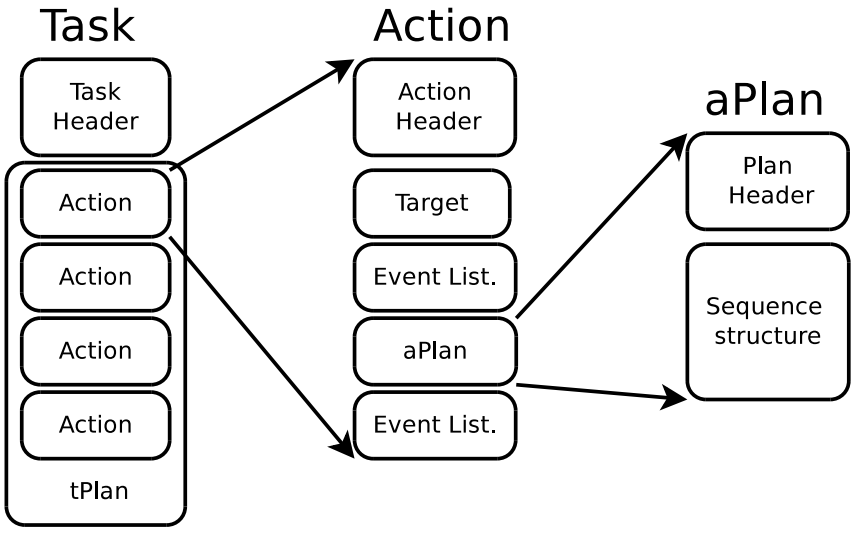


Figure 5.1: *Breakdown of the Task structure*

Another key observation was that only one resource was used at a time. The common robot workflow went as follows: first the robot moves somewhere, then it looks at something and then starts to move its manipulator. In principle, each of these operations could be done simultaneously. In practice, the operation is restricted by the actuated parts of the robot that it needs to use. The parts are bounded into a restricted section of the space on the basis of where on the robot the part is attached. Thus the only independent part of the robot is its mobile base, which is still bounded by the environment. These robot parts are called Resources in this context.

The architecture that is created could be classified as a knowledge-based system with some resemblance to PRS [30]. It can perform time-sharing and concurrent multitasking utilising an Action as a token to the robot's resources with similarities to the token-passing approach in [80]. An overview the ActionPool is given in Figure 5.2.

5.1 Division into Resources

Resources naturally form a tree-like dependency structure, where the pose of the robot's mobile base is the root node. One definition of a Resource is that it occupies space and can be actuated to change its pose in the space and is reusable, i.e. it can only be in one place at a time. Thus the Resource is not only mutually exclusive but different tasks would need it in different locations. Examples of Resources are the above-mentioned mobile base, a camera with pan-tilt-zoom capabilities, and a manipulation arm. The Resource division could go further down. For example, the manipulation arm could be divided into arm segments and a gripper and each one of these could be considered a Resource. When a Resource is utilised in the proposed architecture, a fixed set of functions is used:

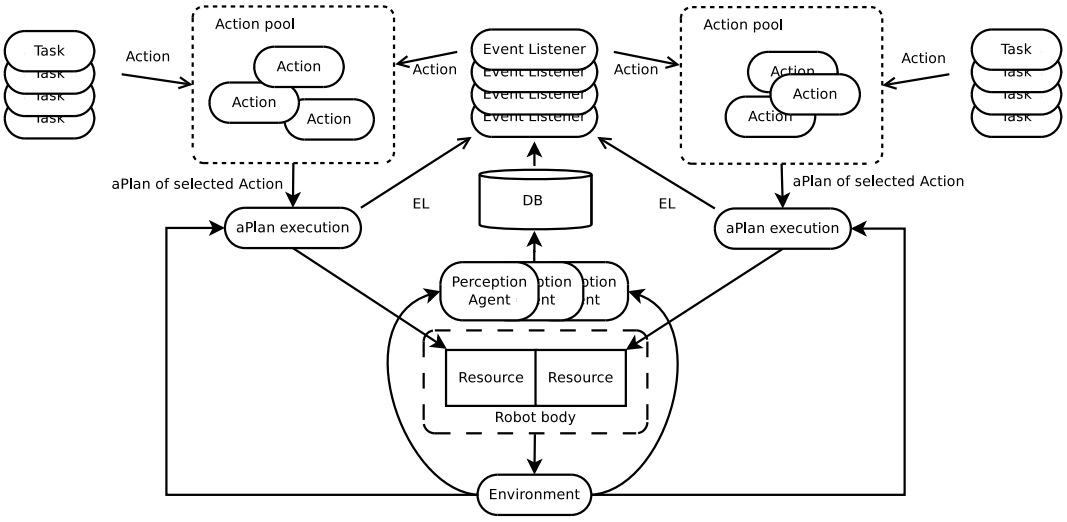


Figure 5.2: *Overview of ActionPool method* See the text and Table 5.2 for explanation of the component names.

- query for current state
- change the state
- cost estimate of state change

The state of the Resource is formulated into a pose in the space. The state change function and the cost estimation function then need a pose as a parameter. An explicit pose or reference object can be used for this purpose. The functions can retrieve the pose related to the reference object from the world model database (described in Section 5.4). In this way the system can adapt dynamically if the pose of the object changes. When the resource is abstracted like this, the same implementation of the ActionPool (which is described in Section 5.2) can be used for managing different Resources.

5.1.1 Context Switch

Any multi-tasking system using limited resources has to share the resources between different tasks. This is time-sharing multi-tasking. The operation to release the resource and pass it to the next resource is called a context switch and it needs to be carefully designed. In the proposed architecture the command to change the state of the Resource is the context switch function. It is noticeable that, unlike in the computer architectures, the current state in Resource usage is not explicitly stored anywhere at the time of the context switch. We cannot return directly to the saved state of a robot in the same way as can be done with computers. For example, in the Action of opening the fridge door, it is not necessary to remember if we were already gripping the fridge door handle or still reaching towards it when interrupted. To continue that, we would have to start from the beginning anyway. The state of

the Task, i.e. which Actions are executed and which are not, stores this information implicitly.

Normally, a context switch occurs after an Action has finished. Actions naturally form safe divisions of resource usage. If an Action using the Resource is interrupted by some important event, the Action is just considered undone. The next time the Action is selected for execution again, it starts from the beginning, all the way from Resource reservation aka context switch. This is a way to ensure the safe restoration of the state of the Resource. The aPlan of every Action has a cancellation routine called at the time of the interruption for a safe winding-down process. Potentially dangerous Actions should also have Event Listeners (described shortly) that monitor the hazard in the case of internal or external interruptions to the process.

A context switch is necessary to accommodate time-sharing multi-tasking. But when considering the overall performance, the time spent on the context switch is non-productive. The Action selection process in the proposed architecture favours Actions with shorter *context switch times* from the state of the Resource at the time of the selection. If the state after the Action could be estimated, some planning and more far-reaching Action selection could be made on the basis of the minimisation of the context switch times. Perhaps the location assigned for the Action could be used as an estimate for the state after the Action too. Unfortunately, this has had to be left for future studies.

5.2 Actions and Action Pool

Each Resource can execute Actions by itself, but only one Action at a time. After all, one cannot be in two places at the same time. Although the Resources are physically interdependent, they can perform Actions independently. So, a robot with multiple Resources can do concurrent multitasking. There can be several Actions waiting for execution by the Resource. The data structure to keep track of and manage the set of Actions available for execution by a particular Resource is called the *Action pool (AP)*. The AP communicates with the other components of the system, which will be described shortly. In effect, the AP is an abstraction of the Resource. It manages the usage of the Resource by holding all the Actions pending admission to use the Resource.

The granularity of the Action and its aPlan is defined as an operation with the Resource that cannot be successfully completed if interrupted by some other operations with the Resource, i.e. it cannot divide the Resource. This is not a complete definition but rather a design principle. There is no built-in system mechanism to utilise contingency because of the the lack of understanding of the surrounding world and the effects of Actions on that. The plan inside the Action should consider the possibilities of fortunate occurrences, i.e. the Actions would not be skipped, but they can be executed very fast when nothing needs to be done. In the case of more common and significant contingencies the Event Listener could also be brought into play.

Each Action is a part of some Task, which is also described in the following sections. An Action is selected on the basis of a comparison value function evaluated for each Action in the pool. The behaviour of a robot can be adjusted by varying the selection policy, for example, by being greedy and always selecting the most valuable option or exploratory by favouring Actions that have not been taken very often. Some human-like behaviour could perhaps be achieved by roulette selection, where Actions with a higher value are more likely to be selected. The value function could be seen as a predictive projection into the very immediate future and thus as planning. On the contrary, the value function does not consider the possible consequencies of the Action, except the time it would take, and thus this cannot be considered to be genuine planning. Actually, with greedy selection this could be close to *minimum slack scheduling*, which is considered a fairly competitive option for fully-fledged planning in complex and dynamic domains [26].

The value function of the Action is based on its importance and expected execution time. The importance is a user-given priority p. With the priority safety-critical or otherwise urgent Actions can overrule some regular activities. The expected execution time has two components with their respective uncertainties as standard deviation.

1. Context switch time, which has two sub components:
   (a) Resource reservation time (t_r and its variance v_r), i.e. the time it takes to move the Resource from the current state to the state where from the aPlan can be started. This is a Resource-specific estimation function.
   (b) Wind-up time of the Action currently being executed (t_w and its variance v_w). Currently, this is just a static estimate.
2. Expected execution time of the aPlan. This is statistically derived from all the times the Action is executed. (t_p and its variance v_p)

One example to evaluate the comparison value λ is

$$\lambda = p + e^{\frac{-t}{t_{ave}}}, \text{where} \tag{5.1}$$

$$t = \begin{cases} t_p + t_r + \sqrt{v_p + v_r} & \text{if running Action} \\ t_p + t_r + t_w + \sqrt{v_p + v_r + v_w} & \text{other Actions in the pool} \end{cases}.$$

To balance out the effect of a Resource's temporal characteristics, the average context switch time of the Resource (t_{ave}) can be used to normalise the expected execution time. The priority p is initialised when a Task is started and is inherited to its Actions. When the first Action from the Task is executed, the Task's priority p is raised to ensure that the executing system continues to focus on the current task at hand. p can also be raised as a function of time to allow time-based scheduling. Thus, even actions with a lower level of priority will be executed as long as their execution time is significantly shorter than that of Actions with a higher level of priority. For example, the robot is given a high-priority task of fetching water from

the kitchen, based on the above description, while in the kitchen it can also execute the lower-priority Action of checking that the stove is turned off before taking the Action of bringing back the water.

Action selection is initiated by the addition of an Action into the pool or by the removal of an Action from the pool by a user or by the AP. The AP removes the Action after it has been executed or as a reaction from the Event Listener, described later on in this chapter. To manage the Tasks and the execution flow, some special Actions are needed. One is a *placeholder Action*, which cannot be selected by the Action selection process. Only some external event or intervention can remove it; in this way the Task can be "on hold". The other type of special Action is a *remote Action*. That and some other aspects of the Actions are elaborated further in the Task section, after some other components of the ActionPool method have been explained.

5.3 Event Listener

Initially, the tasks were defined in a specific procedural language [12, 13]. The next observation about the nature of the tasks was that there were a huge number of repeated condition checks cluttering and complicating the plan. Typically, condition checks needed to be executed very frequently and exactly the same condition check was present in many different tasks. An example of these condition checks could be if a human comes too close to the robot. This kind of condition check is typically an unexpected event that the task execution has to be aware of.

The condition check and related response was then encapsulated into a component called *Event Listener (EL)*. Event Listener acts in parallel with the execution of the aPlan. EL can be distributed to other computers and its execution loop frequency can be adjusted (Figure 5.3). EL also contains a response to the event as a set of added Actions or Actions to be removed. One common procedure is to restart the Action and that would be achieved by the removal of the

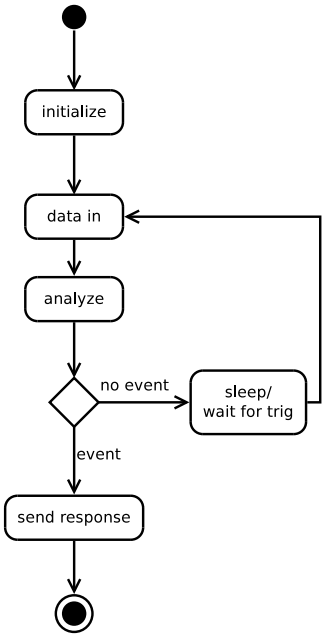


Figure 5.3: *Flowchart of the execution process inside Event Listener*

Action currently being executed and adding it again to the AP. It would most probably be selected again since it was already selected and the context change time is short. The EL could be compared to the exception-handling semantics in general programming languages.

The starting and stopping of ELs is embedded into the Action before and after the execution of the aPlan. The EL functionality is defined separately from the Action and the Action carries just the reference to the EL, the threshold level for the event, and the response. In this way the same EL implementation can be reused by different Actions. The usage of the same implementation in several places ensures the maximum number of iterations for the event recognition algorithm. This is desirable for optimisation and learning algorithms that are dependent on a large number of iterations, as one of the most fundamental abilities for the service robot is to recognise these events.

5.4 World Model

A typical service robot has multiple sensors that can complement each other. The information extracted from each sensor needs to be fused to build a picture of the situation and utilise the heterogeneity of the sensors. The sensor sets and their placement and fields of view are also different from robot to robot. Still, on a more abstract level the system should see the sensors similarly, despite the diversity. To accommodate the demands, a unified object-level world model *database (DB)* is required.

The world model portrays the current situation and does not predict its future states. The sensor readings are refined into observations about objects by a separate process that only posts them into the model. If some predictions are needed, it is the job of the process that needs the prediction to derive the estimate from data available in the world model. A typical operation is to estimate the location of an object on the basis of its last known position and velocity. This kind of DB separates the representation and the process of collecting the data. The perception processes can be allocated dynamically and the sensors of different robots, or even their databases, could be incorporated transparently.

This type of sensor fusion has a few main challenges: a) the observation about the object has to be correctly associated with the right instance in the database; b) the object has to be recognised or classified correctly as being what it is; c) the data from the sensors can support multiple hypotheses about the state of the world and the database should accommodate this; d) the sensors that are used do not provide absolute information and their limitations should be made clear.

These are partly tackled by the recognition process and partly by the database offering statistical probability data about the spatial state, size, or existence of objects (see Table A.2 in the appendices). The spatial state is defined as an estimate of the current pose, velocity, and acceleration. The pose is represented as a union of Gaussian distribution and uniform distribution. A multi-hypothesis for one object with multiple Gaussians is being considered, but has not been implemented yet.

Figure 5.4: *View of X3D file created from database as a snapshot of the robot's understanding of the world*

Another feature of the world model is that objects are in a hierarchical tree structure and their pose is indicated relative to the object's parent node. This is to prevent the error accumulation in the position of the object from co-ordinate transformations when compared, everything being fixed in one world co-ordinate frame. Tree structures have some other benefits related to the data updating and handling, but these lie beyond the scope of this work.

As mentioned, for the ActionPool system the DB provides abstract sensor readings. It can also hold other abstract data, such as the names of places. The other components of the system can use the DB as a global memory and indirect communication channel to coordinate their interactions. As Brooks [56] said, *"The world is its best model"* and promoted tasks and robots interacting through the environment instead of explicit coordination and communication. For a more abstract level of control a more abstract representation than the world itself is used. Otherwise the principle is the same; just a more machine-readable version of the world is provided.

The DB is also used to integrate a human user into the robot's execution. It is very convenient to view what the robot "sees" and manually input objects into the world model, i.e. tell the robot where something is. Visual representations of the world model can be created to observe the robot's understanding of its environment. A static snapshot of the world model was implemented using the X3D file format [110] (Figure 5.4) and a more dynamic version that provided the possibility of a human operator manipulating the world model was realised utilising the Panda3D game engine [111] (Figure 5.5).

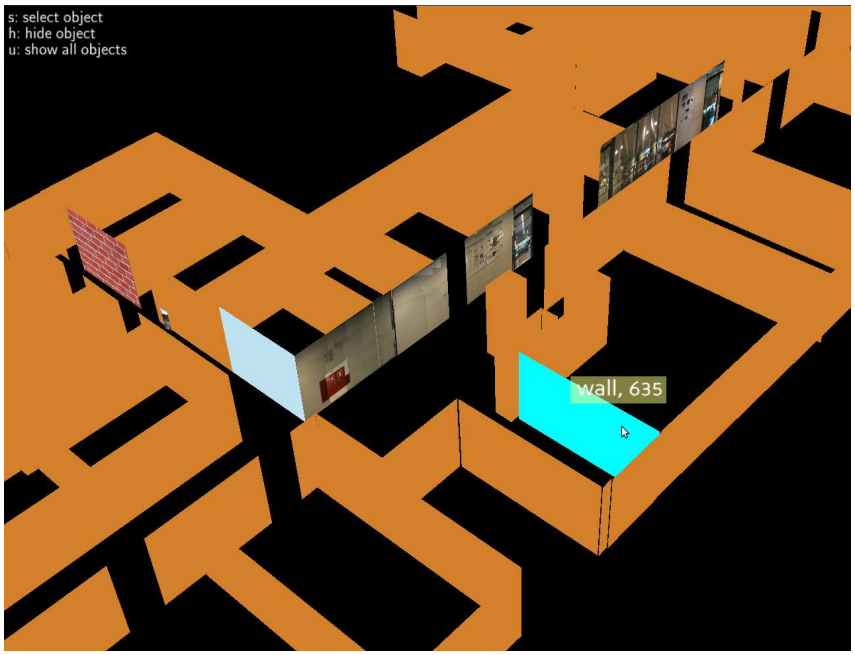


Figure 5.5: *Screen shot from dynamic presentation of the robot's world*

5.5 Perception Agents

Event Listeners rely mostly on the information in the database. So it is important that the data are up to date. Initially, ELs updated the DB by themselves. This seemed convenient for the computing resources because only the information that was actually used at the given time was gathered. Unfortunately, several ELs from different tasks executing the same perception process exhausted the computing resources. Naturally, it is more efficient to use only one instance of one perception process and just the output of the perception is evaluated via the DB.

For this purpose, independent Perception Agents were created. The Perception Agent reads information from sensors and filters it to observations from the environment and stores it to the DB. Currently, Perception Agents run continuously at the rate at which information from measurements comes available. It is clear that when the number of these agents grows, some scheduling for execution is required or the computing resource would be exhausted again. The scheduling should be dictated by the active ELs but it is also left for future work.

5.6 Task

The plan in the Task (tPlan) is represented as a series of sets of Actions. Conceptually, it is in the form of a *partial-order plan* [26] and with the taxonomy of Chapter 4 it is a combination of disconnected distributed and reduced procedural representations. The sequencing is a series of sets of Actions and the sets are ordered with a combination of *and* and *or* rules. If an Action n in tPlan were represented as A_n, an

example of a tPlan with three sets could be $[A_0 \wedge (A_1 \vee A_2), A_3 \wedge A_4 \wedge A_5 \wedge A_6, A_7]$. This differs from the generic ordering constraints (A_1 comes before A_5), which require fewer declarations but lack some flexibility. Additionally, the representation that is used allows alternative Action sequences, while a simple ordering constraint assumes that all the Actions in the Task plan are to be executed.

For the Task execution all Actions from one set are added to the AP at the same time. The next set is added after all the Actions from the previous set have been executed. The so-called non-deterministic excluding configuration, i.e. "or:ing" of Actions, is managed by the AP component. For example, in the previous tPlan representation sample, if A_1 is executed, A_2 is also removed from the AP.

Actions themselves define the *causal links* of the partial-order plan as ELs. The EL can protect the state in the world that the Action has achieved and, for example, add the Action again if the state has been violated.

In addition to the plan, the Task always contains a priority and a description. The priority indicates the importance of the Task, while the description is human-readable documentation about the meaning of the task and its *goal*. Furthermore, each Task can have a set of parameters that are forwarded to the Actions and their aPlans. There is a special kind of parameter called a target, which can refer only to a DB object which can be instantiated dynamically. A priority is assigned each time the Task is started and all the Actions of the Task inherit this priority. An overview of the Task's structure was given in Figure 5.1.

The Actions have some fixed form of pre-conditions (possibility axioms) and post-conditions (effect axioms) in them. Conditions are not explicitly described but they are so common and universal in this context that there is a fixed way of representing them (Table 5.1).

Table 5.1: Definition of conditions in the Action

Possibility Axioms	**Effect Axioms**
Location (explicitly stated in the Action) or object (reference to the DB)	Temporal order specified in the tPlan
Temporal order specified in the tPlan	Conditions to be held as a list of ELs to be started or stopped after the aPlan
Conditions to be held during the Action as a list of ELs to be started or stopped before the aPlan	Human-readable description of the Action's effects

The overall functionality of the AP architecture was presented in Figure 5.2. From the figure two cascaded control loops from the environment can be seen:

first, the aPlan-level control loop with direct observations from the environment, and second, the somewhat slower loop through Perception Agents and ELs for the Task-level control.

5.6.1 Interdependency

As already mentioned, a service robot typically has interdependent Resources and each Resource has its own AP to manage its use. A Task is started only in one of the APs but sometimes the Task needs Resources managed by other APs. These interdependent resources are also handled by Actions. The so-called remote Action is sent to the AP of the desired Resource. The process starts with placing an Action to the original AP, representing the request to the other AP (*remote AP*). When it gets selected, there are three types of remote Actions that can be sent to the remote AP.

1. **Unmonitored Action** The Action is just sent and it does not matter when it will be executed; the Task can continue in the original AP without interference.

   If the unmonitored Action is interrupted, it is treated just like other Actions in the remote Action pool. If the original task is removed, the remote Action just stays in the remote pool.

2. **Monitored Action** A placeholder Action that cannot be selected is added to the original pool and removed after the remote Action is executed. The Task cannot add Actions from the next slot before all Actions, whether local or remotely monitored, are executed.

   If the monitored Action is interrupted, it is treated just like other Actions in the remote AP. If the placeholder Action is removed, the remotely monitored Action is also removed.

3. **Mutual Action** The original AP waits for the start of its execution. The aPlan of the Action in the original AP just keeps the Resource reserved for the mutual Action. This is used when the resources need tight cooperation and the aPlan utilises both Resources.

   The mutual Actions can be chained and sent to many remote APs one by one, starting from the original Action. This facilitates the tight cooperation of multiple Resources. If the mutual Action is interrupted, it is treated just like other Actions in the remote AP, but the Action in the original will be restarted. If the Action in the original pool is removed, the remote Action will be removed too.

The Task is finished when all the Actions are executed or removed and the final state reached. Then all remaining ELs are terminated too. In some cases the Task is not *episodic* but continues (Table 2.1). For this purpose the Tasks can be defined

as *continuous* or *periodic*. Episodic Task execution is the default option. Continuous execution means that the Task will not remove the ELs and itself when all the Actions are executed. In the periodic case the Task is restarted from the beginning when all Actions are executed.

5.6.2 Graphical Representation of Task

The graphical representation (Figure 5.6) of a Task described here was developed mainly for illustrative purposes. It does not carry information about the variables that are passed. Its main purpose is to show the interactions between different Actions and ELs in the Task. It loosely follows the syntax of activity and sequence diagrams of the Unified Modelling Language (UML) [76].

