KTH Numerical Analysis
and Computer Science

A Component Framework for Autonomous Mobile Robots

ANDERS OREBÄCK

Doctoral Thesis
Stockholm, Sweden 2004

TRITA-NA-0429
ISSN-0348-2952
ISRN-KTH/NA/R–04/29–SE
ISBN 91-7283-892-2

KTH Numerisk analys och datalogi
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen fredagen den 19 november 2004 i Kollegiesalen, Administrationsbyggnaden, Kungl Tekniska högskolan, Valhallavägen 79, Stockholm.

Tryck: Universitetsservice US AB

Abstract

The major problem of robotics research today is that there is a barrier to entry into robotics research. Robot system software is complex and a researcher that wishes to concentrate on one particular problem often needs to learn about details, dependencies and intricacies of the complete system. This is because a robot system needs several different modules that need to communicate and execute in parallel.

Today there is not much controlled comparisons of algorithms and solutions for a given task, which is the standard scientific method of other sciences. There is also very little sharing between groups and projects, requiring code to be written from scratch over and over again.

This thesis proposes a general framework for robotics. By examining successful systems and architectures of past and present, yields a number of key properties. Some of these are ease of use, modularity, portability and efficiency. Even though there is much consensus on that the hybrid deliberate/reactive is the best architectural model that the community has produced so far, a framework should not stipulate a specific architecture. Instead the framework should enable the building of different architectures. Such a scheme implies that the modules are seen as common peers and not divided into clients and servers or forced into a set layering.

Using a standardized middleware such as CORBA, efficient communication can be carried out between different platforms and languages. Middleware also provides network transparency which is valuable in distributed systems.

Component-based Software Engineering (CBSE) is an approach that could solve many of the aforementioned problems. It enforces modularity which helps to manage complexity. Components can be developed in isolation, since algorithms are encapsulated in components where only the interfaces need to be known by other users. A complete system can be created by assembling components from different sources. Comparisons and sharing can greatly benefit from CBSE.

A component-based framework called ORCA has been implemented with the following characteristics. All communication is carried out be either of three communication patterns, *query*, *send* and *push*. Communication is done using CORBA, although most of the CORBA code is hidden for the developer and can in the future be replaced by other mechanisms. Objects are transported between components in the form of the CORBA valuetype. A component model is specified that among other things include support for a state-machine. This also handles initialization and sets up communication. Configuration is achieved by the presence of an XML-file per component. A hardware abstraction scheme is specified that basically route the communication patterns right down to the hardware level.

The framework has been verified by the implementation of a number of working systems.

Keywords: robotics, mobile robots, autonomous robots, component-based software engineering, software architectures.

Acknowledgements

Research is never work in isolation. A good research environment is a melting pot of ideas and notions that are contributed by all members of the group. This is certainly the case in an open and friendly atmosphere such as the one at CVAP/CAS, where you are always welcome into peoples rooms for a discussion. I would like to thank all present and former members of CVAP/CAS, but here is only room to name the most important ones.

Thank you

Henrik, for providing excellent guidance and support. You never mind being asked about anything from soldering cables to the designing of expert systems.
Joe, for some reason you always believed in me and I owe you very much.
Patric, because without you, where would we be now? The most reliable and hard working man I know. You even became a good programmer!
Mattias, for being a good friend, and a clever guy who has deep knowledge about all the stuff that matters.
Peter, for always giving me great support and friendship.
Daniel, for interesting discussions at lunches and elsewhere.
Dani, for being a breath of fresh air!
LarsP, getting things done and make us laugh at the same time.
Lars, for always having an open door policy.
Stefan, for being an interesting and friendly person.
Magnus, for being a good manager and a nice person.

I would also like to mention (without any particular order) Mattias B, John, Morten, Maria, Mårten, David, Magnus E, Martin, Hedvig, Dennis, Olle, Tony, Eric, Ambjörn, Josephine, Ivan, Mikael, Johan, Philipp, Fredrik, Carsten, Jorge, Pär, Atsuto, Kourosh, Matti, Danny, Uwe, Tomas, Herman, Harald, Marco, Guido and many more.

My deepest gratitude is also extended towards my family and of course to my beloved wife **Malin**. Without your support this would never have happened!

This research has been sponsored by the Swedish Foundation for Strategic Research through the Centre for Autonomous Systems. The funding is gratefully acknowledged.

Contents

List of Figures

Chapter 1

Introduction

Robots come in many shapes and sizes as the term has a quite broad meaning. It can be an android, a manipulator arm, a mechanical cockroach, an airplane or a cart.

What people associate to when hearing the word robot differs significantly but is often influenced by literature and film. Here the theme is usually dystopic where artificial super-intelligent beings threaten to take over the world. The conception of the general public of what the robots of today actually can do, is usually quite far from the truth. Those with knowledge of the field know that we at best have mastered to create machines with intelligence on par with insects.

The term *robot* comes from robota which is a Czech word meaning forced labor. Unfortunately for us Swedish roboticists, the word robot in Swedish is also used for military missiles. Automated software programs that crawl the Internet are also called robots. This thesis however only deals with robots that have physical instantiation.

This introductory chapter starts by providing a short history of robotics and then demonstrates areas in which robots are in use. Then we define some robot characteristics followed by an account of the problems facing roboticists today. The introduction ends with detailing the contributions and an outline of the thesis.

1.1 History

There has been academic research within the field of robotics for some decades, but the underlying ideas are much older.