The representation divides the Resources into separate lanes, so each AP has its own lane with its name at the top. ELs have their own lane, which is somewhat different from the others in order to indicate that it is not an AP. The graph flows down from the top but is not tied to any time axis. An Action is represented by a block divided into three. In the middle there is the name of the Action and the rounded blocks represent the EL interaction before and after the plan. ELs are rounded blocks in their own lane and with the name of the event written on them. Their reaction to the event is indicated with arrows coming from the sides of the block.

The Task is assigned to one AP. This is indicated with the typical sign of the initial state in flow charts, a solid circle. In the same lane there are walls. The walls divide the sets of Actions from each other; i.e. Actions for which the execution order does not matter are placed between the walls. The end of the task, the final state, is indicated by an open circle with a solid circle inside. A task can have special conditions for its final state. They are indicated with text under the final state symbol (Figure 7.4).

Naturally, the graph traverse starts from the initial state. Then the Actions in the following set are considered. There are four different kinds of arrows to indicate if something is going to be removed (dashed line) or added (solid line). When the following set of Actions is studied, the Actions from previous sets can be considered non-existent. But ELs that are not explicitly removed can be considered as still existing, even when the Action that added them is gone. If an EL adds an Action to the same pool with a task, it can be placed in a separate lane for the same AP (Figure 7.15). This is to indicate that the place of the Action between the sets is not defined.

The non-deterministic nature of the representation is further indicated in Figure 5.7 with a “fetch drink” task. The alternative Actions are indicated by placing them in parallel on the graph. So, in this case the drink to be fetched would only be one of the alternatives. For example, water could be available in the corridor, but the coffee and tea in the kitchen. Then the robot’s current location would decide the drink to be fetched. If the location of the coffee is very uncertain, that would extend the expected time needed to get to the coffee and again would alter the choice.

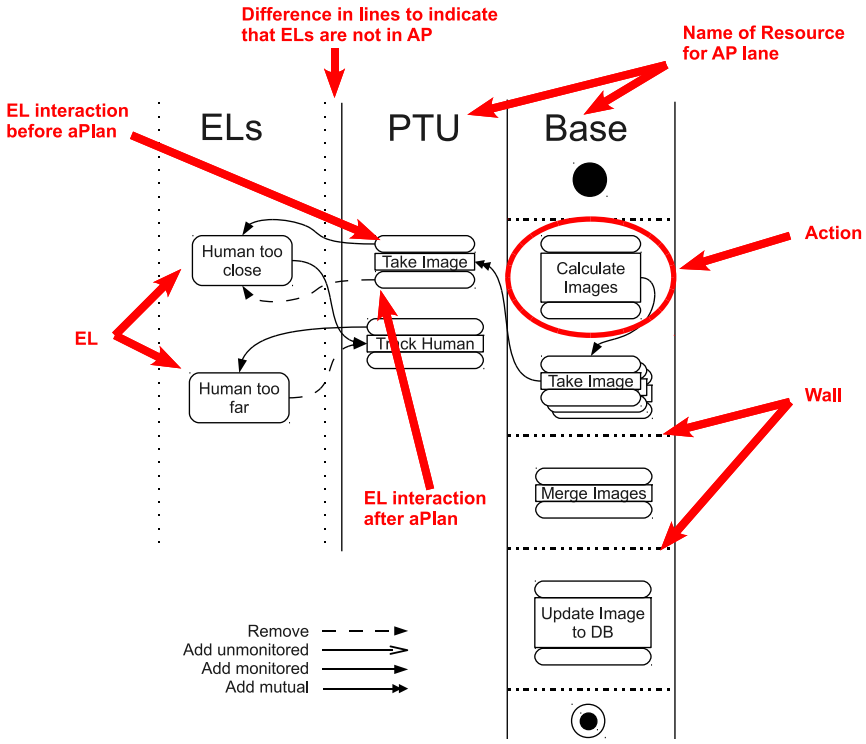


Figure 5.6: *Explanation of graphical representation*: Lanes represent different APs and ELs. Walls separate sets of Actions. The graph is read from the top down, starting from the round initial state symbol.

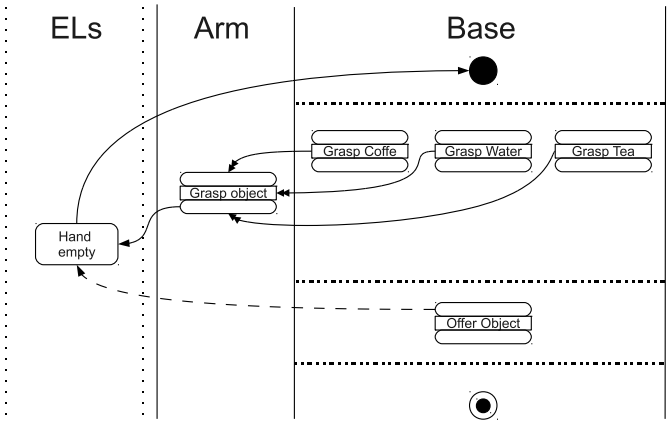


Figure 5.7: *Graphical representation example of "fetch drink" task*: This graph shows how to indicate alternative Actions by putting them in parallel on the graph. EL is the ActionPool solution for the hand monitoring dicussed in [75].

5.7 Control of Action Pool

There are some basic mechanisms and details in the control of Tasks in the Action-Pool method. In the following sub-sections these are described by going through some usage scenarios. Generally the AP can be queried as to its current state and a monitoring component can subscribe to the AP in order to receive status change messages. The AP is an independent software component and it is controlled remotely by issuing commands to add Tasks, pause the operation etc.

5.7.1 Adding a Task

The Tasks are added by the user or ELs. The Task is assigned or started from a specified AP. That AP instance then handles the Tasks. The first set of Actions in the Task is added to the pool, with their inherited priorities. Several instances of the same task can be added at the same time and each would have a unique specifier to reference them. The user can also add just an Action into the pool. Then a placeholder Task containing only that one Action is dynamically created. In this way the Action would show up in the listing of all the Tasks in the pool and possible ELs and remote Actions can refer back to it correctly.

5.7.2 Pausing

A Task can be paused by the user or by an internal error in the execution. When one task is paused, other tasks can continue their execution. Pausing a Task causes the abort of the Action that is currently being executed if it belongs to the paused Task. In the next Action selection the Actions of the paused Task are omitted. The ELs initiated by the Task are left running. After all, the world does not pause even if the Task does. If the ELs add some Actions to the pool, they are also omitted in the selection.

Aborting the Action can occur in one of three stages of the execution of the Action:

1. if it occurs during the resource reservation, the reservation process can easily just be halted;

2. the interaction of the ELs before the aPlan is a blocking operation, so the abortion would take effect only after that. It would have the same effect as abortion during the execution of the aPlan. The aPlan has an interruption procedure to be called and the EL interaction before the aPlan is reversed;

3. finally, the abortion could occur during the very short window after the aPlan. In that case, the Action is not aborted and the final EL interaction is conducted because the assumed effect of the Action should already be in place.

The operation of the whole AP can also be paused by an internal error or user intervention. If the whole robot pauses, all of its APs are paused. In effect, the

Action which is running is paused when AP is paused. Each aPlan has a pause procedure which is then called to drive the process to a safe pause state. The ELs are handled in a similar way to the Task case. If the Action that is currently running is removed, then the AP waits for the resume signal before continuing to the Action selection.

5.7.3 Removal of Task

A Task can be removed by the user or EL. If the Action that is currently running belongs to the removed Task it is naturally aborted and all Actions already in the AP are removed. All ELs that are initiated by the Actions of the removed Task are also removed.

5.7.4 Error Handling

Errors can be initiated by the aPlan or EL. Naturally, the aPlan provides errors from conflicts in the internal processes of the aPlan execution. Typical such errors are missing files and invalid arguments. Error conditions can also be initiated by causes external to the control process. Then the error is handled by the EL.

The consequences of an error regarding the control process depend on the severity of the error. A serious error from which there is no apparent recovery or which is impossible to repair causes abortion of the Action. If there is a chance for the Action to be recovered, for example by providing a missing parameter, the Action can be paused. All the internal errors have an error string related to them. This string describes the type and probable cause and source of the error. For external errors the repairing Actions also have priority that would dictate the effect, depending on the situation. The repairing Actions are added by the EL that notices the error situation, just like in any other condition monitored by ELs.

5.8 Summary

ActionPool is a dynamic task scheduling method constructed from several components. A summary of the main concepts is provided in Table 5.2. These are more ActionPool specific descriptions than the ones in the Glossary. Figure 5.2 also summarises the basic principles of ActionPool.

Table 5.2: Summary of essential ActionPool terms

Term	Description
Task	Element with metadata and a plan to do the Task constructed from Actions
tPlan	A plan to do the Task constructed from Actions
Action	Elementary operation on abstracted Resource, contains additions and removal of ELs and a plan for how to use the resource (aPlan)
aPlan	A plan for usage of the Resource in order to conduct an Action
EL	Monitoring software agent running simultaneously with tPlan and aPlan execution, Can be used to protect a state or confirm some pre-condition
AP	Component to schedule and manage Actions on a Resource, Actions can be from different Tasks
Resource	Physical and logical sub-system of the robot to be controlled
Perception Agent	An agent to store observation of objects from refined sensor readings to DB
DB	Centralised storage for robot's understanding of the current situation in its environment (world model)

Chapter 6

Implementation

Two fundamentally different mobile robots were used to show the feasibility of the ActionPool method, the omni-directional MARY [10, 112] and the differential-drive Rolloottori [11]. MARY was controlled through a PC connecting directly to the actuator with a data acquisition card, while Rolloottori connects to the hardware via microcontrollers communicating with the PC via serial links. So the real-time control of MARY was calculated in the PC and that of Rolloottori on a micro-controller. An overview of the main differences is shown in Table 6.1.

Both robots were originally built for other purposes than these experiments, so they both required some development work and maintenance before they could be used for the experiments. That mainly involved the installation of some new sensors and computing power for autonomous operations.

Table 6.1: Comparison of main differences between robots used

MARY		**Rolloottori**
$\approx 150kg$	mass	$\approx 20kg$
$\approx 1.4m$	height	$\approx 0.75m$
$\approx 0.6m$	width	$\approx 0.42m$
omni-directional	mobility	differential
3	# on-board computers	1
on PC	real time control	on micro controller
Player [9]	middleware	GIMnet / MaCI [23]

6.1 Hardware

6.1.1 MARY

The base of MARY was originally an electrical wheelchair mechanism. The mechanism is based on four mechanum wheels that are controlled independently. Thus the base has four degrees of freedom (DOF) to control the three DOF. This permits omni-directional manoeuvres and the base is actually over-actuated. The manipulator arm mounted on top of the robot has seven DOF. The Mitsubishi-made arm (PA-10) weighs around 20 kg and has a workload of around 10 kg.

MARY is equipped with a Panasonic pan-tilt-zoom camera unit (PTU) and a Hokuyo (URG-04LX) infra-red (IR) laser scanner. The base is controlled with an industrial PC equipped with an IO card in a PCI bus to directly interface with the encoders and motor drivers. Figure 6.1 shows the schematic hardware layout of the main components of MARY. PC2 and PC3 are 2004 model laptop computers with 1.6-GHz processors. Figure 6.2 shows the placement of the components.

The PC1 was running on the QNX operating system and took care of the real-time control of the base and arm. Unfortunately, the researcher's limited exchange time in Japan did not allow the utilisation of the arm. PC3 gathered the data and connected to the real-time control program on PC1. PC2 and PC3 were running Linux operating systems. PC2 then had the mission-level computations and interfaced with PC3 to gain some access on to the robot peripherals.

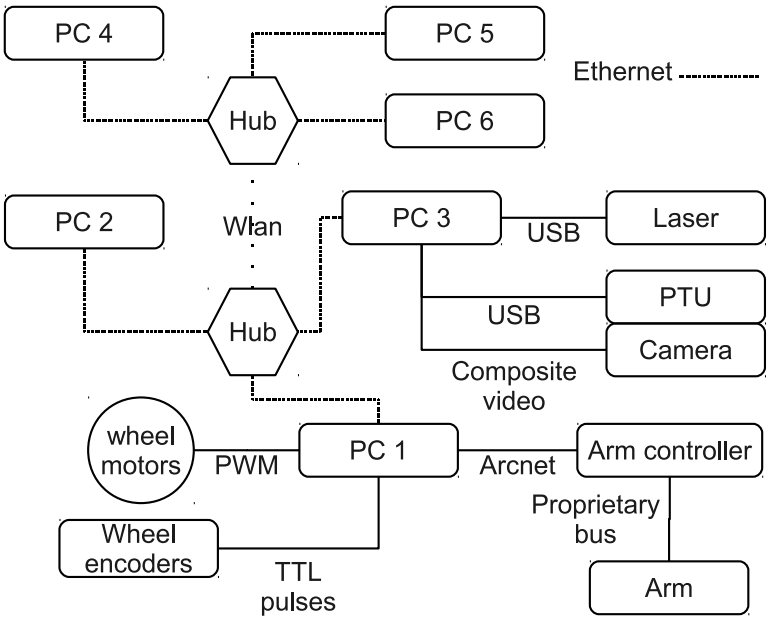


Figure 6.1: *Layout of the hardware structure in MARY*

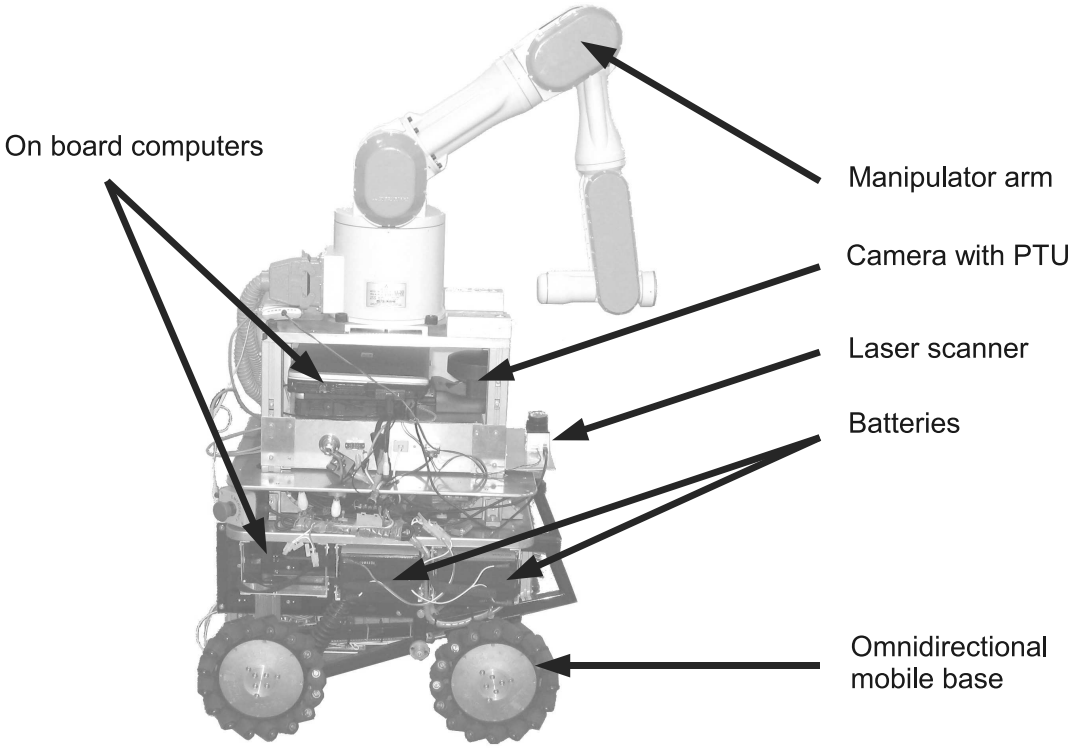


Figure 6.2: *Overview of MARY robot*

6.1.2 Rolloottori

Rolloottori's electrical hardware is originally from a ball-shaped robot with three DOF and is controlled by a 16-bit microcontroller only. One of the DOF was used to tilt a camera. The two remaining DOF were used to control the direction and speed of the ball-shaped robot but have now been adopted to control the left- and right-hand wheels on the robot. Commands to the robot were given through a wireless serial link and the camera image was wirelessly sent directly to an off-board computer for processing.

To be used in the experiments, the robot needed some upgrading. It got an independent three-DOF (pan-tilt-zoom) camera unit (Sony SNC-RZ30) and an onboard computer (an Acer One ZG5 with an Atom processor). Other environmental sensors were an infrared (IR) proximity sensor-based bumper and an IR laser scanner manufactured by Hokuyo (URG-04LX). An overview of Rolloottori and the placement of its components is presented in Figure 6.3.

The most important hardware components are organised as presented in Figure 6.4. The IR bumper is connected to the digital input of the microcontroller. The microcontroller also counted the wheel encoder pulses and produced a PWM signal to the wheel motor drivers. The camera and its pan-tilt-zoom unit (PTU) are connected via Ethernet and a hub. The hub is also used to connect the system to a wired network for maintenance and to connect an occasional extra computer onboard.

The microcontroller and the IO card are connected to the PC1 on board via a USB-RS232 adapter. The laser scanner was connected to PC1 via USB. PC1 has only USB ports built in. PC1 and PC2 were connected via a WLAN connection. PC2 and the rest of the PCs were connected via a regular office Ethernet network.

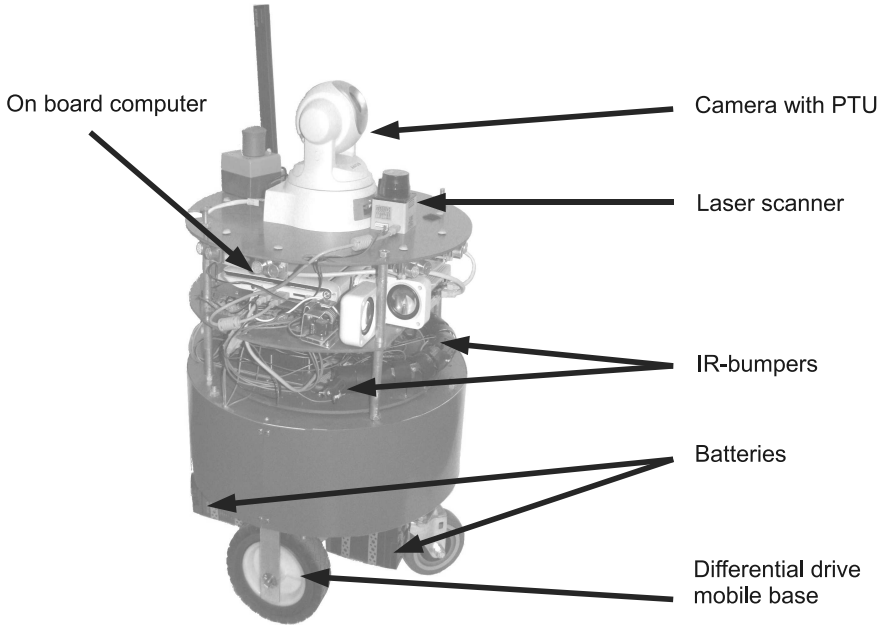


Figure 6.3: *Overview of Rolloottori robot*

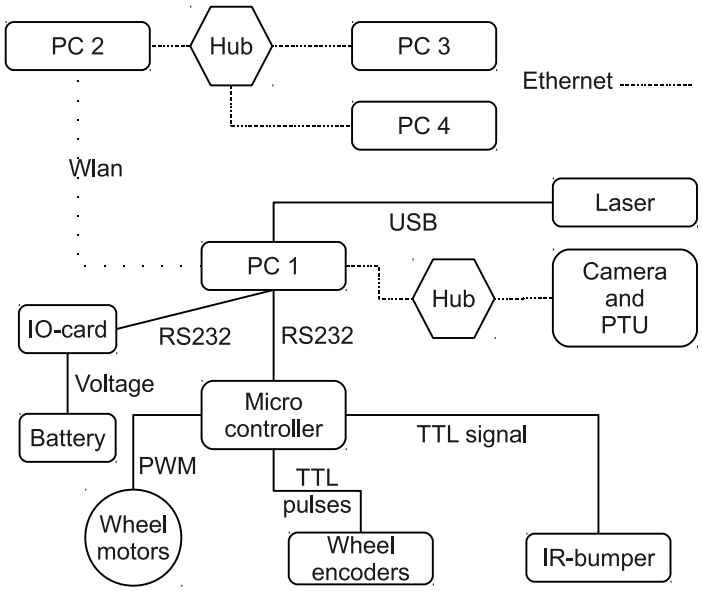


Figure 6.4: *Layout of hardware structure in Rolloottori*

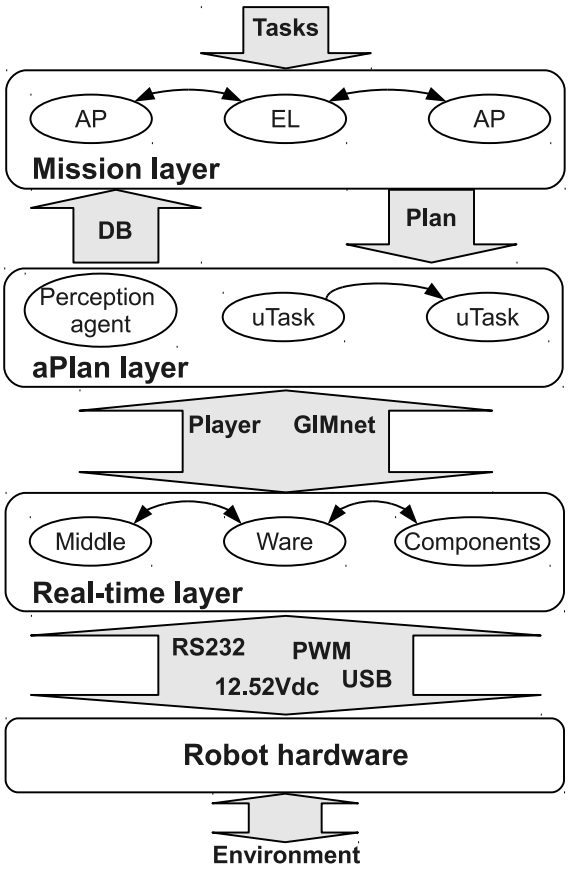


Figure 6.5: *Layers of the software in the implementation of ActionPool method*

6.2 Software

As mentioned in Chapter 2, the complex control problem of the service robot is divided into layers and components. Three layers may be distinguished: 1) a Real-time layer; 2) an aPlan layer, and 3) a Mission layer (Figure 6.5). The implementation of the Mission and aPlan layers is constructed with the Python programming language and an object-oriented programming scheme. The Real-time layer is mainly programmed in C++ or C, with some elements in Python.

The idea of isolating the software into layers is not only to have smaller subproblems but also to have interchangeable solutions for the subproblem, as, for example, was the case with the Real-time layer using different hardware abstraction techniques for different robots. The hierarchical layers isolate different time scales in the control.

6.2.1 Mission layer

The mission layer is formed from Tasks, Event Listeners, and Actions in Action pools. Using the terms from [19], the aPlan and Mission layer form the sequencing layer. The deliberative layer is not implemented because it would require more advanced situation awareness than is possible with today's technology in the service robot scenario. The control layer mentioned in [19] is naturally the real-time layer.

Task

The tPlan is constructed from sets of Actions separated by "walls". When the Task is executed, one set of Actions is added to the AP for execution at the same time. When all the Task's Actions in the pool are executed or removed, the next set of Actions is added. The Task is encoded using XML (Extensible Markup Language). The general XML syntax is described here and the definitions regarding the Task shortly afterwards.

XML is a systematic and formal way of representing information. It is machine-readable, i.e. a computer system can parse the information with relative ease. It is comprehensible to humans, too. The information is organised with so-called tags and their attributes. In order to be understood, the meaning of the tags and their attributes has to be defined. A hierarchical structure can be formed by placing tags as information for a higher-lever tag.

For example, we could define an object, a blue ball, with XML. The information start with a start tag describing what kind of information we are dealing with. The tag name is placed between "smaller than" and "greater than" signs.

```
<object>
```

With attributes, some common features for all information of a particular type can be expressed.

```
<object colour="blue">
```

Finally, the information can be placed between the start and stop tags.

```
<object colour="blue"> ball </object>
```

A stop tag is denoted by a slash in front of the tag name. If just the attributes are sufficient, the slash can be placed at the end of the start tag. Thus the end tag is not needed

```
<object colour="blue" />
```

In the listing below there is an example of a task in XML code as it is sent to the AP. Table 6.2 explains the tags used and their attributes.

```
<Task name="sampleTask" id="12345" priority="0.123"
permanent="0" replay="0" description="does consecutive sleeps">

  <Target id="1" name="sleeper" description="pseudo db id ">
  <Action filename="Sleep.xml" order="0" wall="0" last="0">
    <Parameter type="Int" value="5" name="var_sleepTime"
    description="time to sleep"/>
  </Action>
  <Action filename="" wall="1" order="1" last="0"/>
  <Action filename="Sleep.xml" order="2" wall="0" last="0"/>
  <Action filename="" wall="1" order="3" last="0"/>
  <Action filename="" wall="0" order="4" last="1"/>
</Task>
```

Actually, this XML code is in a file on the same computer as the AP program. Even though the AP could receive and act on the Task XML code sent over the network, just the file name is sent instead for convenience. The directory where the Task's XML file resides can be found on the basis of the common configuration file. The configuration file also describes the structure of the system and the addresses of other components. With that information the AP knows how to connect to the robot, where to send the ELs, and what the other APs are. The same configuration file is used by all the APs and ELs of one robot.

Action and EL

The listing in Appendix B describes an example XML code representation of an Action. The code also includes the Action plan, which is covered in the next section. The Action can be found written in a file in a directory described by the configuration file. The file name is found from the tPlan representation. Table 6.3 describes the tags that are used and their attributes inside an Action and its EL interactions.

Notice that some of the ID attributes in the code can be undefined. This XML code is loaded by the AP when the Action is added. The AP assigns an ID number for the Action. The Task also has a unique ID number within the AP and provided by it. The Action then inherits the parent task's task ID. In this way all the Actions belonging to a particular Task can be identified, for example when the Task is removed. The undefined attributes in the XML are filled by the AP accordingly and the same file format is used to save and resume the state of the AP on the hard disk.

Table 6.2: Definition of XML tags and attributes for Task

Tag	Attribute	Description
Task		Defines a Task
	name	Same as the file name, used mainly in user interface to tell what tasks are running
	id	Place holder for an id assigned for the task by the AP, can be used to save a state of the AP
	priority	Priority of the task that the Actions will inherit
	permanent	Tells if Task and related ELs should be removed after all Actions of the Task are executed
	replay	If task should be started over from the beginning when all Actions are completed
	description	Free-form description of what the task does
Target		Target object for the task; it is always a reference to DB
	id	Identification number of the object in DB
	name	Optional name of the variable
	description	Free-form description of usage of this object in the task (or other relevant information)
Action		Defines an Action in the Task
	filename	Complete name of the XML text file describing the Action
	wall	Indicates a virtual wall separating patches of Actions
	last	Indicates that the task ends here
	order	Indicates the order in which Actions are added into Action pool starting from 0. It is preferable that order numbers are not skipped. If there are actions with the same order number it means that they are "or" conditioned between each other
Parameter		describes a task-level parameter to be used in the Action, overrides the value specified in the Action's file
	type	Type of the parameter (Double, Integer, String, or Unknown)
	value	Value of the parameter
	name	Name of the parameter
	description	Human-comprehensible explanation about what the parameter or return value contains

Table 6.3: Definition of XML tags and attributes for Action and EL

Tag	Attribute	Description
Action		Describes an Action
	name	Name of the Action
	objectID	Reference to the database for resource reservation
	id	Unique identification number inside Action pool
	taskID	Unique identification number of Action's parent Task
	priority	Priority of the Action
	location	Location to start the aPlan if objectID-attribute is not provided
	sigma	Variances for the location coordinates to describe how accurately it should be achieved before starting the aPlan
	lambda	Update weighting for the execution time
	time	Expected execution time and its variance
	description	Human comprehensible explanation about what the Action does
	version	Version control
ELstart		What EL interaction is done before the plan and after the resource reservation
ELfinal		What EL interaction is done after the plan
ADD		An EL is added or initiated
RM		An EL is removed
	taskID	ID number of the task that this EL belongs to
	actionID	ID number of the Action that this EL belongs to
	EL_ID	ID number of the EL
	filename	Filename of the XML file for EL if the XML is not provided
ELheader		Describes an EL
	threshold	Threshold for triggering the EL
	objectID	Reference to DB if Events related to specific object are listened
	exeFilename	Filename of the executable code
	description	Human-comprehensible explanation about what the EL does
RESPONSE		Describes one response for triggering the EL
	command	What to do as a response
	filename	If something is added, its description
	pool	Defines which AP the command applies to
	actionID	If Action is removed, its ID number

Action Pool

For each AP and Resource pair there is an *Action pool manager* program that keeps track of Tasks and manages the Action selection and Action execution process, and through which communication to other components of the system goes. The diagrams of the classes related to the AP are described in Appendix D; i.e., the AP can be seen as a data structure containing Actions. The manager program contains the Tasks and serves as a point of contact for the user to command the robot.

World Model

The robot's current understanding of the world is modelled into a database. The database (DB) is based on objects with some basic properties listed in the appendices in Table A.2. A noticeable feature of the properties is that the uncertainty of the numerical properties is modelled with the first and second moments (μ, σ^2) of the variable.

The DB is implemented utilising the mySQL database engine. The ActionPool architecture is constructed from several distributed software components and this kind of DB is known to be not the fastest option but it provides a simple and accessible interface for the data exchange. From the architecture point of view, the implementation of the DB is irrelevant as long as some queries can be made and data can be updated and retrieved.

To make the usage of the DB easier, a wrapper class for it was created. The methods of the class allowed simple interaction with common chores, such as search queries based on the distance from a specific point in the world model or a command for uploading the texture of an object from an image file.

6.2.2 aPlan layer

The aPlan layer utilises a so-called micro task (uTask) concept [7, 12]. uTasks are functions utilising the HAL. They are atomic in the sense that you cannot divide them without breaking their functionality. Still, they can be executed, paused, continued, and aborted. The output would be success or error, with a string defining the source of the error and a code defining the severity of the error. Several uTasks can be organised into functions called *workTasks*. The aPlan is then constructed from the workTasks and uTasks with a parameter-passing mechanism and flow control components such as "if" statements. The aPlan is also serialised into an XML structure.

The aPlan layer communicates with the Real-time layer via uTasks. uTasks have connections to the actuators and sensors via middleware. The aPlan layer just gets aPlans from the Mission layer and sends some status messages back. A breakdown of the Task implementation can be seen in Figure 6.6. The aPlan and uTasks are parsed and executed in this layer.

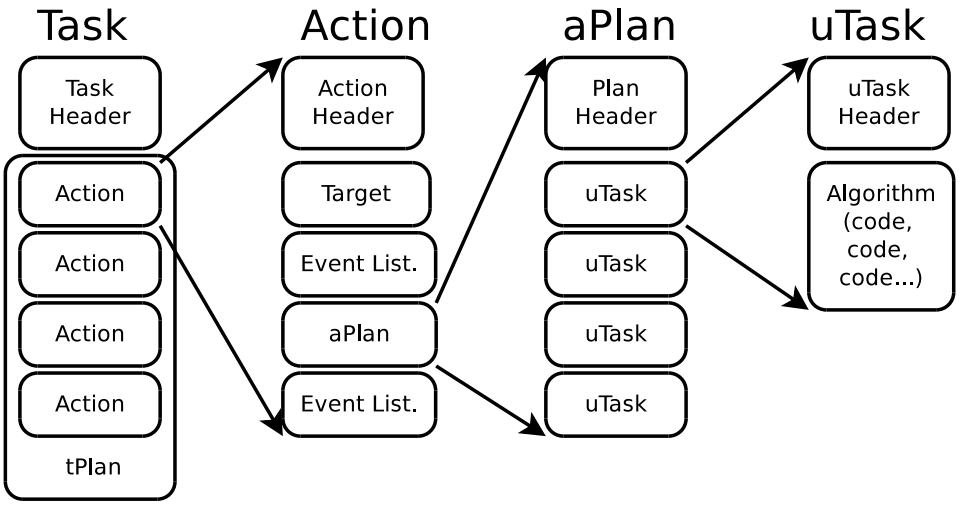


Figure 6.6: *Breakdown of Task into uTasks*

uTask and aPlan

The aPlan is also defined in the XML format and an example can be found inside the XML code example of the Action in Appendix B. It includes the XML representing the uTasks and their connections constituting the aPlan structure. The XML format supports a graphical representation of the plan as a flow chart. The representation closely follows the one presented in [12] and thus there are some elements that are not used in this context. The Table 6.4 defines the tags and attributes that are used.

Perception Agent

Perception agents are independent programs and consequently they can easily be distributed in the system. The basic structure of the agent is fairly simple, except, naturally, the algorithm refining the data. Some very simple ones were created too, just to forward, for example, the readily available pose data to the DB.

6.2.3 Real-time layer

In this implementation the so-called microtask paradigm was selected. It is a procedural representation used in [7, 12] and loosely based on [13]. The microtask (uTask) is an atomic operation of the robot on a hardware abstraction level.

The Real-time layer, as the name indicates, handles algorithms that need to have a tight and deterministic control loop execution. They are constructed from different middleware components, as described in Figure 6.7 and 6.8. They are described in more detail in the middleware section below. Basically, the role of the Real-time layer is to provide a hardware abstraction layer (HAL). It contains algorithms to control the different Resources safely.

Table 6.4: Description of XML tags and attributes for aPlan and uTask

Tag	Attribute	Description
Plan		Describes an aPlan
SingleTask		Describes a uTask
	name	Name of the uTask, the name of the executable file is derived from this
	PS	Defines which resource this uTask is mainly using and in which library it could be found
	id	Identifies the uTask in the aPlan structure
	priority	Not used in this context
	description	Human-comprehensible explanation about what the uTask does
Parameter		Parameter for the aPlan or uTask
ReturnValue		Return value from the aPlan or uTask
	type	Type of the parameter or return value (Double, Integer, String, or Unknown)
	value	Value of the parameter or return value
	name	Name of the parameter or return value
	description	Human-comprehensible explanation about what the parameter or return value contains
Block		Defines the graphical representation of the element
	uiBlockType	Graphical appearance of the element
	uiPosX	X-coordinate of the element on canvas
	uiPosY	Y-coordinate of the element on canvas
	InitText	Not used in this context
	ExeText	Not used in this context
	EndText	Not used in this context
	ErrorText	Not used in this context
Connections		Defines the execution order and graphical wiring of the uTasks in the aPlan
Connect		Connection between two uTasks
	From	From which uTask the connection starts
	FromPos	From which part of the graphical representation the connection starts
	To	To which uTask the connection stops
	ToPos	To which part of the graphical representation the connection stops

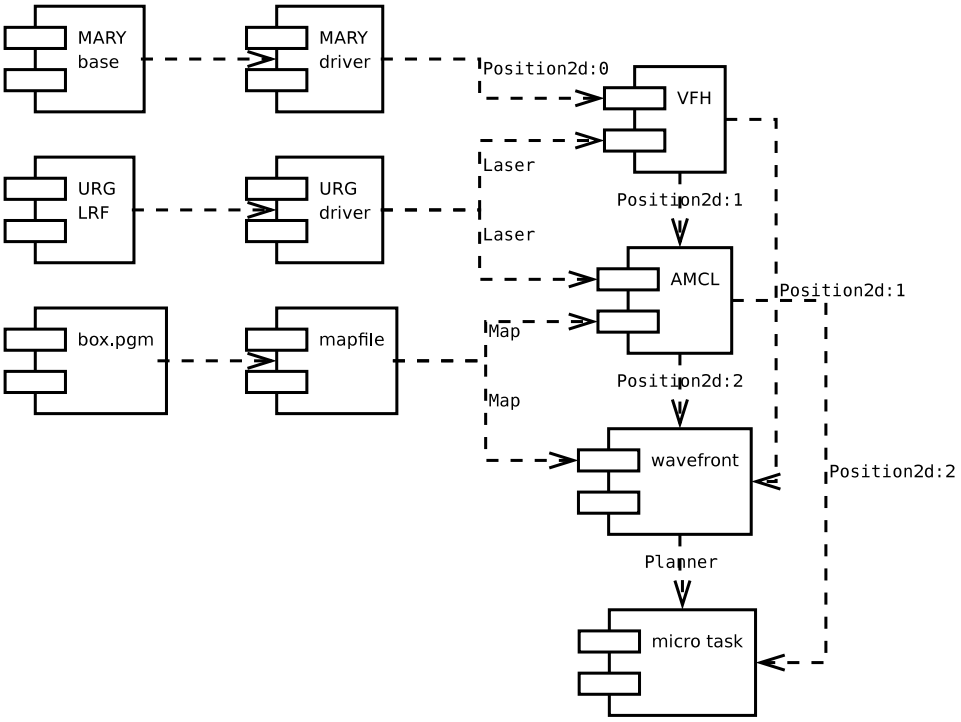


Figure 6.7: *Deployment of Player software components on MARY to control the pose*

Middleware

Two different middlewares were used, one for each robot. The middleware was used to abstract the hardware and provide a method for the software components to communicate with each other. The robotic middleware typically includes drivers for the robotic peripherals, such as the camera or the joints of the robot arm. Most importantly, the robotic middleware defines common data types for these peripherals and a way for them to communicate conveniently across a computer network. In this way, for example, the camera can be changed and a program requiring the image feed from the camera does not have to change. The only thing that it would be necessary to change would be the driver for the camera. Another important feature of the middleware is that the computing can be divided into software components. Software components are independent processes that can be distributed over several CPUs and which communicate during the run time.

As mentioned in 4, with robotic middleware the system definition is close to functional programming or so-called data-driven programming. The system would be built from modules that accept certain kinds of data and/or provide some type of data. The modularisation makes the reuse of the components possible, or alternative components could be used for testing and tuning the performance. Work done by many programmers can be unified conveniently. The possibility of utilising ready-made basic algorithms for localisation and getting some other software support was the major motivator for the use of the middleware in this work.

The Player middleware components to control MARY's pose are shown in Fig-

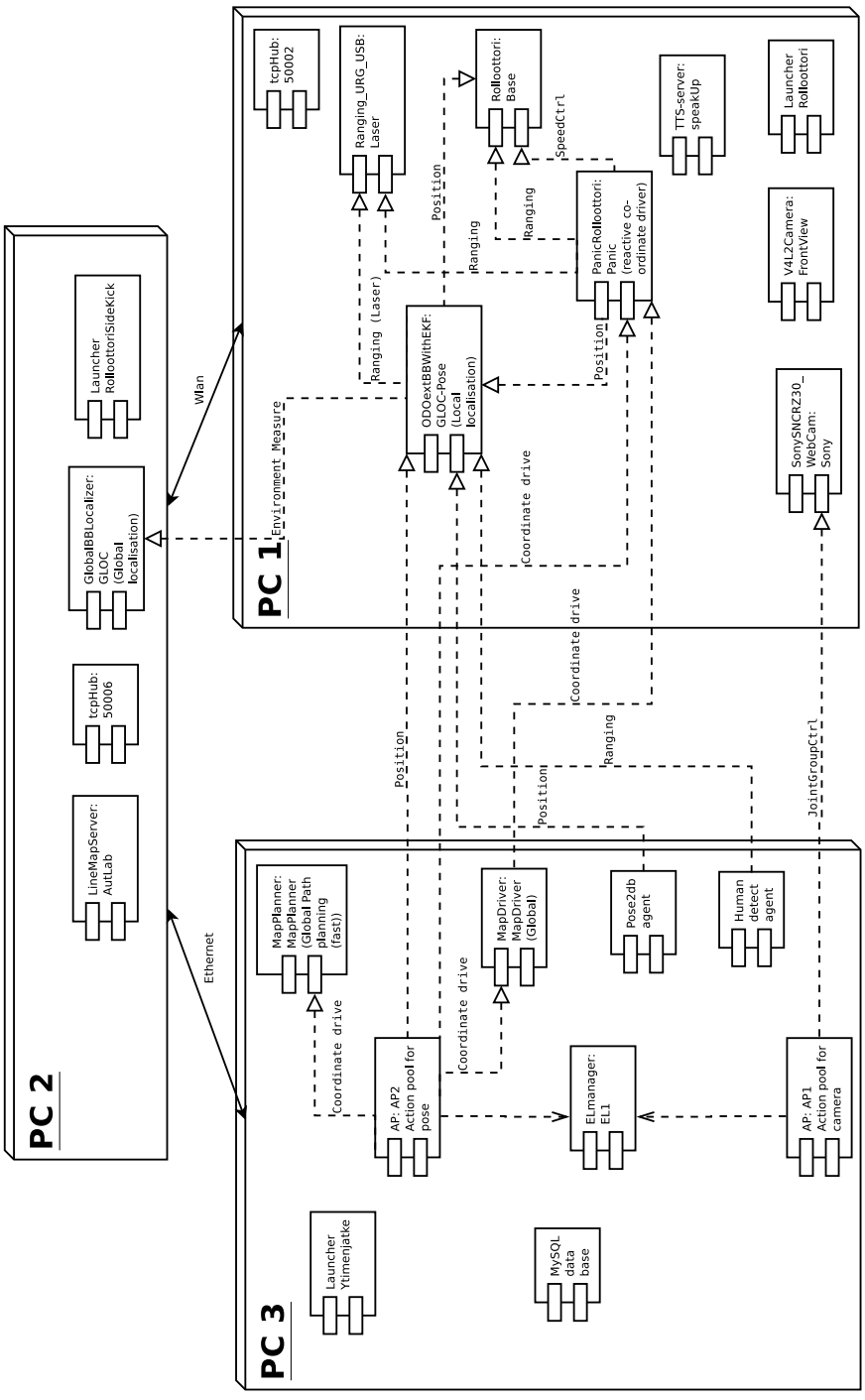


Figure 6.8: *Deployment diagram of MaCI software components on Rolloottori to form the control system*: PC1 has the components for the Real-time layer. PC3 has the rest of the components to form the ActionPool architecture. PC2 acts merely as a connection point.

ure 6.7. The map is read from a file and provided through a ”map“ interface. The odometric data gathered by the computer driving the wheel motors were provided through the ”position2d“ interface. The laser scanner data were acquired via the ”Laser“ interface. Then the data are refined by other computers to provide a ”planner“ interface to command the robot intoa certain pose. The ”planner“ and ”position2d“ interfaces were then utilised by the microtask to control the robot.

A more comprehensive picture of the software components used in the case of Rolloottori is provided in Figure 6.8. In this case, the components on PC1 constitute the Real-time layer. The aPlan and Mission layers are both embedded into the AP components in PC3. Not all the connections are shown; every component is connected to one of the Launcher and tcpHub components. The fourth PC, which runs the GUIs, is not shown either.

6.2.4 Supporting Components

Additionally to the software directly related to the implementation of the ActionPool components, a certain amount of supporting software has to be implemented. The most essential elements are briefly discussed below.

Inter-Process Communication

The software components developed for the ActionPool method, APs, ELs, and user interfaces, required some form of communication channel with each other. This is known as the inter-process communication (IPC) problem. Because the data sent between the components were not data to or from a typical robot peripheral, the middleware could not be utilised for that purpose. The author developed a simple TCP/IP-based IPC protocol and a Python class to utilise it. That was used in the case of MARY. With Rolloottori, the GIMnet [23] was used as middleware and the IPC protocol was changed to take advantage of the communication stack provided by the GIMnet. The functioning of the Python class stayed identical for the applications utilising it.

Replacement of Player Components

The Player provided a complete set of tools for the navigation of MARY. In the case of Rolloottori, some of the functions required their own implementations.

The dynamic obstacle avoidance was performed by a module named *PanicRolloottori*. PanicRolloottori follows the reactive control scheme. The module builds up an instantaneous local occupancy grid map around the robot in polar coordinates on the basis of the readings from the ranging devices. It also creates a rotational and linear velocity towards the target position from the robot's current pose. At the target position it rotates the robot to the target orientation. The program is based on several states that are selected according to the local map and the recent history of the states. The states scale the velocities towards the target pose or override them completely. The states are based on the following guidelines.

Algorithm 1 Obstacle avoidance

1. If something gets near, slow down the linear velocity.
2. If something gets close, steer away and slow down.
3. If something gets very close, stop and turn until the way ahead is clear and then go ahead a little.
4. If something gets very, very close, stop, go back a little, and turn more than 45 degrees.

With these reactive controls the robot can navigate safely but could get caught into a local minima. To prevent this, a so-called panic reaction was added. It uses a so-called ”stress level“ that is raised if the robot has to perform obstacle avoidance manoeuvres and lowered when there is free space around the robot. If the robot has to manoeuvre around obstacles for too long, the panic state reacts and causes one of the three panic reactions.

- Go ahead for five seconds.
- Turn a lot and go ahead for three seconds.
- Wait for ten seconds.

Since the robot is relatively small and sturdy, these panic reactions work quite well. It can push some small objects, like chairs or doors, out of the way and react to people obstructing its way. Naturally, these panic reactions would not be feasible in some other environment or with some other robot. The state of the ”stress“ level for panic is intialised.

The Action selection process in the AP evaluates the cost of reserving the Resource for each Action in the pool. The process can occur relatively often, so The lines in the roadmaps were formed from connected nodes for path planning purposes the process of calculating the cost should be fast. When the Resource is the pose of the robot, the reservation cost is directly related to the length of the path. In this way the cost calculation is reduced to path planning between the robot's current location and the locations of the Actions in the pool. The wave front algorithm is a good path planning algorithm but it takes too long in a slightly more complex environment than that used with MARY. Thus a faster roadmap-based path planning was implemented. The cost of the speed is the reason for creating the roadmap before hand. The roadmap used and outline of the walls of the experimental space as a vector line map are illustrated in Figure 6.9. The algorithm descibed below was used with the roadmap.

Algorithm 2 Path planning based on roadmap

1. See if there is already a free path for the direct line between the start and end points.
2. Find the closest node in the roadmap with a free path from the start point.
3. Find the closest node in the roadmap with a free path to the end point.
4. Find the shortest path within connected roadmap nodes between the previously found nodes. The famous Dijkstra algorithm is used for that.