Robots in the Literature

The idea of artificial persons dates back at least to ancient Greek mythology, e.g. in *Pygmalion* where the statue of Galatea came to life. There is also the *golem* from Jewish legends, a clay monster created by magic. Leonardo *da Vinci* designed drawings of a mechanical knight, probably based on his anatomical research. As the technology advanced in the eighteenth and nineteenth centuries, several mechanical creatures were constructed. In 1738 Jacques *de Vaucanson* created an android that played the flute, as well as a mechanical duck that reportedly ate and defecated.

The book *Frankenstein*, written in 1818 by Mary Shelley, is analogous to the theme of robots replacing their human creators. This is further expressed in the classic movie *Metropolis* (1927). Similar conceptions are found in horror movies from the fifties but also in more recent movies such as *Blade Runner* (1982) and *The Terminator* (1984). In *Star Wars* (1977), the image of robots was much more positive.
The science fiction writer Isaac Asimov, created the Three Laws of Robotics (Asimov 1950):

1. A robot may not harm a human being, or, through inaction, allow a human being to come to harm.

2. A robot must obey the orders given to it by the human beings, except where such orders would conflict with the First Law.

3. A robot must protect its own existence, as long as such protection does not conflict the First or Second Law.

Modern History

Following is a collection of sample milestones in the modern history of mobile robotics.

1953	A robotic tortoise was created from the ideas of Norbert Wiener, who founded the field of *cybernetics*, a combination of information science, control theory and biology.
1956	Joseph Engelberger together with George Deroe formed the first robot company, called Unimate.
1961	Unimate ships the first commercial robot.
1968	Shakey was constructed at Stanford Research Institute (SRI). It featured a television camera and bumpers as sensors.
1970	The Swedish company ASEA formed a industrial robotics division (today ABB Robotics) and delivered their first unit in 1974.
1973	KUKA developed an industrial robot.
1975	HILARE was a robot project at LAAS in Toulouse, France. To perceive the world it used a video camera, laser range finder and ultrasonic sensors.
1977	The Stanford Cart used stereo vision in order to navigate. Since the vision processing was slow, the robot only moved about four meters per hour.
1984	Waseda University in Japan presented a piano-playing humanoid robot.
1990	Autonomous highway driving at high speed was demonstrated at Carnegie Mellon Institute in 1990 with the vehicle Navlab 5, a converted Pontiac Trans Sport.
1994	Dickmann's group in Munich demonstrated autonomous driving through heavy traffic around Paris.
1995	In a project called *No Hands Across America*, Navlab 5 was used to drive most of the way from Pittsburgh to San Diego.
1996	The Swedish company Husqvarna introduced a robotic lawn mower.
1997	The rover Sojourner operated on the surface of Mars.
1998	HONDA presented the first in a range of very sophisticated humanoids.
1999	SONY launched their well known AIBO dog, an entertainment robot.
2000	Electrolux shipped the first robotic Trilobite vacuum cleaner.

1.2 Uses of Robots

There are many uses of robots today, and more are anticipated for the future. Here is a list of some areas.

Industrial manufacturing Industrial robots gained wide use in manufacturing plants during the 1970s. Especially the automotive industry has replaced many human laborers with robots. The typical industrial robot is programmed to carry out one specific task, such as welding or painting.

Mining Robotic vehicles are deployed within the mining industry, e.g. in Australia and Sweden.

Unmanned aerial vehicles (UAV) Flying vehicles are primarily in use for military reconnaissance. The Predator, an Unmanned Aerial Vehicle (UAV) was deployed in the 1999 Kosovo air campaign as well as over Afghanistan from 2001.

Underwater robots Underwater robots are used for e.g. in cable inspection and repairing.

Automated cars Tremendous research has gone into developing automated systems for ordinary cars so that they can stay on the road and avoid colliding with other vehicles. These systems are mainly designed for highways and perform very well. A concept called platooning means that cars drive very closely spaced in order to provide space for more vehicles on the road. This technology is mature and safe, but the deployment is hindered by human attitude and the unsolved insurance situation.

Service robots A service robot is a system that services humans where the cost/performance ratio is beneficial. Examples are the automated lawn mower and vacuum cleaner. A sophisticated type of service robot is the personal assistant robot. These are not common today but are expected to have a great future. This is motivated by an aging population in the West and Japan. Robots could here aid the elderly with household tasks such as cleaning, cooking, and even feeding. In hospitals, food-plates and medicines can be delivered to the rooms, and in office environments mail can be delivered to the staff. Pyxis has delivered 120 Helpmate robotic couriers to health care facilities. Another area of service robotics is acting as tour-guides in museums and similar places. For a vision for service robotics, see (Engelberger 1989).

Bomb and mine disarming, hazardous environments Tele-operated robots are in widespread use at tasks where a human would be at risk. For example the police use them for disarming bombs.

Space missions The use of robots in space exploration is rather well-known to the general public after the successful missions to Mars. First the Sojourner (Figure 1.1) in 1997, then the Spirit and Opportunity in 2004. There are other uses of robots in space, e.g. in unloading of cargo on-board space shuttles.

A recent topic for research involves having multiple robots cooperating in carrying out tasks. This however has not yet reached practical uses but will undoubtedly do in the future.

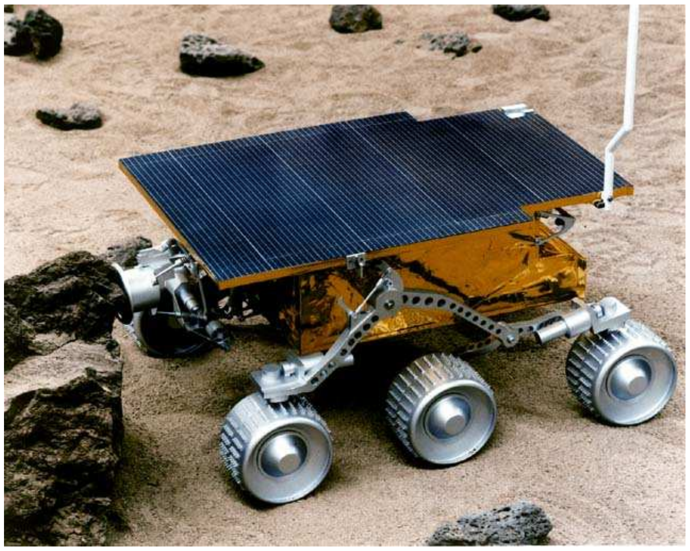


Figure 1.1: The Sojourner rover.

1.3 Characteristics of Robotics

Robotics is a multi-faceted field covering many different aspects. Here we take a look at three of the these.

Mobility

Real robots always have moving parts. With mobile robots we mean robots that can locomote, i.e. move in its entirety in space. On the ordinary ground this usually means the robot has wheels, legs or tracks. If the robot acts in the air or underwater, surely other methods of transport are used.

Industrial robots are almost always stationary, although they usually have moving arms (manipulators). This type of robot does then not qualify as a mobile robot. For robots that perform fetch-and-carry tasks in an indoor setting, mobility is of course a key property. Likewise, robots that are used outdoors generally needs mobility.

Autonomy

Another degree of freedom is autonomy which describes how independent from humans a robot can operate. At one end is a robot that is fully controlled by a human operator, such as a tele-operated bomb disarming robot. At the other extreme, the robot is totally autonomous. Most would argue that we have not yet achieved this. We have however seen autonomous driving in rush hour traffic (Maurer et al. 1995) and autonomous flight over extended distances. But the robustness to different kinds of disturbances is still limited, in the sense that they have a hard time coping if 25% of the sensors fail, or when mechanical failures occurs.

The notion of autonomy is tied to the level of disturbances that can be handled in an "intelligent" fashion. To be autonomous, the robot needs to be able to adapt reasonably well to unexpected changes in the environment. The primary motivation for the industrial robots in wide use today, is it high degree of repeatability that ensure a homogeneous quality. Such systems rely on no or minimum sensing, and as such require engineering of the environment for particular tasks. As such the system is not termed autonomous. In order to achieve a level of autonomy, awareness of the world surrounding is necessary. Robots use sensors such as sonars and cameras to obtain this. Dealing with sensors therefore forms a major part for anyone working with autonomous robots.

A high degree of autonomy demands that reasoning and planning are part of the system. The word intelligence is sometimes used in this area, but is not really a good term since intelligence is mostly in the eye of the beholder. A robot could look very smart but actually have a very simple design and vice versa. For example, by adding speech synthesis and recognition, a robot will be perceived as much smarter.

On a side note, most roboticists would not agree that they are working within the field of artificial intelligence (AI), as the fields are fairly decoupled nowadays. None the less, techniques from AI are often used when developing autonomous robots. Examples are artificial neural networks, fuzzy logic and different methods for learning and planning.

Versatility

Robots are often built to perform one specific task. Specialized robots are easier to construct and can be made more robust. Again, industrial robots serve to provide an example of this kind, but also the automated vacuum cleaner as it performs only one function.

Other robots, such as a personal assistant robot, or *robotic butlers*, should in contrast be able to carry out a wide range of tasks. This puts higher demands on both the hardware and software. Robustness is also much harder to obtain in this kind of robot. Researchers building anthropomorphic robots (humanoids aka androids) even have higher goals, as they strive to mimic the capabilities of human beings.

This thesis is primarily concerned with robotic systems that are mobile, autonomous, and versatile.

1.4 Problems of Modern Robotics

Robotics is also a multi-disciplined field. In encompasses control-theory, computer science, computer programming and mathematics. Also electronics, mechanics and mechatronics are necessary to build and maintain the hardware platforms.

This thesis deals for the most part with the computer science part of robotics. However both hardware and software related problems are discussed in this section.

Hardware

Robotics was initially part of several fields. The most spectacular robots were probably derived from artificial intelligence, but efforts have also been made in mechanical engineering, control engineering, sensory systems, and vehicles technology, to mention a few. It was only recently that it was realized that there is a need to integrate all of these disciplines to facilitate design of systems.

Most researchers coming from artificial intelligence experienced that making the leap from theory to practice was more difficult than anticipated. Real robots are hard to build but also very hard to maintain - just ask any robotics grad student. Robot labs around the world are filled with robots that no longer works. Also robots that are bought off the the shelf cease to function as the vendors go out of business, or to which parts can no longer be obtained. Constructing new hardware is also very costly. Most labs cannot afford to make new designs, so the evolution of hardware is quite slow.

The situation today is however much better than it used to be. A decade ago, researchers had to build their own hardware, but today robots can be readily purchased off the shelf. The cost decline of computing power and data storage has also been beneficial. The area of sensing has also improved, e.g. cameras and laser-rangefinders have become smaller and cheaper.

One specific problem area of robot hardware is that of batteries. Power consumption is often high and batteries usually do not last longer than a couple of hours. Batteries are also very heavy which reduces the utility payload. There are example of robots that can dock to a power source and recharge, but this severely limits the range and versatility.

Figure 1.2 shows a SONY AIBO with hardware parts marked out, and Figure 1.3 shows the same robot with the cover plating removed. These images give a hint at how complicated the mechanical structure of robots usually need to be.

Software

This section pinpoints the most important problems of robotics today, problems that this thesis aims to address.