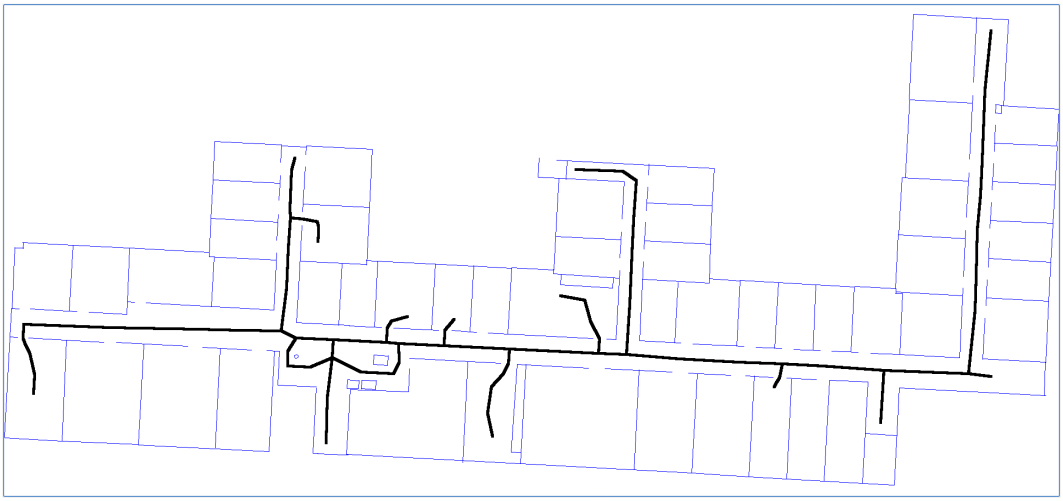


Figure 6.9: *Line vector map of walls with overlay of road map*: The lines in the road map were formed from connected nodes for path planning purposes

User Interface

To use and test the method that was developed, some way to access the AP and EL modules running in the system was required. For this purpose three GUI programs were developed, one for APs and Actions in them (Figure 6.10), one for Tasks in APs (Figure 6.11), and one for ELs (Figure 6.12). The programs are quite rudimentary engineering interfaces, just to gain some access to the system's internal operations, apart from the endless log files. The experiments described in the next chapter were conducted by commanding the robot through these interfaces. Of course, today's technology allows much more intuitive and elegant ways to interact with the robot, but the user interface side was demarcated out from this work. Furthermore, a more complex user interface would also have made the evaluation of the proposed method more complex.

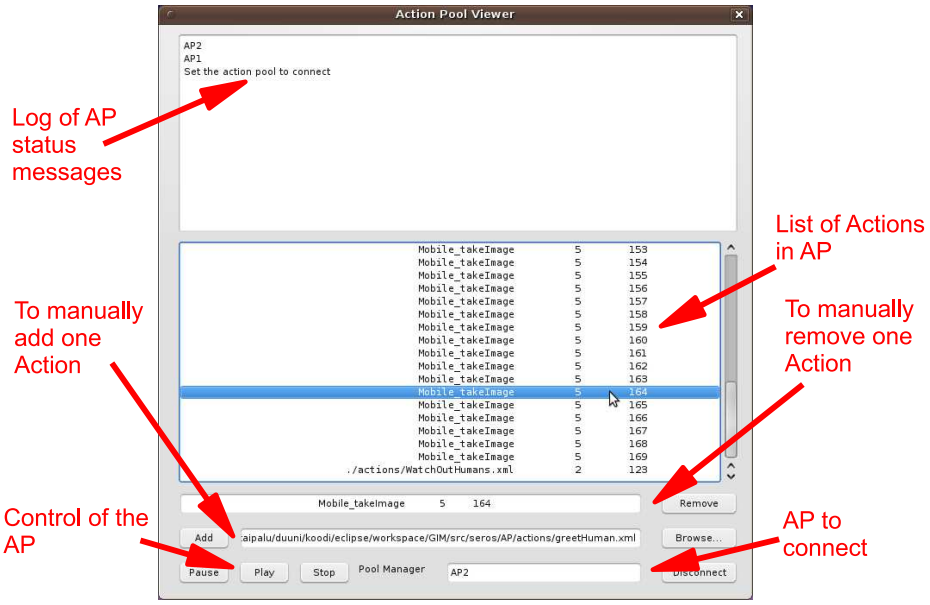


Figure 6.10: *Screen shot of graphical user interface for management of Action pool and Actions in it*

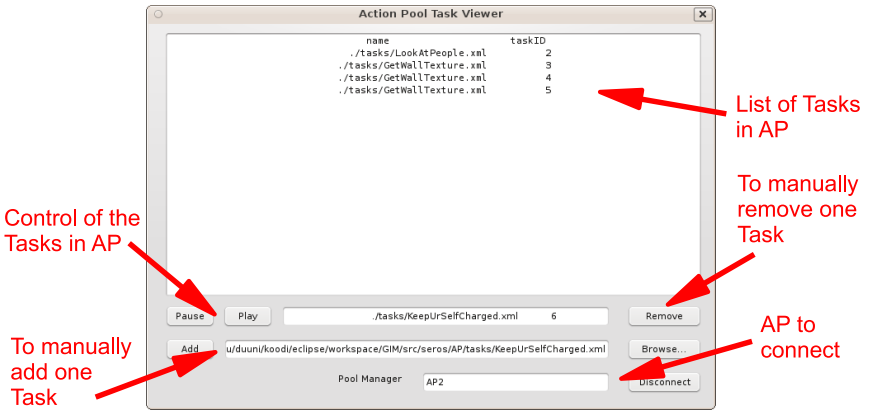


Figure 6.11: *Screen shot of graphical user interface for management of Tasks in Action pool*

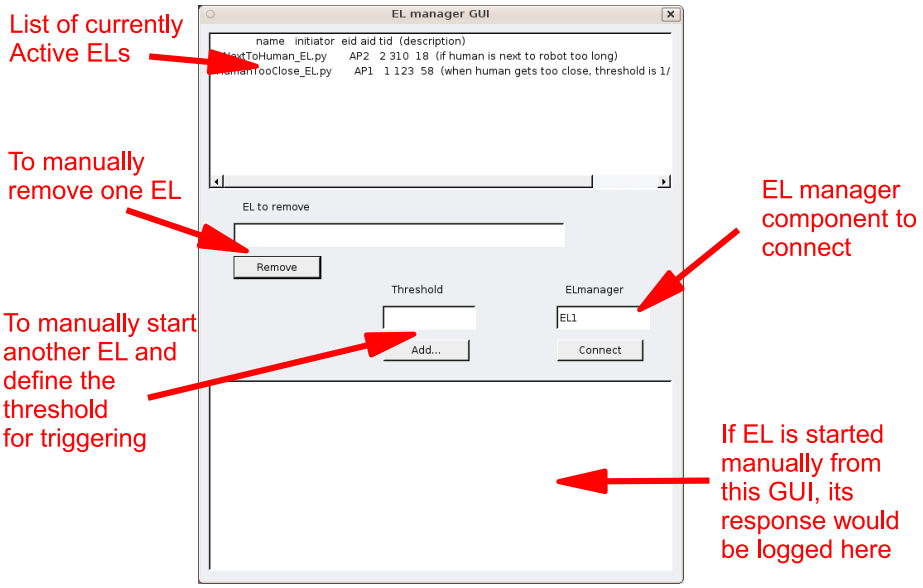


Figure 6.12: *Screen shot of graphical user interface for management of Event Listeners*

Chapter 7

Verification Through Experiments

Several experiments were conducted to verify the soundness of the ActionPool method and task knowledge representation, as well as to prove the applicability of the ActionPool method. The research problem was described as service robot control in an environment shared with humans. The environment shared with humans, i.e. our world, is way too complex to model reliably by simulation. Additionally, the inherent problems caused by the robot's internal structure, such as communication delays, fluctuating sensor data, misalignments in the robot's construction, nonlinear and fluctuating actuator responses, etc., are very hard to model, and those problems are exactly what the ActionPool method should be tested against. This is why it was seen as necessary to conduct the experiments with real robots instead of simulations.

In the following sections a set of experiments (Table 7.1) are described. The selection of the experiments was made so that requirements specified in Section 2.7 could be tested. In the next chapter, Chapter 8, the results of these experiments are presented from the point of view of system performance analysis. The experiments were repeated and the experimental setup refined until each one of the experiments could be performed without interruptions caused by its inadequate implementation.

Table 7.1: Experiments conducted with two robot platforms

MARY	**Rolloottori**
• pose reservation	• pose reservation
• find object and take picture of human	• texture mapping a wall segment
	• texture mapping a wall segment with exception
	• texture mapping of wall segments

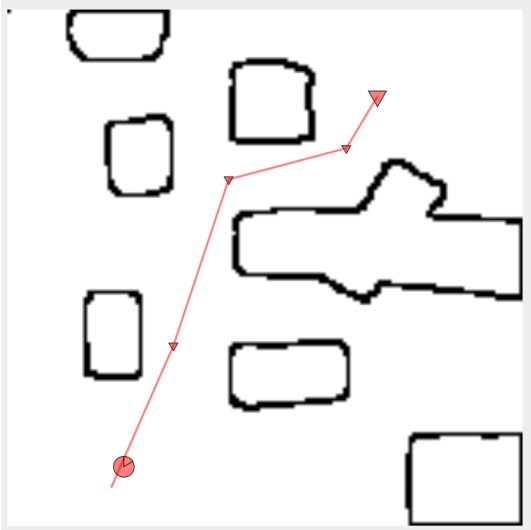


Figure 7.1: *A path created with wave front path planning algorithm implementation included in Player*

7.1 MARY

The first experiments were done with a mobile manipulator robot called MARY [10]. The experiments were done in the System Robotics Laboratory at Tohoku University, Sendai, Japan.

7.1.1 Pose reservation

MARY's base is used as a Resource in the pose reservation experiment. The pose reservation equate context switch of the base Resource. In this experiment the localisation and navigation are done utilising Monte Carlo localisation [38], wave front path planning [41] (Figure 7.1), and the vector field histogram (VFH) obstacle avoidance [41] of the Player [9] package. The algorithms that were provided required refining for the measurement range of the laser scanner that was used. The experiment was done in two stages: simulation and a real-world experiment. The simulation was done with complex and simple environment models (Figure 7.2). Naturally, a model of the robot had to be created for the simulation. The simulation was done utilising the Stage simulation engine with the Player package. The real-world experiment was done in a simple and safe environment (Figure 7.3). The obstacle avoidance was tested with a human occluding the planned path.

The experiments went well after the parameter tunning of the VFH algorithm for the real robot, because the algorithm assumed the location of the laser scanner to be in the middle of the robot when in fact it was just at the front edge of the robot. The dynamic obstacle avoidance worked well too. All in all, the base Resource of MARY was successfully abstracted for Action pool (AP) use.

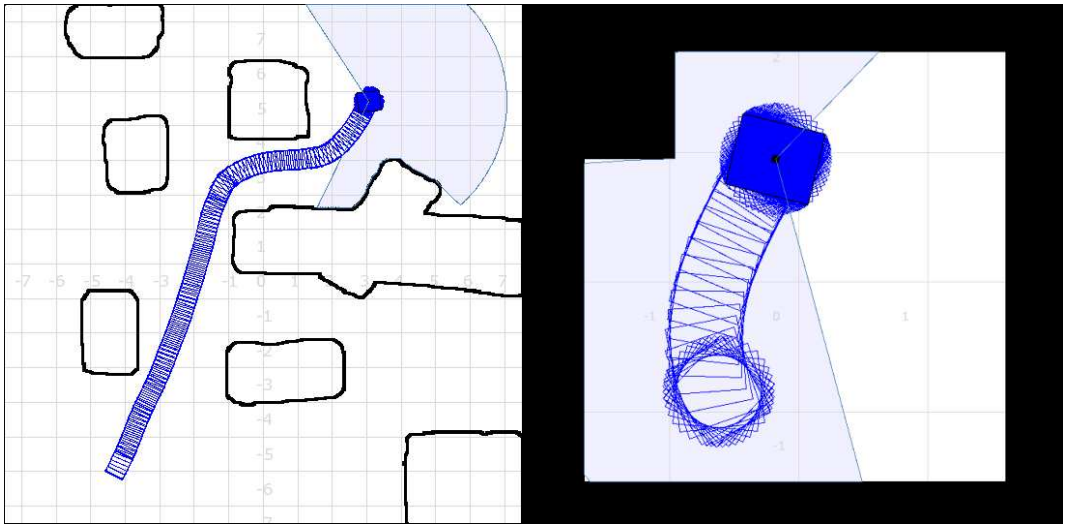


Figure 7.2: *Experiments with MARY in simulated environment and sensor response*

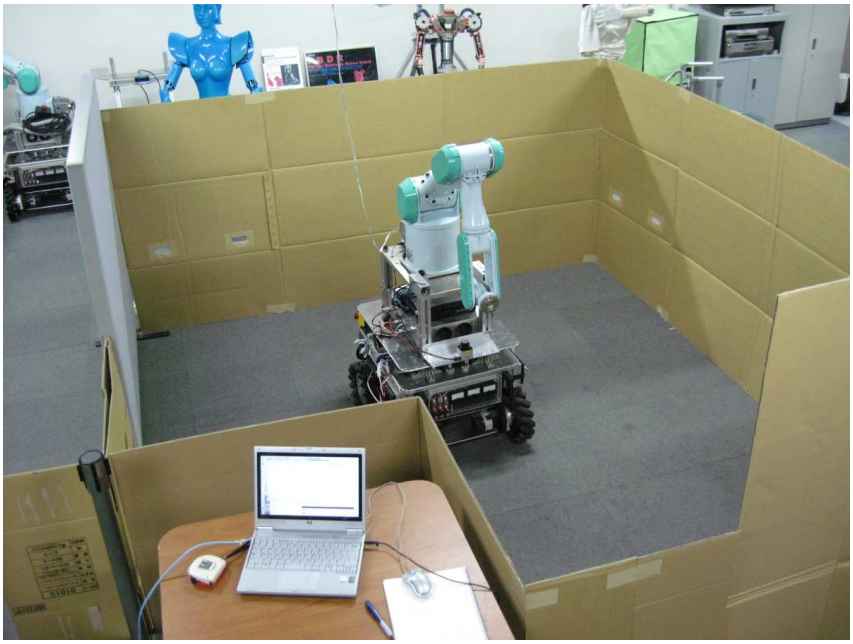


Figure 7.3: *Experimental setup for MARY in pose reservation experiment*

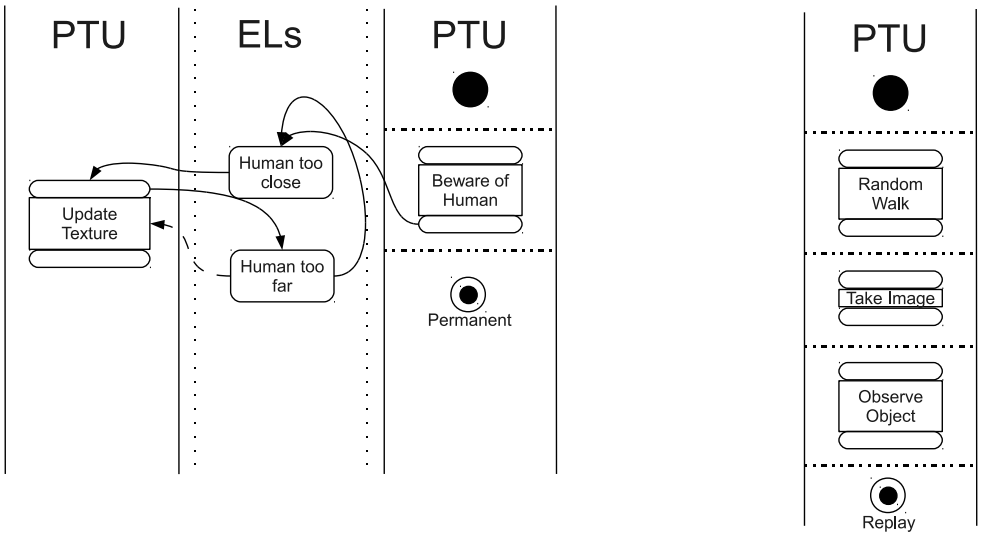


Figure 7.4: *Graphical representation of the tasks in the experiment*: "photograph human" on the left and "search for an object" on the right. The Update Texture Action also includes image taking.

7.1.2 Find object and take picture of human

In the next experiment, the PTU Resource was divided between two tasks: to "photograph a human" and to "search for an object" (Figure 7.4). The experimental setup is presented in Figure 7.5. In the "photograph a human" task humans are detected and if one is found, the PTU is directed toward the human and a picture is taken. This picture is then stored as a texture for the human object in the world model. In this task, EL is responsible for watching whether a human has entered the scenel, in which case it would add a picture-taking Action to the AP of the PTU. Meanwhile, a human-detecting perception agent is continuously updating the world state. Only the reference to the object in the database is communicated between EL and Action.

The perception agent localised and recognised humans on the basis of the laser scanner (Figure 7.6). It detects the legs from around knee height and uses the following heuristics to differentiate humans from each other and from objects similar to humans:

- Minimum and maximum leg diameter
- Leg's cross-section is roundish
- A human has two legs
- Legs can not be too far from each other
- The human COG is between the legs
- Legs can move
- Usually there is a minimum distance between two humans
- Humans have a maximum velocity

Figure 7.5: *Experimental setup for MARY in pose reservation experiment*

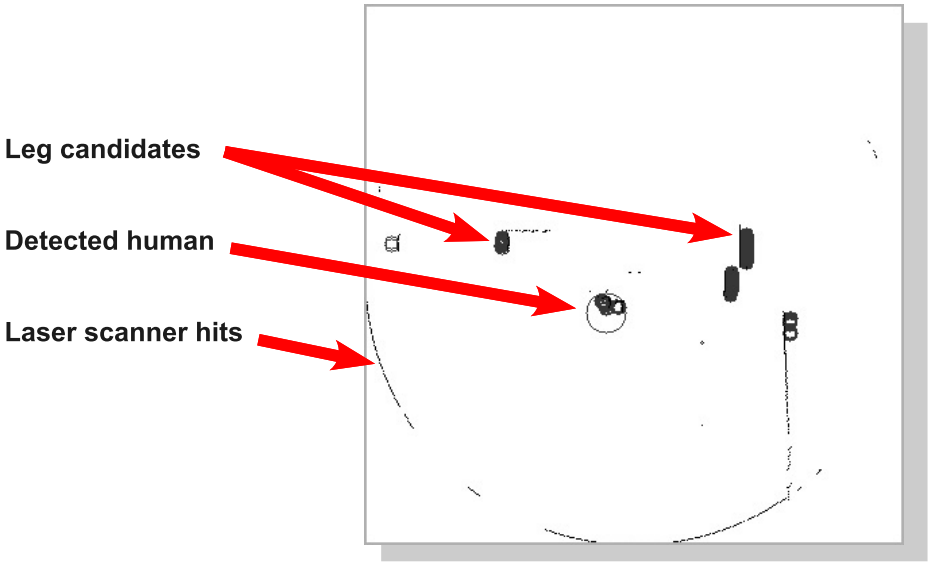


Figure 7.6: *Snapshot of laser scan and human detection*

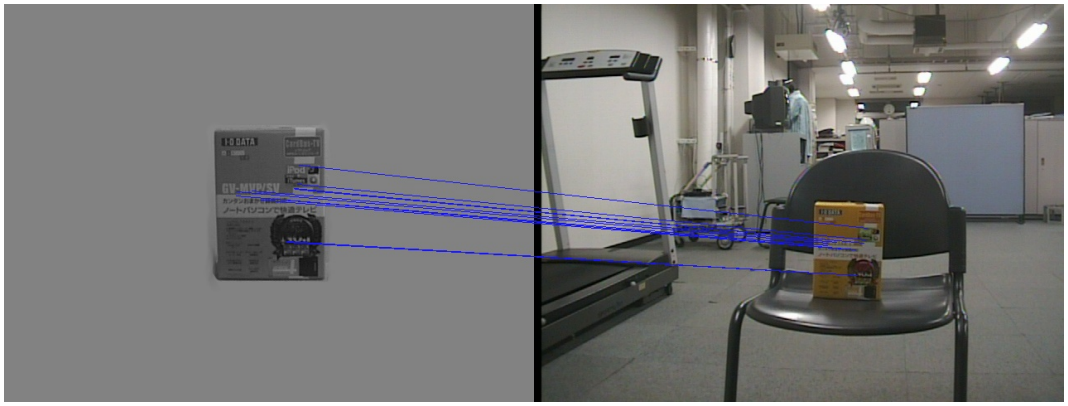


Figure 7.7: *Testing of trained object features for search function*

The perception agent utilised a Bayesian filter to differentiate humans from objects and to estimate their motion. The algorithm proceeds as follows:

Algorithm 3 Human detection with laser scanner

1. Detect clusters from laser scanner hits
2. Estimate the pose of previously located clusters
3. Match and pair the detected clusters from previous clusters
4. Use the heuristics to evaluate whether the cluster is a human leg
5. Update the legs to previously found humans
6. Use the heuristics to find humans from leg candidates
7. Update the likelihood of the humans found actually being humans

In the“search for an object” task, the PTU is randomly turned in different directions and the object in the camera image is searched for using SIFT [113] feature points (Figure 7.7). A reference set of feature points is acquired beforehand as follows. Reference pictures of the object are taken at different angles and distances. For each picture the SIFT feature points are detected. Only feature points found in all images are used as a reference.

The scale part of the feature point is used, together with the knowledge of the distance to the object in the reference image. By averaging the ratio of the detected feature points' scale and the ones from the reference a fairly good estimate of the distance from the camera could be achieved. The bearing to the object was calculated from the average of the feature points in the image plane and from the known

orientation of the PTU. From the distance and bearing to the object its location could be estimated.

This "search for an object" task is, by nature, continuous or permanent, meaning that the task is started from the beginning after its execution. It also has a lower priority than the previously described task of "photographing humans". So, when the latter task is added to the AP, action selection is triggered, resulting in a prompt reaction.

7.1.3 Results

The base Resource reservation experiments were successful. The experiments with the"photograph a human" and "search for an object" tasks were also successful. To illustrate this a snapshot of the world model was recorded before (Figure 7.8) and after (Figure 7.9) running the tasks with a human passing by. The snapshot was saved as an X3D file and viewed with an X3D browser. The textures used for other objects than a human (or in this case a mannequin) were obtained manually.

Naturally, there were some challenges on the way, mainly caused by the camera and the PTU. The camera turns relatively slowly and if the human is passing the robot too close and along a perpendicular trajectory, the PTU may not be able to keep up with the human. The output of the camera unit was an interlaced analogue composite video captured with a USB video capture module. Thus, the images of the moving target were distorted. Furthermore, the PTU was designed for surveillance use and its position information had some latency which made it difficult to synchronise with the human position retrieved from the laser scanner data. These limitations are natural shortcomings of the non-ideal hardware. This shows that the ActionPool method can be implemented and utilised up to the limitations of the hardware and it can cope with them.