Barrier to Entry

Autonomous robotic systems are what is called *complex systems*. They need to have a number of competences in order to function. Examples of these are:

- locomotion
- navigation
- sensory data interpretation
- localization
- planning

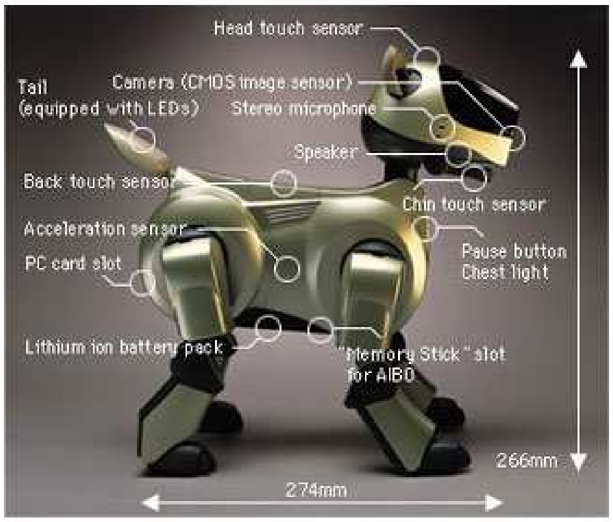


Figure 1.2: External view of the SONY AIBO.

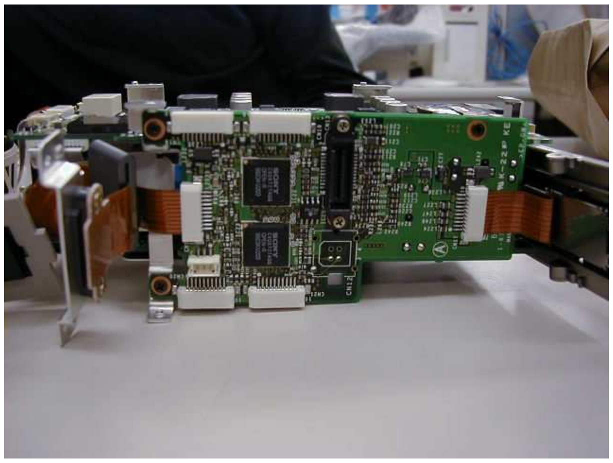


Figure 1.3: The SONY AIBO with removed cover plating.

- interaction with humans
- obstacle avoidance

Every competence requires specific domain knowledge from the developer and are research topics in themselves. Many researchers specialize in one or more of these topics, which usually involves development of algorithms. However, in order to test a competence on a real robot, a complete system is needed involving most of the above mentioned. Many of these are required to run in parallel and need to communicate both synchronously and asynchronously. This puts a tremendous burden on the developer if he or she has to build everything from scratch. Even for a robotics laboratory involving several people, the effort is still considerable to construct a complete and robust working system. The barrier to entry into robotics research is thus high. If the manpower has not been available to furnish a complete software system, pure simulation has often been the only alternative.

Comparison and Verification

Most other disciplines pride themselves by following the standard scientific method of comparing different solutions and algorithms in a controlled way. This has been very difficult in the robotics community since basically every laboratory has been running their own software system. These systems are not compatible, so a program implementing an algorithm can not readily be executed at another site. So comparing different algorithms that try to solve the same problem has been very difficult. From this follows also that verifying a solution in different environments and under different circumstances has been hard.

Software Re-use

Another area which also needs to be mastered, apart from the specific domain knowledge, is computer science. Domain expertise in this area is often overlooked, and cannot be expected from everyone. Too seldom however, are computer scientists called in to assist. The results are often inferior solutions that are hard to re-use and maintain.

As stated above, the situation today is that most researchers have to write most of the code they need themselves. Code cannot be shared since each piece of software have different dependencies. Even re-use between platforms and software generations at the same laboratory is quite rare.

Re-use by adapting source-code is possible but laborsome and error-prone. Only if the shared programs are in a binary format can effective re-use and sharing be possible. This would also open up for a market of re-usable software.

Conclusion

The overall conclusion is that the major problems for robotics today lie not in the hardware but on the software side. There is however no shortage of well functioning and robust algorithms developed by competent researchers. The field has matured and most needed tasks are today understood and have good solutions. No, the biggest problem is the lack of a standardized framework in which these researchers can plug their work without having to hassle with too much complexity.

These modules need to be put in an elaborate system that prescribes the interfaces, communication methods and limitations that the modules must adhere to. The system should also provide services so that the individual modules can be made as coherent and simple as possible. A long list of requirements can be made for a system. Here are some of the most important properties that can be required:

- modularity
- ease of use
- hardware portability
- run-time efficiency
- extendability and scalability
- robustness
- run time flexibility

To create this kind of system and to perform this integration of competences is exactly what this thesis is all about.

1.5 Contributions

The contributions of this thesis regard the structural principles for software architectures for mobile autonomous robots. A first iteration was the ISR (BERRA) architecture. This system was an early example of a standardized client-server topology exhibiting network transparency. It was designed and implemented in cooperation with Mattias Lindström, with an equal share.

This system became a very successful demonstrator with a large set of capabilities. Several abilities were also added by short-term guest researchers, proving its extendability and relative ease of use. It has been in operation on several platforms for several years at the Centre of Autonomous Systems at KTH and also at other institutions.
ISR/BERRA is described in the following publications.

- Andersson, M., Oreback, A., Lindstrom, M. & Christensen, H. (1999), Intelligent Sensor Based Robotics, Springer Verlag, Heidelberg, chapter **ISR: An Intelligent Service Robot**.
- Lindstrom, M., Oreback, A., & Christensen, H.I.: **BERRA: A research architecture for service robots**, Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on ,Volume: 4 , 24-28 April 2000 Pages:3278 - 3283 vol.4

Next the emerging science of Component Based Software Engineering (CBSE) was studied and the implications for robotics were addressed. The conclusions were that CBSE could simplify:

- exchange of software parts between labs, allowing specialists to focus on their particular field.
- comparison of different solutions.
- startup in robot research.

The work was presented in the following paper:

- Oreback, Anders, **Components in Intelligent Robotics** Component-Based Software Engineering: State of the Art, Mälardalen University, 2000

An extensive comparative study was later performed, where several existing software systems were evaluated. This was done by implementing the same capabilities on one platform using the different software. A number of key principles and characteristics could be defined from this study which is published in the following journal article.

- Anders Oreback & Henrik I. Christensen: **Evaluation of Architectures for Mobile Robotics**, Autonomous Robots, Volume 14, Issue 1, January 2003, Pages 33 - 49

The second major design for a new system was within the EU sponsored project OROCOS. The lessons learned from the previous work resulted in a component-based framework. The system is peer-based with standardized communication but with very little restrictions elsewhere in the system. The rationale was that many components cannot be defined as either a client or a server. A strict layering is also abandoned for the same reasons. Communication patterns play en integral role. Furthermore, a hardware abstraction scheme is included allowing for easy extension of supported hardware. This system has been deployed in a number of projects around the world. It has been renamed to ORCA, is available at Sourceforge and has an active user base.

This work has not been published elsewhere except for in this thesis and in reports in connection with the evaluation for EU. However one application using this system has been described in the following conference article.

- Wenfeng Li, Dingfang Chen, Christensen, H.I., & Oreback, A., **An architecture for indoor navigation** Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on ,Volume: 2 , 26 April-1 May 2004, Pages:1783 - 1788

1.6 Outline

The thesis is comprised of two main parts.

Part I: Fundamentals

This first part aims at providing a foundation on which the second part is based.

Chapter 2: Robot Architectures

This chapter contains a brief exposé of the evolution of software architectures. This will give the reader a sense of how the fundamental view has changed over the years on how robots should be programmed.

Chapter 3: Software Systems

Here we take a look at a number of successful software systems at closer detail. While chapter 2 is at a more theoretical level, these are actual implementations.

Chapter 4: Software Engineering Issues

Software engineering plays a large role when implementing a robot system. The choices made have a significant impact on the end result. Here we address issues such as operating systems and communication technologies.

Chapter 5: Development

Since the development of a piece of software in practice never reaches a final state, the development environment influences the daily life of a robot researcher. In this chapter, development methods, processes and tools are discussed.

Part II: A Design for Robotics Software

This part of the thesis shows in some detail the design and implementation of a proposed software framework for mobile robotics.

Chapter 6: Introduction

In this chapter the important task of stipulating prerequisites for our framework is carried out. Designing a system that should be able to carry out any task in any environment is not feasible, so limitations are defined for what our system should be able to do.

Chapter 7: Communication

Communication is a very important part of a robot software framework. This chapter introduces a proposed communication scheme.

Chapter 8: Ontology for Mobile Robotics

Types and objects that are used in robotics is important since it influences performance but also implicitly defines a “world view”. This chapter defines the types and objects of the proposed software system.

Chapter 9: Component Model

A component model serves the purpose of establishing a standard for developing components withing the framework. This eases development and interaction between components. In this chapter a proposed component model is presented.

Chapter 10: Hardware Abstraction

In order to provide portability across different hardware platforms, a scheme for hardware abstraction is very valuable. Such a scheme is laid out in this chapter.

Chapter 11: Architecture and Implementation

This chapter addresses the activity of assembling different components into a complete running software system. Deployment of the system is also discussed.

Chapter 12: Summary and Future Research

A summary and a section on future research concludes the thesis.

Part I

Fundamentals

Chapter 2

Robot Architectures

This chapter introduces the fundamental ideas that shaped robot architectures over time. For in depth reading on this subject, see e.g. (Arkin 1998) or (Kortenkamp et al. 1998).

The word architecture needs a short discussion. Traditionally, the word was always used when describing the software systems developed for robots. The architectures prescribes and limits how the different activities in a system are organized and scheduled. Recently however, more general frameworks have become popular. In these frameworks, modules can be assembled in order to produce many different architectures.

In this thesis we make the following distinction. An *robot architecture* is a theoretical, organizational view of the software. The implementation of an architecture becomes a system. An architecture can actually be implemented in a number a ways. When grounding an architecture into a system, a lot of issues, such as communication, must be solved. This is discussed in later chapters.

2.1 Sense-Plan-Act

Early robotic directed at single functions were designed as control systems with a clear feedback model. A sensor generates feedback, which is compared to the expected feedback which is derived from a model of the system. Any deviation is used to update the control signal so as to minimize the error over time. As complexity grew and the robots needed to perform more than one function, the perception-action loop was extended to have a planning component. This was a natural linear extension beyond traditional control towards cybernetics. Early AI, sometimes referred to as GOFAI, good old fashioned AI, used this methodology to build robot systems. This resulted in a hierarchical system having an elaborate model of the world, using sensors to update this model, and to draw conclusions based on the updated model. Actions were not a direct consequence of perception. This is sometimes called the sense-plan-act paradigm.