Figure 7.8: *World model with empty experimental space and MARY*

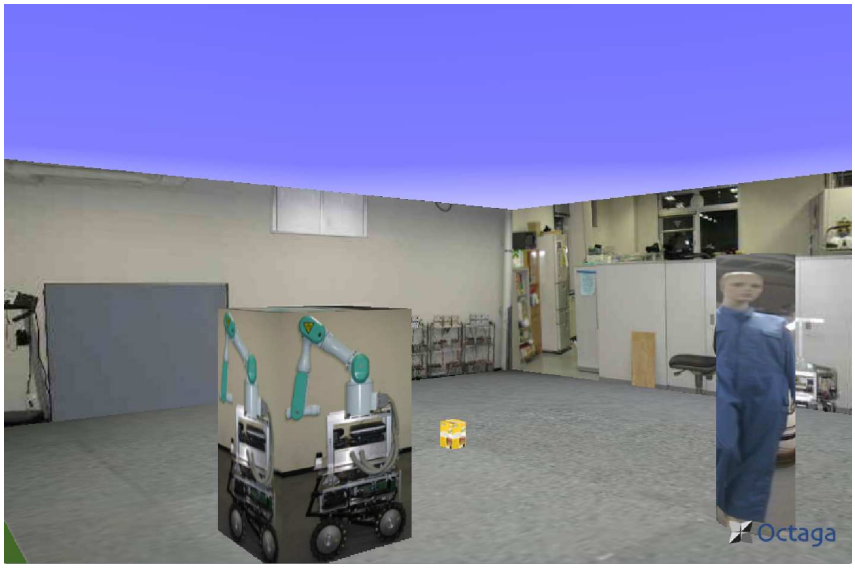


Figure 7.9: *Projection of MARY's world model after having found person and object*

Figure 7.10: *Experimental setup for Rolloottori*

7.2 Rolloottori

Rolloottori is a classic design of a differential drive robot originally built for home automation for the elderly [11]. Rolloottori is equipped with a Sony PTU and a Hokuyo URG laser scanner. The experiments with Rolloottori involve two APs, Resources of the robot's base and PTU corresponding to control of the robot's pose and the direction of camera. The GIMnet software [23] is used for reactive level control. Localisation and navigation are based on the branch and bound localisation method of [114], road-map path planning, and reactive multi-stage obstacle avoidance inspired by the vector field histogram. The experimental setup is presented in Figure 7.10. Below, we explain four of the experiments conducted with Rolloottori in greater detail.

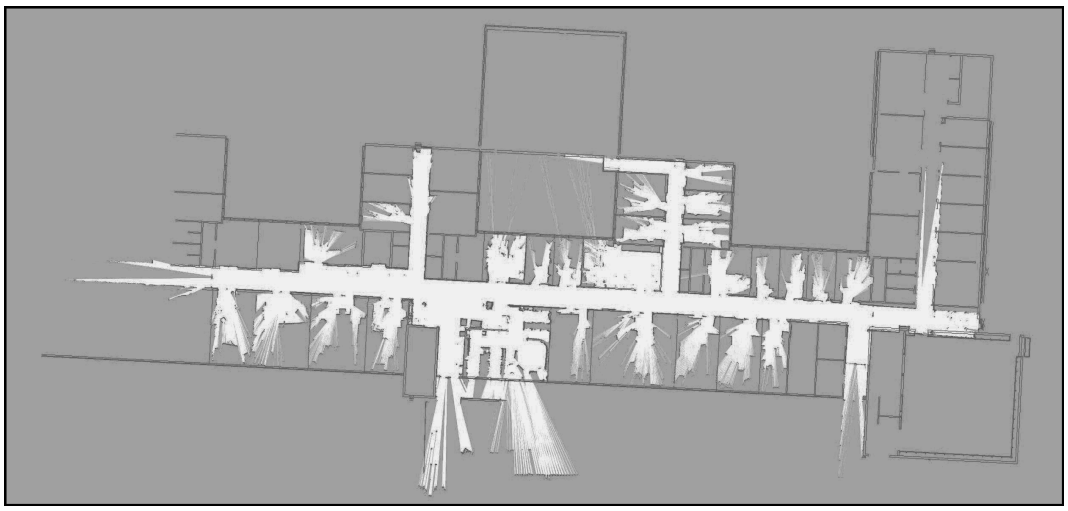


Figure 7.11: *A sample of the occupancy grid map of the experiment environment used for localisation, as viewed from above*: The darker regions correspond to areas with greater occupancy such that an area in black is completely occupied, gray is unkown, and white corresponds to free areas. Includes initial data from CAD drawings and data from laser scans.

7.2.1 Pose reservation

The experimental space is an office environment. Four maps are used for the experiment. The first one is a vector based line map of the walls used for visualisation in the graphical user interface (GUI). The map was obtained from the CAD drawings of the experimental space. With the GUI, the robot's current location and sensor readings could be monitored. An other simplified line map of the walls, where the number of lines is reduced, is used to obtain an obstacle-free path in the experimental space. That is combined with a road-map, also represented as a line map (Figure 6.9) to do the path planning between two arbitrary positions. The last map is an occupancy grid map illustrated in Figure 7.11, and created from numerous laser scans in the environment. The last map is used for localisation purposes only.

The experiment consists of a number of manually added "Random goto" Actions in the base AP of the robot. The "Random goto" Action selects one of ten predefined free locations from the environment and considers that to be the target location for the Action. Naturally, the Action did not have any aPlan for the target location.

The environment was in regular use as an office during the experiments. In practice that means that there were dynamic and static obstacles present and the robot was even slightly "harassed" from time to time.

7.2.2 Texture Mapping of a Wall Segment

The structure of the experimental space, with its walls divided into segments, is stored in the DB. To have a more accurate model of the experimental space, these wall segments need to be texture mapped. Considering the distance between the robot and the wall, a wall segment is too big to fit into one frame. Therefore, multiple pictures from different locations are needed to texture map a wall segment. Only the DB references of untextured wall segments are given to the task by the user.

In this second experiment, the robot acquires a texture for a single wall segment. This demonstrates the usage of interdependent Resources of the PTU and robot base (Figure 7.12). The first Action in the task calculates the total number of images required for texture mapping the wall segment and the location and angle from which each image has to be taken. Figure 7.13 illustrates the calculation process. Next, for each of these images a picture-taking Action is added to the base's AP. The Action corresponding to the image whose location is closest to the robot, based on the shortest expected reservation time, is selected first (the expected execution time for the aPlan is the same for all Actions). The robot then reserves the base Resource by traversing to the picture-taking location. At that location, the aPlan adds a picture-taking Action to the pool of the PTU and waits for its execution. The PTU's AP then reserves its Resource by directing the camera to the desired section of the wall segment. This process is repeated until all the images have been taken and then an image-stitching Action is added to the base's pool. The resulting image is stored in the DB. The whole process is shown in Figure 7.12.

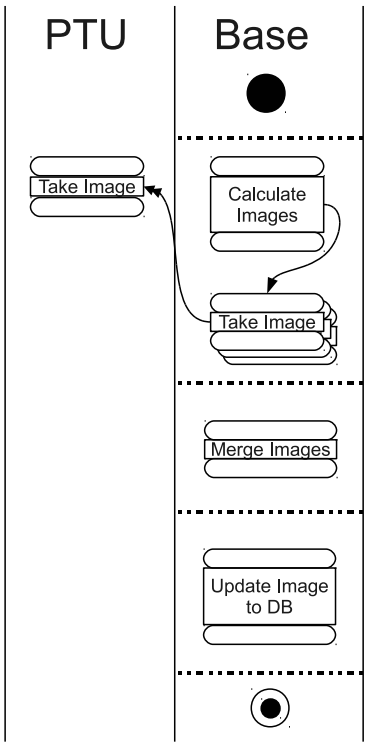


Figure 7.12: *Graphical representation of "texture mapping a wall segment" task*

The target for a Resource to reserve in an Action can be expressed in two ways: i) as a reference in the DB or ii) in absolute world coordinates. A reference to the robot itself is used for Actions, such as image stitching, which require only processing power and where the Resource is irrelevant.

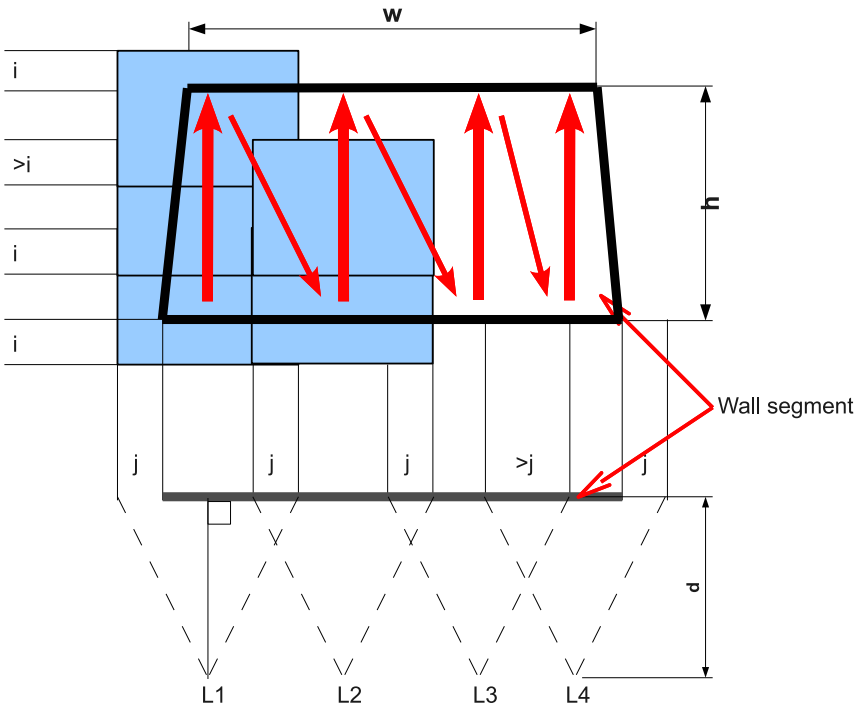


Figure 7.13: *Illustration of the algorithm to define image-taking parameters in "texture mapping of a wall segment" task*: Above is the wall segment from the robot's perspective and below is the wall segment viewed from above. The size of the wall segment (w,h) is acquired from the DB. The overlap parameters i and j, as well as the robot's distance from the wall d, are parameters for the algorithm inside the Action. The outputs are the locations L1...LN and the centre points of the images to be taken from those locations. The distortion caused by the perspective is compensated for in the image-stitching process. The image calculation order is defined with the arrows. In the case illustrated, three images, from the bottom up, are taken from four different locations, making a total of 12 images to be taken for the texture map.

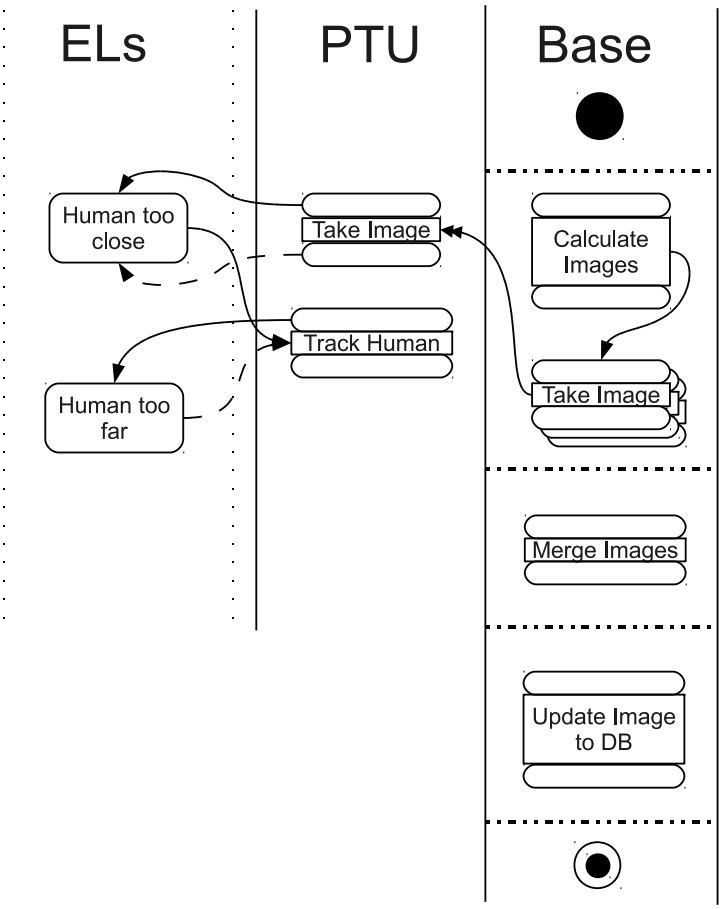


Figure 7.14: *Graphical representation of "texture mapping a wall segment with exception" task*

7.2.3 Texture Mapping of a Wall Segment With Exception

It is inevitable that while Rolloottori is moving autonomously in the laboratory, avoiding objects and taking photographs of walls, the random people in the corridor would also show up in the captured images, which would corrupt the final texture of the mapped wall image. In order to avoid this, a human-detecting EL is introduced to be activated during the picture-taking Action. This demonstrates the exception handling and the gradual evolution of the task in the ActionPool method. The task is described graphically in Figure 7.14.

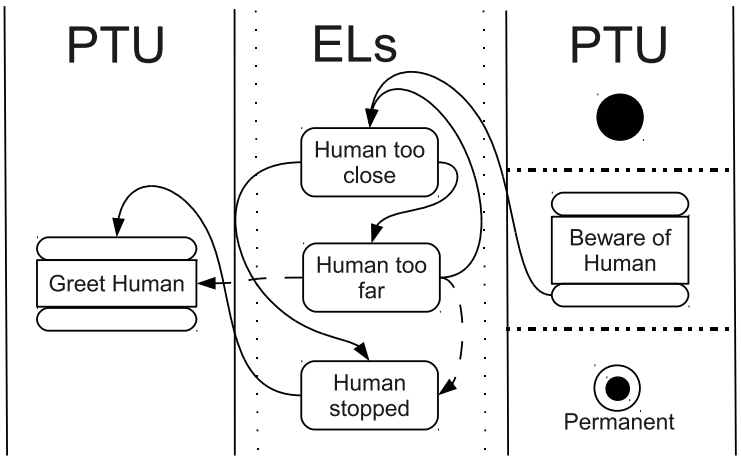


Figure 7.15: *Graphical representation of "greet humans" task*

7.2.4 Texture Mapping of Wall Segments

The last experiment involves two different tasks: wall segment texture mapping and greeting humans. The "greet humans" task (Figure 7.15) is similar to the "photographing humans" task, with the difference that one more EL is added. This EL checks whether the human also stops by the robot and, as a response, greets him with an audible greeting. Now there are two tasks involved in tracking the human. The behaviour of these functions can be tuned by task priorities and ELs' thresholds.

The experiment demonstrates time-sharing and concurrent multi-tasking. The PTU resource is divided between the two tasks mentioned. The base resource is divided between several instances of the "texture mapping a wall segment with exception" tasks working on different wall segments. The concurrent multi-tasking occurs when the base Resource is occupied by the wall texturing tasks and meanwhile the PTU Resource is used for the "greet humans" task.

7.2.5 Results

The first pose reservation task was successful. The second experiment to texture a wall segment worked out well too. The path the robot took is shown in Figure 7.16. The texture created by the task can be seen in Figure 7.17. The effect and usage of the textures can be seen in Figures 7.18 and 7.19. The results of the last two experiments are presented as time steps (Figures 7.20 and 7.22) describing what is happening inside the Action pools while the task or tasks progress.

Figure 7.16: *Path of Rolloottori robot while texturing a wall segment*: the robot approaches from the right and stops at five locations to take images

Figure 7.17: *A sample of a stitched wall segment texture*: The stitch marks in the image are due to the auto gain of the camera but it can still be used as a texture map.

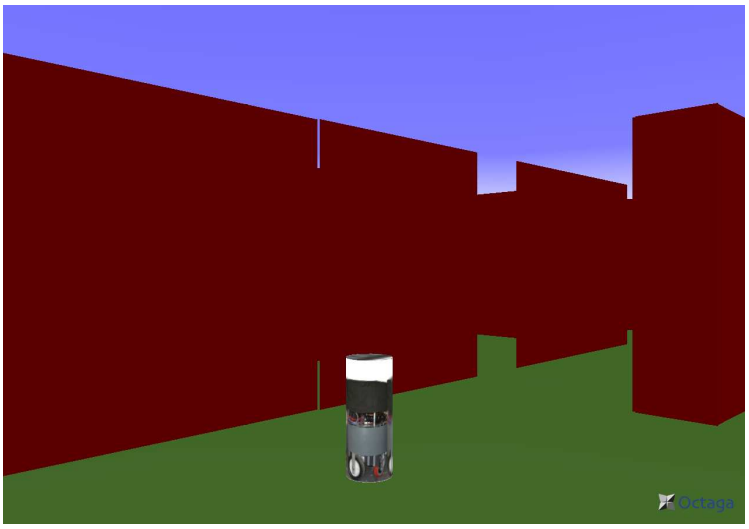


Figure 7.18: *An empty world model of the Rolloottori*

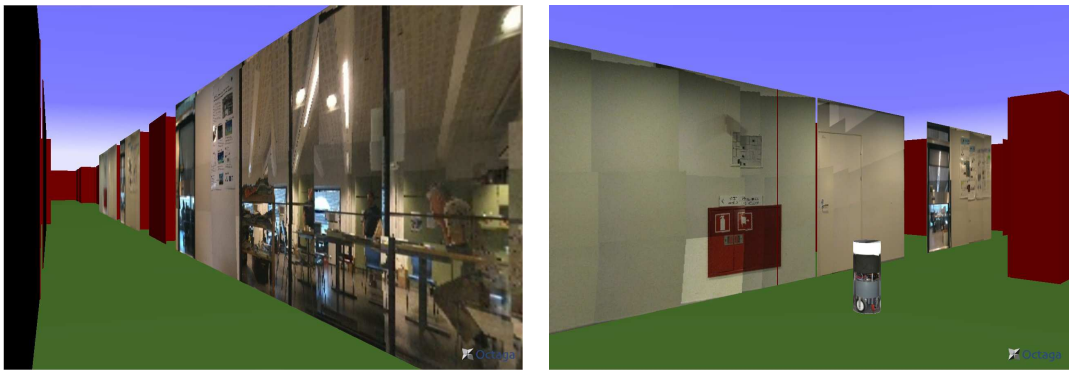


Figure 7.19: *Textured world model of the Rolloottori*

Texture Mapping a Wall Segment With Exception

Figure 7.20 illustrates a part of the execution of a "picture-taking" task as a human approaches the robot. Figure 7.21 illustrates the same task in a timing diagram. The detailed steps taken in this process are explained below.

Step 1 The first Action calculates the locations and angles from which to take images and adds Actions to do just that to the base AP.

Step 2 On the basis of the Action selected from the base AP, the robot is driven to the target pose. The aPlan of Action in base AP adds a "Take Image" Action to the PTU AP. Base AP awaits for the Action in PTU AP to be executed.

Step 3 Action in PTU AP is selected which initiates "Human too close" EL. A human approaching the robot triggers the EL to add a "Track Human" Action to the PTU AP.

Step 4 Because of the addition of the above Action, the "Take Image" Action is interrupted and the "Track Human" Action is started. This Action tracks the human with PTU and initiates a "Human too far" EL. This EL is triggered once the human moves further away and as a response, it removes the "Track Human" Action.

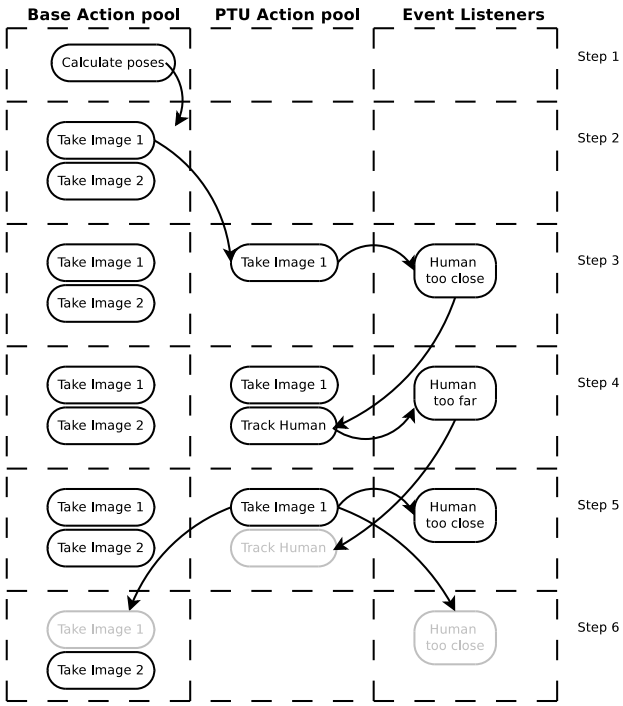


Figure 7.20: *Content of different Action pools and Event Listeners during different stages of "Texture mapping a wall segment with exception" experiment*: See the text for an explanation of the steps. The time steps go from the top to the bottom.

Step 5 The "Take Image" Action is reselected and the camera is directed for picture-taking. The aPlan then saves the image for stitching. After the execution, the Action is removed from both the PTU and base AP.

Step 6 The Base AP selects a new action for execution. If there are any Actions left in the PTU AP, both the base and PTU APs' Actions would be selected and executed in parallel.

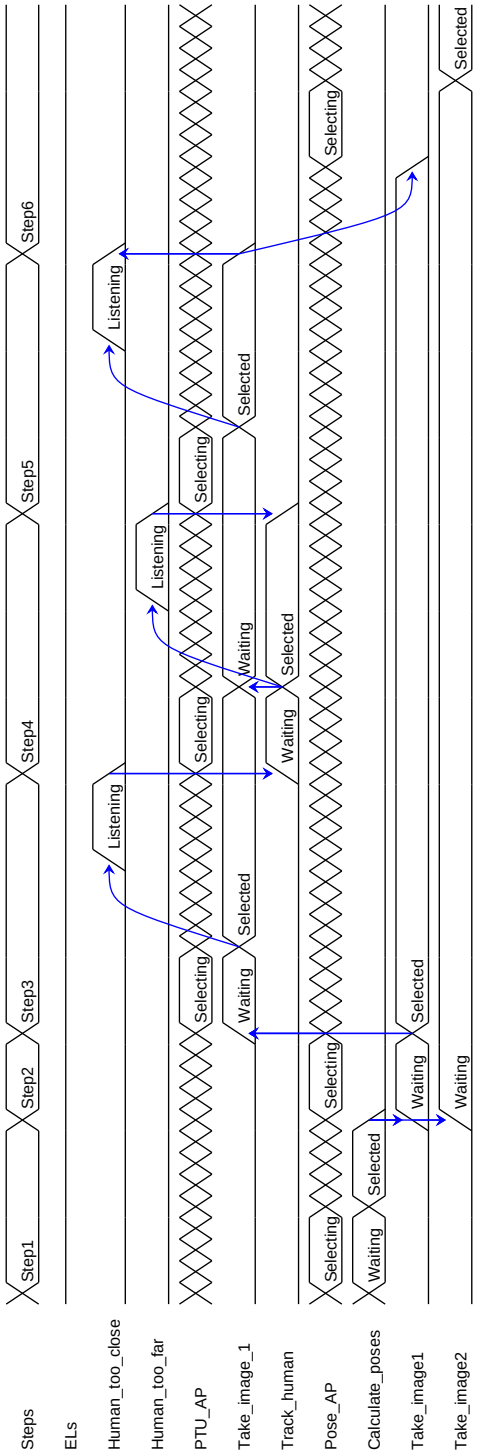


Figure 7.21: *Timing diagram of "Texture mapping a wall segment with exception" experiment*: see the text for an explanation of the steps. Time goes from left to right.

Texture Mapping of Wall Segments

Figure 7.22 shows the steps taken by the robot when it "texture maps a wall segment" (which was shown in Figure 7.20) and "greets a human" in a multi-tasking fashion. As a result of this additional task, the following would be added to the previously described steps:

Step 1 The "human greeting" task initiates the "Human too close" EL.

Step 3 The newly added "human greeting" task's "Human too close" EL is triggered and as a response the "Human too far" and "Human stopped" ELs are triggered.