Albus et al proposed the Real-time Control System reference architecture RCS (Barbera et al. 1984) which is a highly layered system. Each layer consists of four parts; sensory processing, world modeling, task decomposition and value judgment. All layers share a global memory where representational knowledge is stored. The United States government even decided in the mid 1980s to fund a standard architecture for tele-robotic control. It was jointly developed by NASA and NIST and was was based on RCS and called NASREM (Albus et al. 1987). The architecture was never widely accepted but is still used today. In fact it must be used by those competing for contracts for tele-operated robots in the space program.

The sense-plan-act systems did not perform very well, especially in dynamic and unpredictable environments. Partly because of the difficulty in the modeling of the world, partly because of relying too much on inadequate sensors. Most would say that the complexity was underestimated. Another problem was that of grounding symbols in reality. Especially simulation, which was the used extensively, suffers from this problem.

2.2 Behavior Based Systems

In 1986 Rodney Brooks revolutionized the field by presenting an architecture based on purely reactive behaviors with little or no knowledge of the world. Brooks used the phrase

Planning is just a way of avoiding figuring out what to do next.

Brooks called it the subsumption architecture (Brooks 1986).

The architecture consists of horizontal layers, not vertical layers as in the state-plan-act architectures. Each layer has a distinct task and they all execute asynchronously and concurrently. Avoid-Objects and Explore are examples of layer tasks. A priority-based coordination is achieved by inhibition and suppression. Communication between layers is allowed but only at low bandwidth. Lower layers have no notion of higher layers, which makes the system easy to extend. More recent work by Brooks is COG(Brooks et al. 1999). COG is based on the idea of developmental psychology, in which the set of behaviors are acquired over time from simple to complex. The rational is that the interplay between control, representation and planning might be too complex to be engineered into a system, so it might be beneficial to use learning for this. To test scalability beyond triviality, an upper-body torso has been constructed with human like competencies in terms of motor control.

Another behavior-based approach that surfaced shortly after the subsumption architecture was *motor schemas* (Arkin 1989). Examples of schemas are; *Move-ahead, Move-to-goal, Noise, Avoid-obstacle*. Each motor schema (behavior) outputs a vector (orientation and magnitude). Vectors from all active schemas are simply summed.

The Reactive Action Packages (RAPs) (Firby 1989), should also be mentioned. It lies in between the behavior based and hierarchical methods. RAPs are task based rather then behavior based, and relies heavily on a world model. In the architecture, a set of task-situations are identified and for each of these a method is described. The methods contain sequences of steps that are to be taken in order to accomplish the task. Several systems were built upon RAPs.

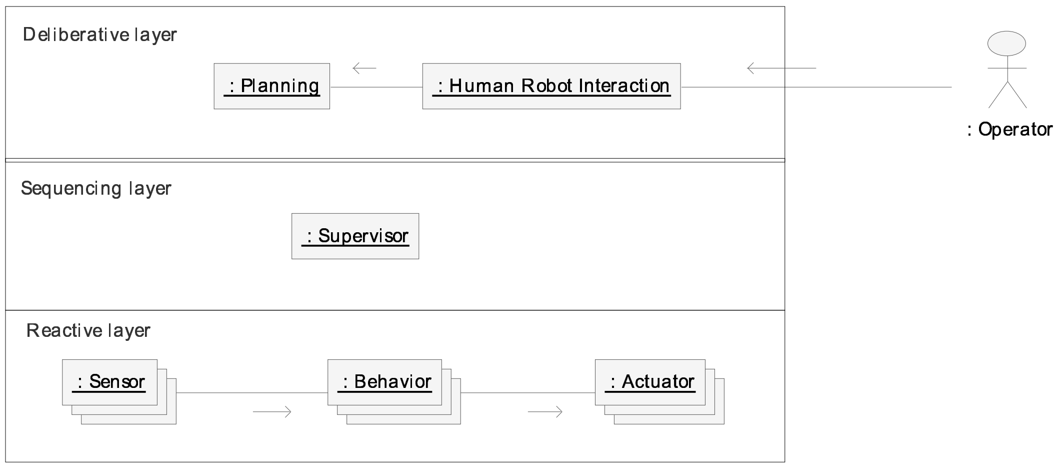


Figure 2.1: The hybrid deliberative architecture. (No vertical messaging shown.)

2.3 Hybrid Systems

Robots that were running reactive behavior based systems performed very well, also in changing environments. However, the purely reactive scheme is not capable of performing complex tasks.

A hybrid approach, combining low-level reactive behaviors with higher level deliberation and reasoning, has since then been common among researchers e.g. (Arkin 1990).

The hybrid systems are usually modeled as having three layers; one deliberative, one reactive and one middle layer.

The Reactive Layer

The reactive layer of a hybrid system is often behavior based. This means that the subsystem consists of separate behaviors running in parallel, where each behavior has one specified non-complex task. Example behaviors are goto-goal, avoid-obstacles and traverse-door.

The behaviors represents a tight coupling from the sensors to the actuators. Part of the reactive layer are also sensors and actuators. Sensors produce data that are passed on to one or more concurrently running behaviors. Sensor fusion modules can extract higher level data from two or more sensors.

Since several behaviors can be active at the same time, the results must be fused into a single crisp actuator command. This is done in an actuator command fusion module. A model of a generic hybrid deliberative architecture can be seen in Figure 2.1.

The calculations in the reactive layer should be carried out in near real-time for safety-critical considerations. The modules in this layer are normally stateless.

The Deliberative Layer

The deliberative layer handles

- mission planning and reasoning,
- localization,
- path planning,
- and interaction with human operators

Here is the global state of the system decided. Different types of state-machines can be used here. Often planning is viewed as configuration of underlying layers. Tasks in this layer are allowed to be computationally expensive and therefor take relatively long time. Ideally, this layer should be occupied with preparing for activities in the future. But in most implementations, this layer is dormant as the reactive layer is active, except for monitoring human interaction. Learning techniques can be deployed that makes the system more fault tolerant.

Localization means having an á priori map of the world and comparing perceptual inputs to this map. Whether this task belongs to the deliberative layer can be debated, but is put here since it often is very time consuming. It should be noted that a hot research topic is having the robot automatically construct the map as it navigates new environments.

There are several algorithms for path-planning, A* being a a very common one. A path-planner should take into account distances but also the time it takes to reach the targets. If route is blocked, the planner should be able to reroute if possible.

Interaction with humans can be through e.g. keyboard or speech. A syntax and vocabulary need to be defined as well as common reference points in the world. The human input need to be parsed and translated into commands that makes sense for the robot.

Obviously the skills and complexity that are needed in the deliberative layer are highly related to the amount of autonomy one is seeking.

The Sequencer Layer

The middle layer, often called either the sequencer layer, or supervisory layer, bridges the gap between the deliberative and the reactive layers. Its basic function is to rewire the reactive layer according to a global state obtained from the deliberative layer, thus deciding which set of behaviors that should be running. It should monitor the reactive layer and be informed of when steps are done, but also catch instances where the reactive layer fails. It could then execute auxiliary plans or surrender control to the deliberative layer. The sequencer can e.g. be rule-based or a finite state machine.

2.4 Lessons Learned

The hybrid deliberate/reactive has proven very successful, practical and robust in a large number of implementations, and there is general agreement that this the best type of architecture that the community has produced. However, some type of modules are hard to force into any particular layer, so the strict layering can be open for discussion.

When constructing a general framework, no specific architecture should be enforced. Nevertheless, good support for builders of the hybrid deliberate/reactive architecture is important. This implies e.g. parallel execution of behaviors.

Chapter 3

Software Systems

This chapter presents a number of implemented robotic systems. The selection is based on success, popularity and to a certain degree fame, but is by no means complete. At first we look at some of the properties by which a system can be described

3.1 Properties

As stated in section 1.4, there are a number of properties that can be defined for a system. Here are the same ones noted again as well as a couple of more:

- ease of use
- portability
- efficiency
- generalizability
- versatility
- extendability and scalability
- process distribution
- documentation
- ease of development

Ease of use refers to whether the system is easy to install, execute, and not least of all, shut down. *Portability* means if the software can be run on different hardware platforms and different operating systems. *Efficiency* is about the run time overhead which is defined by memory and CPU requirements. *Generalizability* is a measurement on how much a system imposes a certain architecture. *Versatility*, as stated in section 1.3, refers to if the system is designed to carry out many or just one task.

Extendability refers to whether there is support for adding new software modules and also indeed hardware devices. A system is said to be *scalable* if the adding of modules without the system being bogged down, which might be the case where a bottle-neck is present in the system. Research environments tend to evolve in terms of both hardware and software. Adding new sensors is pretty much a standard activity in such environments. In terms of software, in behavior based systems the addition of new behaviors is also a common practice.

A robot software system can be running in one single process, in several processes on a single host, or on processes spread over several hosts. The latter type of systems are called *distributed*. Obviously this presents both pros and cons. The pros are that the load can be spread over several machines. Robustness is also increased since if a module crashes, it will not necessarily bring down the whole system with it. Instead the faulting module can restarted or exchanged. The cons are e.g. increased complexity and more difficult debugging.

Documentation is very important for developers as well as for users. Documenting is unfortunately not the favorite activity of most programmers. Related to documentation is *ease of development*. This refers to how easy it is for other developers to add new modules to, rewrite, or debug a system.
See (Orebäck & Christensen 2003) for more on evaluating properties of robotic systems.

3.2 AuRA

The concept of the hybrid deliberative architecture is generally attributed to Arkin (Arkin 1986, Arkin 1987). The approach was implemented in the AuRA architecture. AuRA consists of a mission planner, a spatial reasoner (path planner), a plan sequencer, and a reactive system. The reactive system is based on motor schemas (section 2.2). A schema manager controls and monitors the behavioral processes during execution. A graphical view of the architecture can be seen in Figure 3.1.

Each behavior is associated with a perceptual schema that provides the stimulus that the behavior requires. The control commands from the behaviors are summed and normalized in a special process and then sent to the hardware.

AuRA is very modular and flexible. Several of the modules have been replaced over time. Learning has also been incorporated in various forms. AuRA has good generalizability which has allowed it to be used for a wide range of tasks, such as manufacturing, indoor and outdoor navigation, mobile manipulation, and military scenarios.

Our conclusion is that AuRA has a number of strengths that has attributed to its success. One is modularity which helps extendability and ease of development. It does have some generalizability but it does impose a certain architecture.

3.3 Task Control Architecture (TCA)

The Task Control Architecture (TCA)((Simmons 1994)) includes capabilities for both inter-process communications and task-level control. The inter-process communications

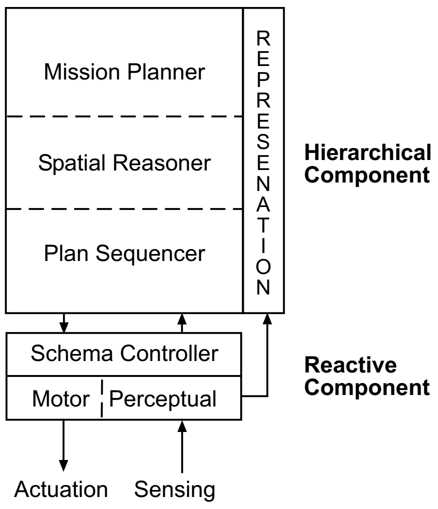


Figure 3.1: The AuRA architecture.

features anonymous socket-based communication using TCP/IP that supports both publish/subscribe and client/server modes of message passing. TCA also supports automatic marshaling and unmarshalling of data based on a format definition language. All communications in TCA are routed through a central server, which can log all message traffic.