Step 4

a) Once the human stops by the robot the "Human stopped" EL adds the "Greet Human" Action.

b) The "Greet Human" Action is selected and executed. When the human moves away, the "Human too far" EL is triggered and it itself initiates the "Human too close" EL in preparation for the next repetition of the above cycle.

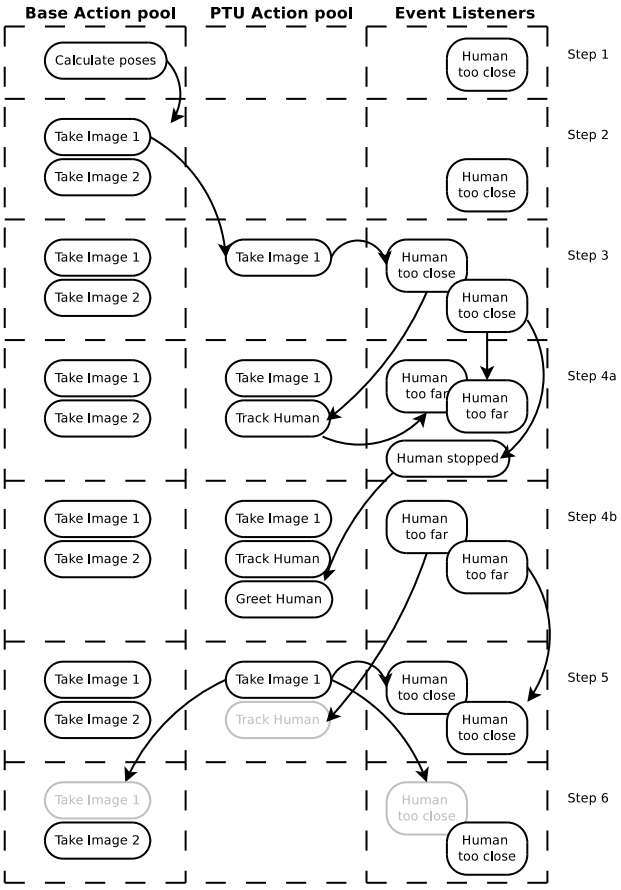


Figure 7.22: *Content of different Action Pools and Event Listeners during different stages of "Texture mapping of wall segments" experiment*: See the text for an explanation of the steps. The time steps go from the top to the bottom.

Chapter 8

Analysis

The objective evaluation of the performance level of a complex system, such as a service robot, is an undervalued challenge. The performance can be evaluated from many perspectives. The evaluation cannot be done simply on one scale of good or bad performance. As mentioned earlier, there are many measurable performance criteria that contradict each other, such as speed and reliability. For reliable evaluation several aspects of the performance should be assessed.

The dynamic operating environment dictates that the experiments should be done with a great number of repetitions to statistically cancel out the effects of uncontrolled variables. Unfortunately, these complex systems are always challenging to implement. As a result of limited time and material resources, even some of the variables that are controllable in theory are not so in practice. So even if numerical results are extracted they would not only reflect the ability of the architecture to execute tasks in the environment but also the quality of the overall system implementation.

One objective evaluation method for overall performance, within the border conditions used in this work, is presented in robocup's @home league [17]. The evaluation in robocup is still mainly against other robots in the same competition and thus it can not be conducted independently. Some of the evaluation was based on a collection of subjective opinions of peers. Furthermore, the target for the evaluation in @home is the complete robotic system and the features isolated for evaluation are related to perception, skills in using the actuators and the user experience, and not the control architecture as such.

For the evaluation, the perspective should be based on the problem setting described in Section 1.2. In the following Sections, the ActionPool method is first analysed independently and then the analysis is extended to related work.

8.1 Non-Functional Requirement Analysis

One evaluation method used in the literature involves setting a list of features and seeing how well the system fulfils those features. Unfortunately, the creation of the feature set and its evaluation is unavoidably a subjective process. Regardless of that, this kind of feature set-based analysis is one of the most commonly used methods and it is partly used here too. The feature set derived from the features that motivated this work and the features identified by others were combined in Section 2.7 into Table 2.2. The proposed method is now analysed on the basis of this table.

Programmability (1)

Many different tasks were demonstrated in the experiments. The Tasks and Actions were programmed using an XML-based language that was interpreted during the execution. In practice the Actions from a Task that are not already in the pool can be modified when the Task is otherwise running. The programming of the Tasks can occur on many levels. uTasks, aPlans, Actions, ELs, and tPlans are all programmed separately.

Adaptability (2)

The Action selection process assesses the internal and external state of the world. Actions in the pool and active ELs at any instant in time describe the internal state. The internal state is modified by the recent changes in the environment through ELs changing the priorities of the Actions and the Actions themselves. The external status is then coded to the relationship between the current state of the Resource and the locations of the Actions. The selection of the Actions then uses the cost function to find a suitable Action for the situation. The Action selection itself is triggered by changes in the operating environment, i.e. changes in the Action setup.

Reactivity (3)

The pose reservation experiments showed that the real-time layer responded to the quick dynamic changes that are typically present in an environment shared with humans. The fastest safety reflexes of the system, for example obstacle avoidance, are, and should be, coded into the real-time layer. For higher levels the "Find object and take picture of human" and "Texture Mapping a Wall Segment" experiments clearly demonstrated the Event Listener mechanism reacting to the changes in the environment.

Predictability (4)

The predictability has two flavours; 1) can the future operation be predicted from the experience of what happened last time in the same situation and 2) can the future operation be predicted from the current operation, i.e. is the behaviour stable?

The former is affected by the Action selection process. A greedy selection policy creates more predictable behaviour than the roulette selection. In some cases, an exploratory and unpredictable behaviour could be desired and then the cost function could consider how many times the Action has been selected previously. The latter flavour is affected by two mechanisms. First, the cost function (Equation 5.1) considers the wind-down cost of the Action that is currently running, making it more likely to be selected. Second, when an Action from a Task is selected, the priority of the Task's other Actions is considered as being slightly higher.

Priorities are one of the key factors in the cost function of the Action selection process. The effect of the priorities is demonstrated in the "Texture Mapping of Wall Segments" and the "Find object and take picture of human" experiments. In the first one, the "Greet human" task had a higher priority than the "Texture map wall segment" task. In the second one, the "Take picture of human" task had a higher priority than the "Find object" task.

Robustness (5)

There are many mechanisms in the proposed algorithm to cope with the uncertainties and they proved to work in the experiments. The world model has uncertainties measures for each object it contains. The Action selection process considers the current probabilities of the resource reservation and plan execution at each selection. Event Listeners react to changes in the probabilities of whatever condition they are set to follow.

Pose reservation could be done easily, while the variation in the pose information changed all the time during the experiments. The target for the object search was always originally unknown. The human detection merged several observations, improving the certainty that the object observed was in fact a human and that he was at the correct estimated place. The "Texture map wall segment with exception" Task demonstrated how the known effects of the environment can be compensated in the Task itself.

In a more serious case, an Action or a whole Task can fail. Thus the Action is considered not done and can be placed back into the pool or it can be removed altogether. In such cases, a new Action from the remaining ones is simply selected for execution. The whole Task can be removed together with its Actions from the pool.

The failure of one complete Resource will not affect the other Resources directly, but they are still interdependent. If a remote Action is sent for execution into a failed Resource, the sending and thus the sending Action in the original pool would fail. As a natural effect, only the Tasks dependent on the failed Resource would be obstructed and other Tasks would continue uninterrupted.

Extensibility (6)

The incremental extension of the Tasks was demonstrated in the experiments with the "Texture map a wall segment" and "Texture map wall segment with exception"

Tasks. Similarly, the addition of new Tasks was demonstrated in the "Texture mapping of wall segments" experiment in Section 7.2.4. The AP itself is not aware of other pools but the Tasks are. There are no architectural restrictions on adding new APs, even during the execution of Actions in the old ones. In fact, that is exactly the case during the system start-up. First, the base pool is started and and tested and later the PTU pool is initiated. In some testing scenarios, it is sufficient to simply start the PTU pool.

Multi-tasking (7)

This is one of the main motivations for the development of ActionPool. Time-sharing the ActionPool divides the limited resources of the robot between different tasks. This was demonstrated in many experiments. The "Find object and take picture of human" experiment used a single resource for two tasks simultaneously. The "Texture Mapping of Wall Segments" experiment used two resources of the robot between the concurrent multi-tasking numerous instances of the "Texture map wall segment" task and a single instance of the "greet human" task.

Resource management (8)

Different Resources can be used simultaneously by one or different tasks. The case of different tasks is an inherent feature of the ActionPool method, where each Resource has its own independent Action pool. Unfortunately, Resources are interdependent and it means that the use of Resources should be synchronised many times. Three methods were mentioned in Section 5.6.1 for the case of a single task using different Resources at the same time. If there are different tasks there are two options: 1) the Action for the Resource is independent of the state of the other Resources and can be executed independently and 2) the Action is dependent on the state of another Resource and thus the Resource(s) that is higher in the dependency tree has to be reserved first, despite not being explicitly used. So, in other words, the second case reduces to a single task case.

The "Texture Mapping a Wall Segment with exception" experiment demonstrated the single task case, because both the pose and the PTU direction were critical for the success of the task. The "Texture Mapping of Wall Segments" experiment demonstrated the case of different tasks, while the "Greet human" task used only the PTU Resource independently of the pose of the robot. The only critical measure for the task was the distance of the human from the robot.

Attention control (9)

Separating the control of the perception was one of the original motivations for creating the ELs.

Other computing units could be used to separate the slow and heavy perception functions but the Task flow should stay responsive and be controlled by its own computing unit. The information that ELs are observing is relevant to the situation

and thus controlling the robot's attention. There is also a measure to adjust the observation frequency of each EL in order to optimise the computing needs.

As will be mentioned in Chapter 9, having the parallel ELs calculate the same observations from the same data was a very inefficient method and thus the perception agents were created to improve the efficiency of the technique. The perception agents now operate independently and future work will deal with controlling their execution frequency on the basis of the active ELs and again gaining situation-aware attention control and the more efficient use of computing resources.

World model (10)

When the complexity of the tasks grows, the need to remember things about the task and the environment increases accordingly. If the task is considered to be a command or a request sentence in a linguistic sense, both the predicate and the subject should be known and remembered. The tasks themselves could be considered as the predicates and their parameters as subjects. In the proposed method, subjects are objects in the world model. The world model is also utilised in the consideration of the next Action. All this has been demonstrated in the experiments. For example, the "greet humans" task has an EL that reacts to the change of human objects in the world model. Furthermore, the "texture map a wall segment" task calculates the poses from which to take the pictures on the basis of the world model, before the direct perception of the wall segment.

Sensory information (11)

The perception agents utilise the sensory information to perceive information from the environment. They compress the sensory information into observations about the environment and store it into the world model database. In the real-time layer of the system, the sensory information is utilised directly by the reactive control modules. Typically, no higher-level observations are made in those cases. The distributed simultaneous use of the sensory information was provided by the Player [9] or GIMnet [23] middleware.

Interleaving planning and execution (12)

The planning, as in the sense of creating a plan for the task, is currently done offline by the user, so the expensive planning operation does not hinder the Task execution. On the other hand, the Action selection process can be considered as some kind of short-term inter-task planning based on the current situation. Action selection is, though, a fast operation compared to the execution time of the Action itself. Furthermore, the system is distributed and the Real-time layer and aPlan layer are responsible for the execution of the aPlan. Typically, they are isolated on their own processor to guarantee responsiveness. The distribution makes planning and execution parallel processes. If the plan or Action for some particular task is not ready by the time the Action is selected, an Action of some other Task can be

selected instead. If the computing power is available, a slightly more far-sighted online planning could be done between the Action selections. That possibility is elaborated further in the future work section in the next chapter.

Modularity (13)

The modularity helps in the development process and is also a strong motivator in the proposed method. First, the same ActionPool method was implemented in two different robots, so the whole method can be considered as a module. Second, the architecture was divided into modular layers communicating with each other. The whole layer is interchangeable if the communication interface is kept constant. Third, an AP is a module communicating with other APs and EL modules. Fourth, the perception agents are independent modules providing information through the abstract shared memory, i.e. the database of the world model. Fifth, Tasks are independent modules of software developed independently. Tasks are divided into modules by the Actions and ELs related to them. The Actions inside a pool are independent modules. The modularity that is used and the distribution of the control are illustrated in Figures 6.8, 6.7, 6.5, 6.6 and 5.2.

The modularity encourages code reusability. A new system or functions could be created simply by rearranging the components. Two "human too close" ELs from two different tasks were running simultaneously in the "texture map wall segments" experiment. They were just two instances of the same EL but with different thresholds. At a later time instant in the experiment, there were two ELs, "human too far" and "human stopped", that were active at the same time. They utilised not only the same algorithm and program code for the detection of a human but the very same instance in the form of a perception agent.

Platform independence (14)

Platform independence has been designed to take place on discrete steps. The steps are between layers (Figure 6.5). If the robots are copies of each other, the whole Task and everything related to it can be directly and completely transferred. The transfer would be on the real-time layer level. For robots that are structurally close to each other, i.e. similar in size, mobility, sensor placement etc., the Task can be transferred from the aPlan layer. The aPlans could be the same but the uTask (Figure 6.6) would need a robot-specific implementation. uTasks abstract away the differences between the middleware and other implementation details. If the robots are the same in terms of category but differ somewhat in size, sensor placement, etc, the same aPlans are not most probably applicable any more. Then, the Task can be transferred from the mission layer, of course with the prerequisite that the same Actions are implemented for both robots.

The perception agents are more independent of the overall construction of the robot, as long as the sensor required by the agent can be found. Thanks to the middleware the exactly same implementation could be used in several different robots.

Naturally, the implementation should be, for example, sufficiently advanced to consider the placement of the sensor in the robot which is something that is provided by the middleware.

The platform independence of the ActionPool method has been demonstrated by utilising two different robots. The Tasks for the PTU pool could be transferred to the plan layer and the Tasks for the pose pool could be transferred to the mission layer. The transferred Tasks are the rather simple "random walk" Tasks. Some plans for further investigation regarding Task transfer are provided in the future work section.

Independent representation (15)

Task independence is the cornerstone of the proposed method. It is the fundamental assumption. Task independence can have two levels: 1) independent development and 2) independent execution. Independent execution translates into abilities to multi-task and to start and stop Tasks arbitrarily.

Independent development is very important because it allows the integration of efforts from several developers. In practice, this means modularity, discussed previously, and inter-task interaction, handled by the AP and the abstraction level of the Action. The multi-tasking has also been discussed previously. As described previously, they have also been tested..

The ability to start and stop the Tasks arbitrarily was not explicitly tested, although every time experiments were performed, the robot started in an idle state and received the first task at an arbitrary time instant. The subsequent tasks were added by the user at unspecified time instants, without any impact on the performance of the system. During the course of the testing and developing of different tasks, tasks were added and removed at will. This was done through the simple GUI presented in Figure 6.10.

The proposed architecture was purpose-built for this kind of use. When a Task is added, its first Actions are just added to the pool. When a Task is removed, its Actions are removed from the pool. If the Action under execution belongs to a removed Task, it is simply interrupted and a new Action from the remaining Tasks is selected.

Execution control (16)

This feature has to be built into the lowest functional element possible in order to provide a prompt response. All uTasks have methods for stopping, pausing, and continuing them selves. In the case of a uTask that takes a very short time to completely execute, the implementation of these methods may be omitted by the creator. In the higher-level components, aPlan, Action, and AP, these same methods and respective states are also implemented. The implementation for all higher-level components is embedded into the execution system. This was tested many times while the experiments were being performed and the whole system developed, simply because of its practicality. There was no separate experiment that was designed particularly for this feature.

Performance analysis (17)

As mentioned before, there was no specific experiment just for this feature. To create this feature an AP has a query and an event-sending mechanism. The current state of the pool could be queried and the answer would include the Tasks that had been executed, Actions in the pool, and the Action currently under execution. For monitoring purposes, a simple computer GUI element was created (Figure 6.10). A utility program to connect to an AP instance and log its events in a file was also created, along with logging of the messages sent by the AP process itself, even though no further detailed analysis of the logs was performed.

A similar arrangement was made with the ELs. ELs are managed by one process from which the current ELs could be queried and ELs manually manipulated. A simple GUI program (Figure 6.12) then connects to the EL manager and presents its information.

8.2 Analysis Compared to Other Works

There have been many architectures constructed to control service robots over the years [86, 21, 98, 115]. There has also been a number of languages defined to describe the tasks performed by service robots as well [13, 116, 59, 61, 92]. They all try to achieve at least partly the same objectives. So it is not surprising to find similar functional elements amongst them. Table C in appendices illustrates some connections between these different approaches. Even though there are similar elements, their combination and co-operation in the ActionPool method is unique.

Table C indirectly compares the performance of functionalities of the different approaches in the literature. The selected terms or elements in the table also reflect functions in the system. Not all the functionalities in different approaches have an exact or identifiable correspondences to the elements in ActionPool and thus are not listed in the table. Furthermore, some missing functions can be compensated for by other functions. Still, the table gives a general picture of how the ActionPool corresponds to the related work.

8.2.1 Comparison With Some Plan Representation Methods

The proposed method has a unique ability to mix the Tasks dynamically. The two tasks “greet humans” and “texture map wall segment” are described with a state diagram in Figure 8.1. The same figure shows a combination of the two tasks in one state diagram. The combination had to be manually done to ensure the correct interaction between the tasks, while the proposed methods naturally fold them together as if they were the same task. Some automation for the merging of state diagrams can be carried out by utilising specific states that are common in both tasks [87], but that is something that has to be explicitly considered during the development phase.

It is also worth noting that the state diagram cannot express, for example, “Take any of these 16 photos in the order you see best.” With a state diagram, you would

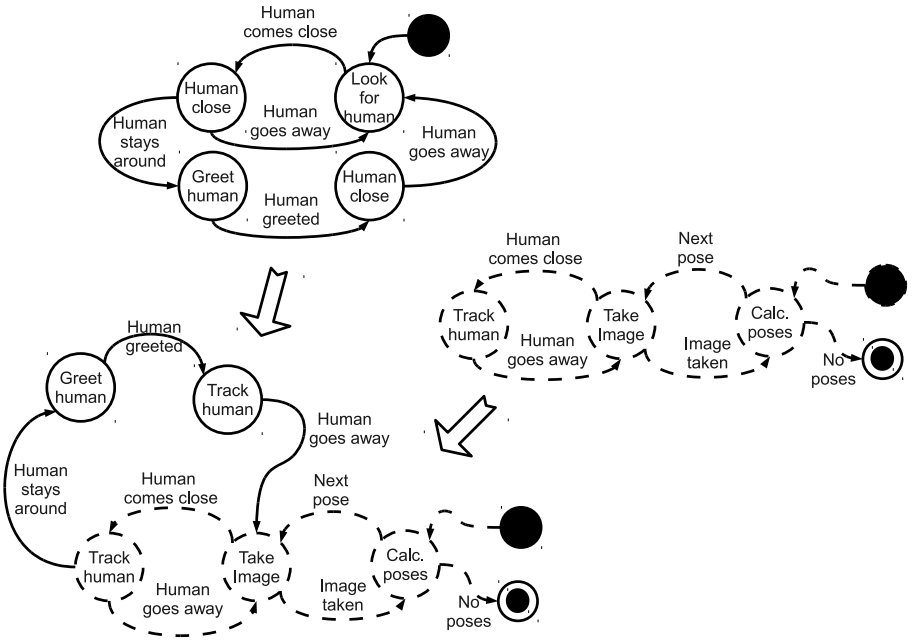


Figure 8.1: *State diagrams describing two tasks and their merged task*

have to iterate through every photo individually or you would have to programme the algorithms to asses the situation in each individual task.

The situation is not very different with Petri-nets. They are more difficult to combine because of the non-unique state and tighter relations of the states and transitions, i.e. the transition can occur only from and to a specific state. With functional representation it would be even more difficult. There, by definition, the representation does not limit the state of the robot in any way and thus there could be race conditions between the tasks.

Procedural and trajectory-based representations more exactly dictate what has to be done. They also have a shortcoming when it comes to task independence. The representations do not convey any information about the abstract division of the states, shattering the possibility of finding safe points for the context switch. Additionally, the complexity of the general expressiveness causes considerable overheads.

8.2.2 Evaluation with Task Execution Principles

Seemingly the closest relatives to the ActionPool method are different derivatives of the BDI paradigm [117], such as PRS [65, 30]. The fundamental difference is that the proposed method survives without explicit notations for reasoning in task's metadata, i.e. using the goals to choose possible plans to use. Many other knowledge-based systems have the same goal-oriented approach.

The knowledge about the task in behaviour-based systems is coded in the interactions of the behaviours. Behaviour networks are more or less fixed and can be seen as a form of *situated automaton*. Even if they work well for a fixed domain, it is still difficult to dynamically change the objectives or tasks of the robot in a multi-tasking fashion. The reactive paradigm faces the same challenges as behaviour networks in dynamic task control.

Hybrid and deliberative systems generally fall short in their requirement to do in situ planning. They can accommodate new tasks and remove others by replanning [118] but it is a slow and difficult operation in the dynamic and open environment.

The task execution in the proposed method is quite an involved process. There are numerous heterogeneous components that interact in the complicated system. This can be seen as a cost of having an abstract and compact representation of the task. It is also notable that there is no one single do-it-all solution for the service robot control and several approaches have to be used. For example, the implementation of the proposed method uses the reactive control paradigm and, in the real-time layer, a procedural plan representation in the aPlan layer and an augmented distributed plan representation to reserve the resources.

8.2.3 Related Work

A related and well studied problem is the so-called *dynamic job-shop* [119, 120]. It involves a shop with a selection of production machines and jobs that have to be processed in a specific order on those machines. In the original job-shop problem, all the jobs are initiated at the same time instant. In the dynamic version, this border condition is relaxed. Since there are more requests from the jobs to use the machines than there are machines available in the shop, some non-trivial scheduling is required for optimal throughput. The biggest difference compared to this work is in the job itself. In this work, the task (job) can have alternative solutions and one phase of the task may fail or take an arbitrarily long time to complete. These times are typically fixed in the job-shop scenarios. Additionally, the resources (machines) are interdependent in this work while in the job-shop they are independent.

Chapter 9

Conclusions and Discussion

9.1 Conclusions

In this work, the ActionPool method of simultaneous multi-tasking for service robots was created. That also incorporates a novel way of representing task knowledge. The method is based on the division of the robot into interdependent Resources and defining the Tasks as a sequence of atomic Actions on the Resource. The Actions are scheduled to the Resource through the Action pool mechanism. The ActionPool method could be classified as following the Belief, Desire and Intention (BDI) paradigm, even though that was not the source of inspiration. The original inspiration for this work lay in the real problems faced in the execution of a task with a service robot.

Six experiments utilising two different robots were conducted to verify the ActionPool method. Other approaches rely heavily on simulations or try to be overly general solutions for a reactive real-time agent. By focusing on the service robot domain and real robots, the author believes that he has found a novel control method for the mission-level control of a service robot. It takes into account the slowness and computational load of the perception process and restrictions of the robot platform in a dynamic and open environment. Common hurdles in the utilisation of the system, such as the independent development of different tasks and perception algorithms, are also addressed.

The task knowledge representation is based on the interactions of Event Listeners (ELs) and Actions. The sequence of Actions in the Task plan (tPlan) corresponds to the classical notation of a plan. The sequence of Actions mainly represents the expected or typical execution of the Task. Each Action carries some information about its execution statistics, Resource usage and importance. This helps in the classification of the Actions in the Action pool (AP). It could also easily be utilised in looking further at the scheduling and classification of Tasks by summing the statistics of the Actions within them. The representation includes the notation of ELs. The EL carries the knowledge of exceptions: a) what an exception is and b) what the response to that is. The knowledge needed to choose the right Task for execution, i.e. the purpose of the Task, is currently only in human-readable form as

documentation included in the Task. The representation of the Task with Actions and ELs can be considered a non-deterministic programming language, where the exact course of execution is defined only at runtime.

9.2 Discussion

The task planning in the proposed architecture was intentionally left for the human operator. The proposed architecture merely reacts to the environment according to the Tasks. It simply defines and follows certain policies in order to deduce what to do next, i.e. schedules. It does not have any higher intentions or urge to motivate the Action selection and it does not have to deduce what to do in order to fulfil those intentions. This is a very fundamental feature and makes a difference compared to deliberative and hybrid control. This way, the ActionPool has some resemblance to the reactive control but the selection of Actions in the pool is constantly changing and the response to the same stimulus also changes over time. Furthermore, Actions are decoupled, and the selection of Action does not occur in a continuous process as is the case in reactive control. Thus, this work does not exactly fit under the umbrella of the reactive control scheme, either.

This does not mean that the system should have no planning at all. The system is designed simply to rely on planning as little as possible. Currently, the hard job of planning is left for humans. The proposed system tries to make this planning job easier by abstracting the task design to Actions and handling the co-operation of different Tasks. With the technology advancement, there is no reason why an automated planner could not do the planning part. In fact, ActionPool enables continuous planning to take place.

As mentioned in Chapter 8, there are numerous methods in the literature to control service robots. In the author's opinion, there is no need to highlight one over another. Nature has shown on several occasions that diversity is essential for survival in a highly dynamic and uncontrolled environment. So we would probably be better off with a heterogeneous base of control methods for service robots, even if that would make the job of task programmers more difficult.

The work in the literature has been surprisingly lightly motivated (declaration and reasoning of the goals for the presented methods) and it is hoped that this work will encourage other researchers to pay more attention to this issue. Another impact of this work on robotics research is that it provides a different perspective to a common problem. The focus in plan representation has been mainly either on the whole architecture and component interaction or on automated planning. The actual implementation and restrictions of the current technology should be incorporated.

Many of the beneficial features in the ActionPool method are derived from the fact that the Actions are isolated from each other. An Action does not know which other Actions are done before or after it. This gives us the freedom to choose any one of the Actions in the pool but that does not come without a cost. There can be undesirable side-effects of different Tasks interacting in the same environment and acting against each other. Furthermore, an Action is assigned to one Task only, so

Actions that would be useful for more than one task can not be combined into one or prioritised to be more important.

Likewise, the way the Tasks are presented also has its side-effects. Goal-based planning or behavioural systems can, as an inbuilt feature, utilise favourable changes in the environment or react to changes in the environment that invalidate the effect of past operations. In the presented system, this functionality has to be explicitly created with ELs and condition checks in aPlans.

In the course of this work, it became more and more evident that perception is a field of technology which is the furthest away from its counterpart in nature (including the sensors and the related signal processing). Perception needs to be developed drastically to enable truly useful functions to be created for service robots.

This thesis work started from studying the problem of a service robot learning to perform a task. Learning can be reduced to two problems: 1) the classification problem and 2) the parameter or function estimation problem. Over the years, many good algorithms for solving these two problems have been developed. The real problem in service robotics task learning is how to or where to apply these algorithms. Another feature of the learning algorithms is their requirement for numerous iterations to converge to the right solution. Furthermore, as the problem becomes more complex, the number of required iterations increases exponentially.

These challenges motivated the creation of several layers and isolation of the different algorithms into ELs and perception agents in the ActionPool method. This provided small-sized units to apply the learning algorithms to. Moreover, the modularisation helps to gather necessary iterations for the learning algorithms as the same module can be used in different Tasks.

The Action sequencing in Task plan (tPaln) has not been as comprehensively thought out as the other parts of the proposed architecture. It can be seen as a separate part from the main method consisting of APs, Actions and ELs. The tPlan sequencing could be changed into the higher-level planner that decides which Actions are worth pursuing. This is similar to the PRS approach. Alternatively, a completely different sequencing mechanism, such as Petri-net, could be utilised. A prominent feature of the ActionPool method is its ability to manage Resources and Task interoperability

9.3 Future work

Perception demands a lot of calculation resources from the robot control and it is easy to exhaust any processing capacity with signal processing functions. This is why some scheduling is also needed for the activation of the perception agent. Some perception processes need some minimum amount of processing power to be applicable at all. For example, the detection of movement from a video feed needs to process two consecutive frames and provide the result before the information gets outdated. Thus, we can not just slow down the not-so-urgent functions by assigning a low priority to their execution process and let the operating system of the computer do the heavy lifting of scheduling. It would not guarantee that consecutive frames

would be captured or that the dynamic information would be au-courant. Thus, when considering more complex and a greater quantity of tasks, the attention of a perception needs to be managed better for the situation at hand.

Currently, the Action selection process does not consider the interruption of an Action being executed very gracefully. The wind-down cost of interrupting an Action that is in the process of being executed is considered fixed. Of course, that is not the case in the real world, especially if the Action execution gets to the aPlan execution phase. In the implementation, there is only one function call telling the Action being executed that it is time to stop. The wind-down process has to be implemented into one function that can not be interrupted. Thus, a lengthy wind-down process, such as putting down a glass of water that the manipulator arm is holding, could block the operation of that particular Resource for that time. There should be a mechanism to even skip the wind-down process if the reason for aborting the Action is sufficiently urgent.

It is possible to estimate the wind down cost easily from the expected execution time and the time already used. Alternatively, the Action could maintain its own wind-down cost value dynamically through the execution. That would be very bothersome for the programmer, because it would be part of the design of every aPlan. Perhaps the wind-down process could be an Action by itself, added to the AP by an EL responding to the event of an aborted Action.

At present, some of the knowledge about the Actions and ELs is only in human-readable form in its XML definitions. If a more computer-readable form for that could be developed, it would not only help the possible automated planning process but would also help the reactive behaviour of the system as well. An attempt could be made to handle an unhandeled exception in the execution process by an EL used in some other Action. This could be done by matching the EL's preconditions with the exception. By evaluating whether the Action could be finished after the corrective Actions from the EL of the other Action, a partly automated error recovery system or some learning could be developed.

As mentioned in earlier chapters, some more planning or slightly more far-reaching Action scheduling could be done. Any planning would require the estimation of what is going to happen in the future on the basis of the actions the system takes. In the Action-Pool method, a computationally low-cost estimation could simply be the location of the Action. The location could be used as a prediction of where the Resource would be after the Action and the estimated execution time would give an idea of when that would be. Then, by making a hypothetical Action selection at the new location, an estimate for the value of the best second Action could be derived. That, in turn, could be used to weight the values of the current Action selection. The process could continue deeper into the planning horizon, if the computing resources are available. One possibility is simply to initiate the Action selection before the running Action has been completed on the basis of the estimate of the final location of the Action. If the estimate was too far off, of course, a new selection could be made. Currently, the Action selection takes a fraction of a second (with tens of Actions on a single 3.00 GHz processor). But if

the number of Actions in the pool is very large, the time would be longer and this preemptive Action selection would be justified.

Currently, this research is funded by the Finnish Centre of Excellence in Generic Intelligent Machines Research (GIM). One of the main objectives of the GIM is the so-called Future Worksite concept, where intelligent machines try to take some menial and arduous work off the shoulders of the human workers. The worksite could be, for example, a construction site of a road or building, mining, tilling or clearing. The proposed method could be used to control the tasks of a machine in such a worksite.

Bibliography

[1] K. Fogarty, "Korea puts usd750m in robotics, aims to lead market by 2018," 2009, accessed 29.4.2010. [Online]. Available: http://www.roboticstrends.com/service_robotics/article/korea_to_put_750m_in_robotics_industry

[2] International Federation of Robotics IFR, "Executive summary," 2009, accessed 20.5.2010. [Online]. Available: http://www.ifr.org/uploads/media/2009_executive_summary_01.pdf

[3] M. Williams, "Panasonic has big plans for robots," Oct 2009, accessed 29.4.2010. [Online]. Available: http://www.pcworld.com/article/173788/panasonic_has_big_plans_for_robots.html

[4] C. Urmson, J. Anhalt, H. Bae, J. D. Bagnell, C. Baker, R. E. Bittner, T. Brown, M. N. Clark, M. Darms, D. Demitrish, J. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. M. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litkouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y.-W. Seo, S. Singh, J. M. Snider, J. C. Struble, A. T. Stentz, M. Taylor, W. R. L. Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar, "Autonomous driving in urban environments: Boss and the urban challenge," *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, vol. 25, no. 8, pp. 425–466, June 2008.

[5] Telemedicine & Advanced Technology Research Center, TATRC, "Home page, medical robotics," 2009, accessed 15.6.2010. [Online]. Available: http://www.tatrc.org/?p=ports/robotics/home

[6] U. Nehmzow and K. Walker, "Quantative description of robot-environment interaction using chaos theory," *Robotics and Autonomous Systems*, vol. 53, pp. 177–193, 2005.

[7] S. Terho, M. Heikkilä, T. Taipalus, J. Saarinen, and A. Halme, "A framework for graphical programming of skilled tasks with service robots," in *Proc. of The 9th Int. Conf. on Climbing and Walking Robots*, Brussels, 2006, p. 8.

[8] A. Halme, I. Leppänen, J. Suomela, S. Ylönen, and I. Kettunen, "Workpartner: Interactive human-like service robot for outdoor applications," *The*

International Journal of Robotics Research, vol. 22, no. 7-8, pp. 627–640, 2003. [Online]. Available: http://www.ijrr.org

[9] T. H. Collett, B. A. MacDonald, and B. P. Gerkey, "Player 2.0: Toward a practical robot programming framework," in *Proc. Australasian Conf. on Robotics and Automation (ACRA 2005)*, Würzburg, Germany, 2005, p. 8.

[10] T. Taipalus and K. Kosuge, "Development of service robot for fetching objects in home environment," in *Proc. IEEE Int. Symposium on Computational Intelligence in Robotics and Automation (CIRA2005)*, Espoo, Finland, 2005, p. 6.

[11] P. Harmo, T. Taipalus, J. Knuuttila, J. Vallet, and A. Halme, "Needs and solutions - home automation and service robots for the elderly and disabled," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems 2005*, 2005, p. 6, iEEE/RSJ International Conference on Intelligent Robots and Systems, Edmonton Canada 2005.

[12] M. Heikkilä, "Configuration of skilled tasks for execution in multipurpose and collaborative service robots," Ph.D. dissertation, Helsinki University of Technology, Espoo, Finland, 2009.

[13] I. Kauppi, "Intermediate language for mobile robots: A link between the high-level planner and low-level services in robots," Ph.D. dissertation, Helsinki University of Technology, 2003.

[14] T. Taipalus and A. Halme, "An action pool architecture for multi-tasking service robots with interdependent resources," in *Proceedings of the 8th IEEE international symposium on computational intelligence in robotics and automation 2009*, 2009, pp. 228–233, the 8th IEEE international symposium on computational intelligence in robotics and automation, Daejeon, Korea, Dec 15-18, 2009.

[15] G. Bannock, R. Baxter, and R. Rees, *The Penguin Dictionary of Economics*, 3rd ed. Middlesex, Great Britain: Penguin Books, 1984.

[16] Japanese Standards Association, *Service robot - Vocabulary*, JIS B 0187:2005 ed., Japan Robot Association, 2005.

[17] *RoboCup@Home Rules & Requlations*, RoboCup@Home league, 2010, 69M, Accessed 29.4.2010. [Online]. Available: http://www.robocupathome.org/documents/rulebook2010.pdf

[18] L. Takayama, W. Ju, and C. Nass, "Beyond dirty, dangerous and dull: what everyday people think robots should do," in *HRI '08: Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*. New York, NY, USA: ACM, 2008, pp. 25–32.

[19] E. Gat, R. P. Bonnasso, R. Murphy, and A. Press, "On three-layer architectures," in *Artificial Intelligence and Mobile Robots*. AAAI Press, 1998, pp. 195–210.

[20] G. McCain, H., R. Lumia, and JS Albus, "NASA/NBS standard reference model for telerobot control system architecture (NASREM)," Technical Note, 1989.

[21] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das, "The CLARAty architecture for robotic autonomy," in *2001 IEEE Aerospace Conference, Big Sky, MT*, 2001, pp. 121–132.

[22] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "Rt-component object model in rt-middleware- distributed component middleware for rt (robot technology)," in *Proc. IEEE Int. Symposium on Computational Intelligence in Robotics and Automation (CIRA2005)*, Espoo, Finland, 2005.

[23] J. Saarinen, A. Maula, R. Nissinen, H. Kukkonen, J. Suomela, and A. Halme, "Gimnet - infrastructure for distributed control of generic intelligent machines," in *Proc. The 13th IASTED Int. Conf. on Robotics and Applications Telematics*, Espoo, Finland, 2007.

[24] B. Gates, "A robot in every home," *Scientific American Magazine*, no. 1, p. 8, January 2007.

[25] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *IEEE International Conference on Robotics and Automation*, 2009, open-Source Software workshop.

[26] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.

[27] D. N. Malik Ghallab and P. Traverso, *Automated Planning Theory and Practice*. Morgan Kaufmann publishers / Elsevier, 2004.

[28] M. J. Matarić, *The Robotics Primer*. Cambridge, MA, USA: The MIT Press, 2007.

[29] L. Chung and J. do Prado Leite, "On non-functional requirements in software engineering," in *Conceptual Modeling: Foundations and Applications*, ser. Lecture Notes in Computer Science, A. Borgida, V. Chaudhri, P. Giorgini, and E. Yu, Eds. Springer Berlin / Heidelberg, 2009, vol. 5600, pp. 363–379.

[30] F. Ingrand, S. Lacroix, S. Lemai-Chenevier, and F. Py, "Decisional autonomy of planetary rovers: Research articles," *J. Field Robot.*, vol. 24, no. 7, pp. 559–580, 2007.

[31] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrad, "An architecture for autonomy," *The International Journal of Robotics Research*, vol. 17, no. 4, pp. 315–337, 1998.

[32] P. Maes, "How to do the right thing," *Connection Science Journal*, vol. 1, pp. 291–323, 1989.

[33] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, "Real-time knowledge-based systems," *AI Mag.*, vol. 9, no. 1, pp. 27–45, 1987.

[34] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1-3, pp. 14–23, 1986.

[35] M. S. Atkin, G. W. King, D. L. Westbrook, B. Heeringa, and P. R. Cohen, "Hierarchical agent control: a framework for defining agent behavior," in *Proc. of the fifth international conference on Autonomous agents*, Montreal, Quebec, Canada, 2001, pp. 425–432.

[36] E. Drumwright and V. Ng-Thow-Hing, "The task matrix: An extensible framework for creating versatile humanoid robots," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, Orlando, FL, USA, 2006.

[37] C. Parlitz, W. Baum, U. Reiser, and M. Hägele, *Human Interface and the Management of Information. Methods, Techniques and Tools in Information Design*. Springer Berlin / Heidelberg, 2007.

[38] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Cambridge, MA, USA: The MIT Press, 2003.

[39] S. Ylönen, "Modularity in service robotics," Ph.D. dissertation, Helsinki University of Technology, Espoo, Finland, 2006.

[40] Finnish standards association SFS, *Industrial Trucks*. Finnish standards association SFS, 1993.

[41] R. R. Murphy, *Introduction to AI Robotics*. Cambridge, MA, USA: The MIT Press, 2000.

[42] R. Arkin, *Behavior-based robotics*. Cambridge, MA, USA: The MIT Press, 1998.

[43] R. Sutton and A. Barto, *Reinforcement learning: An introduction*. Cambridge, MA, USA: The MIT press, 1998.

[44] R. Akerkar and P. Sajja, *Knowledge-Based Systems*. Jones & Bartlett Publishers, 2010.

[45] M. Nicolescu, "A framework for learning from demonstration, generalization and practice in human-robot domains," Ph.D. dissertation, University of Southern California, USA, 2003.

[46] S. Ekvall, "Robot task learning from human demonstration," Ph.D. dissertation, KTH School of Computer Science and Communication, Sweden, 2007.

[47] V. Verma, A. Jónsson, R. Simmons, T. Estlin, and R. Levinson, "Survey of command execution systems for NASA spacecraft and robots," in *Workshop at The International Conference on Automated Planning & Scheduling*, 2005, pp. 1–8.

[48] M. Amoretti and M. Reggiani, "Architectural paradigms for robotics applications," *Advanced Engineering Informatics*, vol. 24, no. 1, pp. 4–13, 2010.

[49] O. Pettersson, "Execution monitoring in robotics: A survey," *Robotics and Autonomous Systems*, vol. 53, no. 2, pp. 73–88, 2005.

[50] L. Pettersson, "Control system architectures for autonomous agents," 1997.

[51] G. Biggs and B. MacDonald, "A survey of robot programming systems," in *Proceedings of the Australasian conference on robotics and automation*. Citeseer, 2003.

[52] RoSTA, "Rosta, robot standards and reference architectures, home page," 2009, accessed 29.4.2010. [Online]. Available: http://www.robot-standards.org

[53] C. Kemp, A. Edsinger, and E. Torres-Jara, "Challenges for robot manipulation in human environments [grand challenges of robotics]," *Robotics & Automation Magazine*, vol. 14, pp. 20–29, 2007.

[54] M. Woof, "Technology for Underground Loading and Hauling Systems Offers Exciting Prospects," *Engineering and Mining Journal*, vol. 206, no. 3, 2005.

[55] T. Heikkilä, *A model-based approach to high-level robot control with visual guidance*. Espoo: VTT, 1990.

[56] R. Brooks, "Intelligence without representation," *Artificial intelligence*, vol. 47, no. 1-3, pp. 139–159, 1991.

[57] R. C. Arkin and D. C. Mackenzie, "Planning to Behave: A Hybrid Deliberative/Reactive Robot Control Architecture for Mobile Manipulation," in *International Symposium on Robotics and Manufacturing, Maui, HI*, 1994, pp. 5–12.

[58] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, "An architecture for autonomy," *The International Journal of Robotics Research*, vol. 17, no. 4, p. 315, 1998.

[59] R. J. Firby, "Modularity issues in reactive planning," in *In Proceedings of the Third International Conference on AI Planning Systems*. AAAI Press, 1996, pp. 78–85.

[60] P. Maes and R. Brooks, "Learning to coordinate behaviors," in *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990, pp. 796–802.

[61] D. C. Mackenzie, R. C. Arkin, and J. M. Cameron, "Multiagent mission specification and execution," *Autonomous Robots*, vol. 4, no. 1, pp. 29–52, March 1997.

[62] M. Nicolescu and M. J. Mataric, "A hierarchical architecture for behavior-based robots," in *Proc. First Int. Joint Conference on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, 2002.

[63] L. Zadeh, "Fuzzy sets," *Information and control*, vol. 8, no. 3, pp. 338–353, 1965.

[64] L. Padgham and M. Winikoff, *Developing intelligent agent systems: a practical guide*. Wiley, 2004.

[65] M. P. Georgeff and F. F. Ingrand, "Desicion making in an embedded reasoning systems," in *Int. Joint Conf. on Artificial Intelligence*, Detroit, Michigan, USA, 1989.

[66] W. Stallings, *Computer Organization and Architecture: Designing for Performance*. Prentice Hall, 2006.

[67] Apple Inc., "Grand Central Dispatch (GCD) brief," 2009, accessed 29.4.2010. [Online]. Available: http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090903.pdf

[68] S. Chen and M. Tseng, "Defining specifications for custom products: a multi-attribute negotiation approach," *CIRP Annals-Manufacturing Technology*, vol. 54, no. 1, pp. 159–162, 2005.

[69] *Staff Organization and Operations*, Department of the Army, Headquarters, 1997, field Manual, 101-5.

[70] Microsoft Developper Network, "Windows api," 2007, accessed 29.4.2010. [Online]. Available: http://msdn.microsoft.com/en-us/library/aa383750.aspx

[71] R. W. Scheifler and J. Gettys, *X Window system: the complete reference to Xlib, X protocol, ICCCM, XLFD*. Newton, MA, USA: Digital Press, 1990.

[72] S. Ekvall, D. Aarno, and D. Kragic, "Task learning using graphical programming and human demonstrations," in *IEEE International Symposium on Robot and Human Interactive Communication*, Hatfield, United Kingdom, 2006.

[73] D. T. Nguyen, D. Kim, B.-J. You, and S.-R. Oh, "Nbha - a distributed network-based humanoid software architecture," in *ICRA*. IEEE, 2006, pp. 456–461.

[74] M. Freed, "Managing multiple tasks in complex, dynamic environments," in *Proc. National Conference on Artificial Intelligence*, Madison, Wisconsin, USA, 1998, p. 7.

[75] K. Konolige, "Colbert: A language for reactive control in sapphira," in *KI-97: Advances in Artificial Intelligence*. Springer, 1997, pp. 31–52.

[76] OMG, "UML (Unified Modeling Language), Home page," accessed 27.9.2010. [Online]. Available: http://www.uml.org/

[77] R. Arkin, "Motor schema-based mobile robot navigation," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 1987, pp. 264–271.

[78] M. J. Mataric, "Integration of representation into goal-driven behavior-based robots," *IEEE Transactions on robotics and automation*, vol. 8, no. 3, pp. 304–312, 1992.

[79] J. Rosenblatt, "DAMN: A distributed architecture for mobile navigation," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 339–360, 1997.

[80] A. Farinelli, L. Iocchi, D. Nardi, and V. Ziparo, "Assignment of dynamically perceived tasks by token passing in multirobot systems," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1271–1288, 2006.

[81] F. F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots," in *In IEEE International Conference on Robotics and Automation, Mineapolis*, 1996, pp. 43–49.

[82] P. Elinas and J. J. Little, "Desicion theoretic task coordination for a visually-guided interactive mobile robot," in *Int. Conf. on Intelligent Robots and Systems (IROS 2007)*, San Diego CA, USA, 2007, pp. 4108–4114.

[83] S. Joyeux, R. Alami, S. Lacroix, and R. Philippsen, "A plan manager for multi-robot systems," *The International Journal of Robotics Research*, vol. 28, no. 2, p. 220, 2009.

[84] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of computer programming*, vol. 8, no. 3, pp. 231–274, 1987.

[85] D. Kragic, S. Ekvall, P. Jensfelt, and D. Aarno, "Sensor Integration and Task Planning for Mobile Manipulation," in *Workshop on "Issues and Approaches*

to Task Level Control",(D. Botturi and P. Fiorini, eds.), IEEE/RSJ International Conference on Intelligent Robots and Systems, D. Botturi and P. Fiorini, Eds. Sendai, Japan: IEEE/RSJ, 2004, p. 8.

[86] C. Arkin, Ronald and T. Balch, "AuRA: Principles and practice in review," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 175–189, 1997.

[87] Y. Matsusaka, H. Fujii, and I. Hara, "An extensible dialogue script for robot based on unification of state transition models," in *CIRA'09: Proc. of the 8th IEEE international conference on Computational intelligence in robotics and automation*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 586–591.

[88] M. P. Steffen Knoop and R. Dillmann, "Automatic robot programming from learned abstract task knowledge," in *Int. Conf. on Intelligent Robots and Systems (IROS 2007)*, San Diego CA, USA, 2007, pp. 1651–1657.

[89] K. Haigh and M. Veloso, "Planning, execution and learning in a robotic agent," in *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS'98)*, 1998, pp. 120–127.

[90] K. Erol, J. Hendler, and D. S. Nau, "HTN planning: Complexity and expressivity," in *In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94*. John Wiley & Sons Ltd, 1995, pp. 1123–1123.

[91] H. Kim and D. Kwon, "Task modeling for intelligent service robot using hierarchical task analysis," in *Proceedings of the 2004 FIRA Robot World Congress*, Busan, Korea, 2004, p. 6.

[92] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *in Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, Victoria, BC , Canada, 1998, pp. 1931–1937.

[93] R. Levinson, "A general programming language for unified planning and control," *Artificial Intelligence*, vol. 76, no. 1-2, pp. 319–375, 1995.

[94] S. Joyeux, R. Alami, and S. Lacroix, "A software component for simultaneous plan execution and adaptation," in *Int. Conf. on Intelligent Robots and Systems (IROS 2007)*, San Diego CA, USA, 2007, pp. 3038–3043.

[95] E. Uchibe and M. Asada, "Incremental coevolution with competitive and cooperative tasks in a multirobot environment," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1412–1424, July 2006.

[96] G. Johnson and R. Jennings, *LabVIEW graphical programming*. McGraw-Hill Professional, 2006.

[97] The MathWorks Inc., "Simulink - simulation and model-based design, datasheet," 2007, accessed 28.6.2010. [Online]. Available: http://www.mathworks.com/products/simulink/?BB=1

[98] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, "The saphira architecture: a design for autonomy," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 215–235, 1997.

[99] J. Tan and N. Xi, "Unified model approach for planning and control of mobile manipulators," in *IEEE International Conference on Robotics and Automation*, vol. 3. IEEE; 1999, 2001, pp. 3145–3152.

[100] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *Int. J. Rob. Res.*, vol. 26, no. 5, pp. 433–455, 2007.

[101] G. Franklin, J. Powell, A. Emami-Naeini, and J. Powell, *Feedback control of dynamic systems*. Addison-Wesley, 1994.

[102] S. Calinon, *Robot programming by demonstration: A probabilistic approach*. EPFL/CRC Press, 2009.

[103] F. Bause and P. Kritzinger, *Stochastic Petri Nets*. Vieweg, 2002.

[104] W. Chung, G. Kim, and M. Kim, "Development of the multi-functional indoor service robot psr systems," *Auton. Robots*, vol. 22, no. 1, pp. 1–17, 2007.

[105] V. Ziparo and L. Iocchi, "Petri net plans," in *Proceedings of Fourth International Workshop on Modelling of Objects, Components, and Agents (MOCA)*, 2006, pp. 267–290.

[106] International Electrotechnical Commission (IEC), *IEC 61131-3: Programmable controllers - Part 3: Programming languages*. Geneva, Switzerland: IEC, January 2003.

[107] K. Konolige, "Saphira robot control architecture," SRI International, Menlo Park, California, Tech. Rep., 2002.

[108] R. Bonasso, R. Firby, E. Gat, D. Kortenkamp, D. Miller, and M. Slack, "Experiences with an architecture for intelligent, reactive agents," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2, pp. 237–256, 1997.

[109] R. J. Firby, "Task networks for controlling continuous processes," in *In Proceedings of the Second International Conference on AI Planning Systems*, 1994, pp. 49–54.

[110] Web3D Consortium, "X3D, Home page," accessed 29.4.2010. [Online]. Available: http://www.web3d.org/x3d/

[111] The Panda3D Development Team, "Panda3d, home page," accessed 29.4.2010. [Online]. Available: http://www.panda3d.org/

[112] T. Taipalus, "Using remote controlled service robot for fetching objects in home environment," Master's thesis, Helsinki University of Technology, Espoo, Finland, 2004.

[113] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

[114] J. Saarinen, "A sensor-based personal navigation system and its application for incorporating humans into a human-robot team," Ph.D. dissertation, Helsinki University of Technology, Espoo, Finland, 2008. [Online]. Available: http://lib.tkk.fi/Diss/2009/isbn9789512299621/

[115] M. Beetz, T. Arbuckle, T. Belker, M. Bennewitz, W. Burgard, A. B. Cremers, D. Fox, H. Grosskreutz, D. Hahnel, and D. Schulz, "Integrated plan-based control of autonomous service robots in human environments," *IEEE intelligent systems*, vol. 16, no. 5, pp. 56–63, 2001.

[116] K. Konolige, "Colbert user manual," SRI International, Tech. Rep., 2001, v8.0a.

[117] A. Rao and M. Georgeff, "Bdi agents: From theory to practice," in *Proc. of the first international conference on multi-agent systems (ICMAS-95)*. San Francisco, CA, 1995, pp. 312–319.

[118] T. A. Estlin, D. Gaines, C. Chouinard, F. Fisher, R. Castano, M. Judd, and I. Nesnas, "Enabling autonomous rover science through dynamic planning and scheduling," DSpace at Jet Propulsion Laboratory [http://trs-new.jpl.nasa.gov/dspace-oai/request] (United States), Tech. Rep., 2007. [Online]. Available: http://hdl.handle.net/2014/40571

[119] M. Aydin and E. Öztemel, "Dynamic job-shop scheduling using reinforcement learning agents," *Robotics and Autonomous Systems*, vol. 33, no. 2-3, pp. 169–178, 2000.

[120] S. Lin, E. Goodman, and W. Punch, "A genetic algorithm approach to dynamic job shop scheduling problems," in *Proceedings of the Seventh International Conference on Genetic Algorithms*, 1997, pp. 481–488.

[121] A. Olivé, *Conceptual modeling of information systems*. Springer-Verlag New York Inc, 2007.

[122] F. Michaud, C. Côté, D. Létourneau, Y. Brosseau, J.-M. Valin, E. Beaudry, C. Raïevsky, A. Ponchon, P. Moisan, P. Lepage, Y. Morin, F. Gagnon, P. Giguère, M.-A. Roux, S. Caron, P. Frenette, and F. Kabanza, "Spartacus attending the 2005 aaai conference," *Auton. Robots*, vol. 22, no. 4, pp. 369–383, 2007.

[123] I. Nesnas, A. Wright, M. Bajracharya, R. Simmons, T. Estlin, and W. Kim, "Claraty: An architecture for reusable robotic software," in *SPIE Aerosense Conference*, 2003, p. 12.

[124] R. Arkin and D. MacKenzie, "Temporal coordination of perceptual algorithms for mobile robot navigation," *Robotics and Automation, IEEE Transactions on*, vol. 10, no. 3, pp. 276–286, 1994.

[125] R. Simmons, J. Fernandez, R. Goodwin, S. Koenig, and J. O'Sullivan, "Xavier: An autonomous mobile robot on the web," *Robotics and Automation Magazine*, 1999.

[126] R. Bindiganavale, W. Schuler, J. M. Allbeck, N. I. Badler, A. K. Joshi, and M. Palmer, "Dynamically altering agent behaviors using natural language instructions," in *In Autonomous Agents*. ACM Press, 2000, pp. 293–300.

Appendices

Appendix A

Description of an Object in the World Model

The table for an object in the mySQL database is described in Table A.2 and details of its "shape" parameter is given in Table A.1.

Table A.1: Description of the shape construction in the world model data base

shape	**size**	**shape_ratio**	**description**
		(shape_ratio*size)	
cube	length of side	-	
box	along Z-axis	along X- and Y-axis	
ball	radius	-	
cone	height along Z-axis	radius of base	place in middle of center axis
plane	along X-axis	along Y-axis	normal or "front" plane points along Z-axis
cylinder	height along Z-axis	radius of base	place in middle of center axis

Table A.2: Description of object attributes in world model data base

Feature name	Data type	MySQL data type	Value range	Default value	Description
id	integer	int	unsigned 32 bit	i+1	specifies unique id for an object. Reserved (0=world, 1=robot)
name	string	char(10)	...	“object” + id	gives a human readable and associative name for the object, not unique
node	integer	int	unsigned 32 bit	1(robot)	tells the data base id of the node that object belongs to
placeX	double	float	[m]	0.0	(X,Y,Z) of the object in “node's” co-ordinate system
placeY	double	float	[m]	0.0	(X,Y,Z) of the object in “node's” co-ordinate system
placeZ	double	float	[m]	0.0	(X,Y,Z) of the object in “node's” co-ordinate system
placeA	double	float	[rad]	0.0	X -axis euler angle of object in (X, Y, Z) order in “node's” co-ordinate frame
placeB	double	float	[rad]	0.0	Y -axis euler angle of object in (X, Y, Z) order in “node's” co-ordinate frame
placeG	double	float	[rad]	0.0	Z -axis euler angle of object in (X, Y, Z) order in “node's” co-ordinate frame
placeX_var	double	float	[m]	99.9	first moment of gaussian distribution of place
placeY_var	double	float	[m]	99.9	
placeZ_var	double	float	[m]	99.9	
placeA_var	double	float	[rad]	99.9	first moment of gaussian distribution of orientation
placeB_var	double	float	[rad]	99.9	
placeG_var	double	float	[rad]	99.9	
placeXY_var	double	float	[m]	99.9	first moment of gaussian distribution of place
placeXZ_var	double	float	[m]	99.9	
placeYZ_var	double	float	[m]	99.9	
shape	string	char(10)	box, cylinder, ball, cone, ...	“ball”	basic shape of bounding volume of the object, used in visualization
shape_ratio	double	float	[m]	0.00	shapes's attribute (like line width) would be described later
size	double	float	[m]	0.5	size of the shape (lentgh of line, side of cube, radius of a ball, hight/length of cone)
size_var	double	float	[m]	99.9	first moment of gaussian distribution of size
weight	double	float	[kg]	1	weight of the object
weight_var	double	float	[kg]	99.9	
colour	double	float	0.0...1.0	0.5	Hue (in HVS)
colour_var	double	float		1	
speedX	double	float	[m/s]	0	velocity along X-axis
speedY	double	float	[m/s]	0	velocity along Y-axis
speedG	double	float	[rad/s]	0	angular velocity along Z-axis
time	double	double	[s]	time()	last time when evidence got from object, (seconds since 1970-01-01 00:00:00)
association1	string	char(10)	object names in database		objects associated to this object
association2	string	char(10)	object names in database		objects associated to this object
association3	string	char(10)	object names in database		objects associated to this object
association4	string	char(10)	object names in database		objects associated to this object
association5	string	char(10)	object names in database		objects associated to this object
pic_front	jpg	blob	less than 65kb	defaultF.jpg	sample image of the object (can be used as a texture in visualization)
pic_back	jpg	blob	less than 65kb	defaultB.jpg	
existence	float	float	0.0...1.0	0,95	Probability of existence

Appendix B

Example of XML-Listing of Action

```
<Action name="Template" objectID="1" id="123" taskID="123"
priority="0.345" location="0.5, 0.3, 0.4" sigma="0.1, 0.2, 0.8"
lambda="0.08" time="123.4, 40.5" description="Does nothing"
version="0.1">

  <ELstart>
    <ADD taskID="" actionID="" EL_ID="1" comment= "IDs will get actions
    IDs, 0s will be independent">
      <ELheader threshold="0.9" objectID="" exeFilename="test_EL.py"
      description="randomly cause an event">
        <RESPONSE command="addAction"
        filename="Sleep.xml" pool="PS" actionID="999"/>
      </ELheader>
    </ADD>
  </ELstart>
  <ELfinal>
    <ADD taskID="0" actionID="0" EL_ID="2" filename="ELtest.xml"/>
    <RM taskID="" actionID="" EL_ID="1"/>
  </ELfinal>
  <Plan>
  <Parameter type="String" value="Plan param" name="var_Name1"
  description="Pseudo variable for plan"/>
  <Parameter type="Integer" value="1" name="var_Name2"
  description="Pseudo variable for plan"/>
  <ReturnValue type="Unknown" value="" name="RETURNVALUE0"
  description=""/>
  <Block uiBlockType="Action" uiPosX="0" uiPosY="0" InitText=""
  ExeText="" EndText="" ErrorText=""/>

  <SingleTask name="STARTSTATE" category="FlowControl" id="0"
  priority="" description="">
    <Block uiBlockType="Start" uiPosX="-346" uiPosY="-861" InitText=""
    ExeText="" EndText="" ErrorText=""/>
  </SingleTask>

  <SingleTask name="PS_uTaskTemplate" category="PS" id="1"
  priority="NORMAL" description="Template to help writing micro tasks">
    <Parameter type="String" value="var_Name1" name="varName1"
```

```
    description="Pseudo variable 1"/>
    <Parameter type="Integer" value="12" name="varName2"
    description="Pseudo variable 2"/>
    <Parameter type="Double" value="1.2" name="varName3"
    description="Pseudo variable 3"/>
    <ReturnValue type="Integer" value="var_Name2" name="retName1"
    description="return value from micro task"/>
    <Block uiBlockType="MicroTask" uiPosX="-346" uiPosY="-911"
    InitText="" ExeText="" EndText="" ErrorText=""/>
  </SingleTask>

  <SingleTask name="PS_uTaskSleep" category="PS" id="2" priority="NORMAL"
  description="non busy Sleep/wait for given time, good for cooling down CPU">
    <Parameter type="Float" value="5.0" name="sleepTime"
    description="time to sleep [s]"/>
    <Block uiBlockType="MicroTask" uiPosX="-346" uiPosY="-951" InitText=""
    ExeText="" EndText="" ErrorText=""/>
  </SingleTask>

  <SingleTask name="IF" category="FlowControl" id="4" priority="Normal"
  description="STARTING IF">
    <Parameter type="Integer" value="var_Name2" name="IFINPUT"
    description=""/>
    <Block uiBlockType="If" uiPosX="-164" uiPosY="251" InitText=""
    ExeText="" EndText="" ErrorText=""/>
  </SingleTask>

  <SingleTask name="FINALSTATE" category="FlowControl" id="5"
  priority="" description="">
    <Block uiBlockType="Final" uiPosX="-346" uiPosY="-861" InitText=""
    ExeText="" EndText="" ErrorText=""/>
  </SingleTask>

  <Connections>
    <Connect From="0" FromPos="3" To="1" ToPos="1"/>
    <Connect From="1" FromPos="3" To="2" ToPos="1"/>
    <Connect From="2" FromPos="3" To="4" ToPos="1"/>
    <Connect From="4" FromPos="2" To="5" ToPos="1"/>
    <Connect From="4" FromPos="3" To="2" ToPos="2"/>
  </Connections>
  </Plan>
</Action>
```

Appendix C

Features and their Terms in Different Approaches

Table C.1: Comparison of terms for common features in different approaches (continues ...)

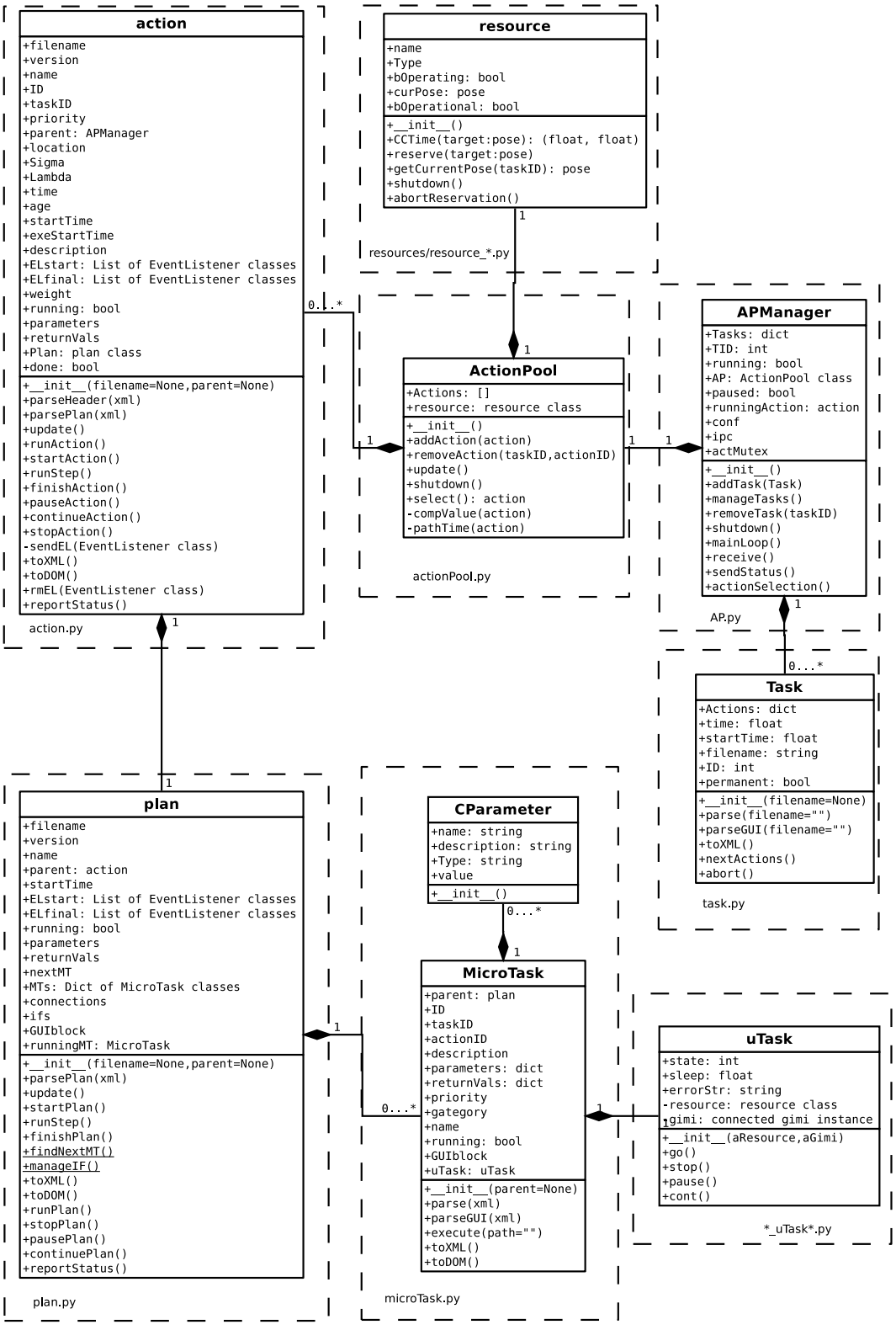
Proposed	**Olivé** [121]	**Hierarchical behaviors** [62]	**OpenPRS** [30]	**MBA** [122]	**PRS** [65]	**BDI** [117]	**CLARAty** [21, 123]
Task	-	-	-	Motivational modules	Knowledge space	Desires	Goal
Action		-	-	Intention or system konw-how	-	Intentions	Task
Plan	-	Behavior network	Achieve goal	Plan	-	-	-
uTask	-	Behavior	-	-	-	-	(Command)
EL	Informative function		Preserve, wait or maintain goal	Emotional module	-	-	-
PTU pool	Memory function	-	Test goal	-	Intention structure	-	Resource usage predictions, State
Realtime layer	-	Abstract behaviors	Functional level	-	-	-	Functional layer
Pose pool	Active function	-	Task graph	Dynamic task workspace or system konw-how	Intention structure	-	Resource usage predictions, State
World model	-	-	Test, assert and retrect goals	-	Knowledge space	Belivies	-
Perception agent	Memory function	-	test goal	-	Monitor	-	-

(... continues) Comparison of terms for common features in different approaches

Proposed	**AuRA** [86, 61, 124]	**NASREM** [20]	**Task matrix** [36]	**TCA** [92, 125]	**Saphira** [98, 107, 75]	**PAR** [126]	**3T** [108]
Task	FSA	Level 5	Macro Task	(task tree)	-	IPAR	-
Action	(assemblage)	Level 4	-	(Action)	Activity schema	IPAR	(RAP)
Plan	FSA	-	Task program	-	Colbert script	Execution steps	reactive and sequencing tier
uTask	assemblage	Level 3	-	GOAL or COMMAND task	(Activity Schema)	-	skill
EL	(trigger agent)	-	In condition	monitor task or exception handling	-	-	event monitor
Realtime layer	motor and perceptual schemas	-	-	behavior layer	state reflector	-	skill level
Pose pool	-	(Level 6)	-	-	-	Agent process	sequencing tier
World model	-	Global memory	-	-	Local Perceptual Space and Global Map Space	Database	-
Perception agent	(perceptual schema)	-	-	-	Sensor Interpretation Routine	-	-

Appendix D

Class Diagram of the Action Pool Implementation


action
+filename
+version
+name
+ID
+taskID
+priority
+parent: APManager
+location
+Sigma
+Lambda
+time
+age
+startTime
+exeStartTime
+description
+ELstart: List of EventListener classes
+ELfinal: List of EventListener classes
+weight
+running: bool
+parameters
+returnVals
+Plan: plan class
+done: bool
+__init__(filename=None,parent=None)
+parseHeader(xml)
+parsePlan(xml)
+update()
+runAction()
+startAction()
+runStep()
+finishAction()
+pauseAction()
+continueAction()
+stopAction()
-sendEL(EventListener class)
+toXML()
+toDOM()
+rmEL(EventListener class)
+reportStatus()
action.py
resource
+name
+Type
+bOperating: bool
+curPose: pose
+bOperational: bool
+__init__()
+CCTime(target:pose): (float, float)
+reserve(target:pose)
+getCurrentPose(taskID): pose
+shutdown()
+abortReservation()
resources/resource_*.py
ActionPool
+Actions: []
+resource: resource class
+__init__()
+addAction(action)
+removeAction(taskID,actionID)
+update()
+shutdown()
+select(): action
-compValue(action)
-pathTime(action)
actionPool.py
APManager
+Tasks: dict
+TID: int
+running: bool
+AP: ActionPool class
+paused: bool
+runningAction: action
+conf
+ipc
+actMutex
+__init__()
+addTask(Task)
+manageTasks()
+removeTask(taskID)
+shutdown()
+mainLoop()
+receive()
+sendStatus()
+actionSelection()
AP.py
Task
+Actions: dict
+time: float
+startTime: float
+filename: string
+ID: int
+permanent: bool
+__init__(filename=None)
+parse(filename="")
+parseGUI(filename="")
+toXML()
+nextActions()
+abort()
task.py
plan
+filename
+version
+name
+parent: action
+startTime
+ELstart: List of EventListener classes
+ELfinal: List of EventListener classes
+running: bool
+parameters
+returnVals
+nextMT
+MTs: Dict of MicroTask classes
+connections
+ifs
+GUIblock
+runningMT: MicroTask
+__init__(filename=None,parent=None)
+parsePlan(xml)
+update()
+startPlan()
+runStep()
+finishPlan()
+findNextMT()
+manageIF()
+toXML()
+toDOM()
+runPlan()
+stopPlan()
+pausePlan()
+continuePlan()
+reportStatus()
plan.py
CParameter
+name: string
+description: string
+Type: string
+value
+__init__()
MicroTask
+parent: plan
+ID
+taskID
+actionID
+description
+parameters: dict
+returnVals: dict
+priority
+gategory
+name
+running: bool
+GUIblock
+uTask: uTask
+__init__(parent=None)
+parse(xml)
+parseGUI(xml)
+execute(path="")
+toXML()
+toDOM()
microTask.py
uTask
+state: int
+sleep: float
+errorStr: string
-resource: resource class
-gimi: connected gimi instance
+__init__(aResource,aGimi)
+go()
+stop()
+pause()
+cont()
_uTask.py
0..*
1
0...*