The task-level control portion of TCA includes capabilities for hierarchical task decomposition, task sequencing and synchronization, resource management, execution monitoring, and exception handling. By "task-level", is meant the integration and coordination of perception, planning and real-time control to achieve a given set of goals (tasks). A central server dispatches tasks.

TCA can be thought of as a robot operating system and can be used for a wide variety of robots, tasks, and environments. One example is the XAVIER robot (O'Sullivan et al. 1997) that has been developed at CMU. The system is composed of four layers with specific functions: task planning, path planning, navigation, and obstacle avoidance.

Later, the properties of TCA was rewritten split into two packages, IPC (Inter-Process Communications) and TCM (Task Control Management). IPC has communication features similar to TCA but also peer-to-peer communications as well as other new enhancements. TCM is a complete reimplementation, written in C++, of the TCA task-level control capabilities.

Later the Task Description Language (TDL)(Simmons & Apfelbaum 1998) was developed. TDL is a superset of C++ that includes explicit syntax for task-level control capabilities. The objective was to facilitate writing task-level control programs by embedding such syntax in a language familiar to roboticists. TDLC, a translator written in Java, transforms TDL code to pure C++.More recently, TDL has been extended to work in a distributed fashion, MTDL (Multi-TDL).

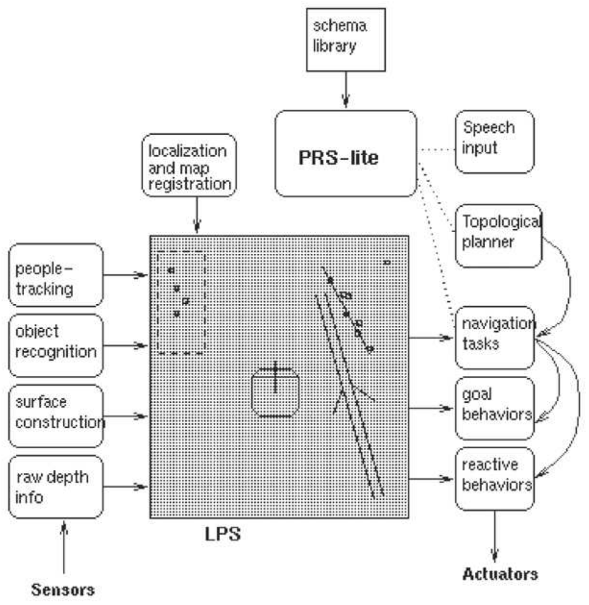


Figure 3.2: The Saphira system.

TCA and its successors have been deployed in a large number of projects at CMU, NASA, and elsewhere. The task-control part stipulates a specific architecture but the communications part is general enough to suit most people but the socket-based approach might be too restrictive to provide a general framework.

3.4 Saphira

SAPHIRA (Konolige & Myers 1996) is a robot control system developed at SRI International's Artificial Intelligence Center. It was first developed in conjunction with the Flakey mobile robot project (Saffiotti et al. 1993), as an integrated architecture for robot perception and action. The software runs a reactive planning system with a fuzzy controller and a behavior sequencer.

There are integrated routines for sonar sensor interpretation, map building, and navigation. At the center of the architecture is the Local Perceptual Space (LPS), see Figure 3.2. It accommodates various levels of interpretation of sensor information, as well as a priori information from sources such as geometric maps. The main system consists of a robot server that manages the hardware, and Saphira which is a client to this server.

Saphira has been implemented on a number of different operating systems and in addition an API across a number of languages has been developed, in particular to support different types of experiments from low-level control to task planning. There are several

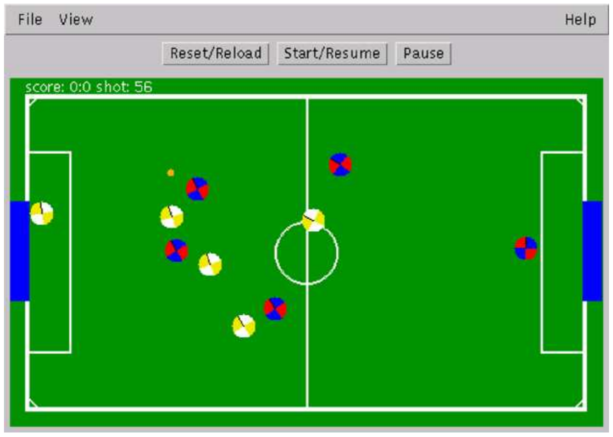


Figure 3.3: A Teambots soccer simulation.

coding methods used in the Saphira architecture. The core of the system is programmed in the C language. A special high-level interpreted language has been designed called *Colbert* (Konolige 1997). It has a C-like syntax with semantics based on finite state machines. A part of Saphira is written in LISP.

One rather confusing matter is the distinction between activities, processes, behaviors, tasks and routines. The learning curve is thus steep to master the system at a reasonable level although colbert presents a convenient scripting language. The robot server concept is crude and means that portability is difficult. All sensor and control data pass through one communication channel. Both this and the LPS are potential bottle-necks. The fact that fuzzy logic control is basically mandated by Saphira puts unnecessary restrictions on the programmer.

3.5 Teambots

TeamBots (Balch 2000) is a Java-based collection of application programs and Java packages for single- and multi-agent mobile robotics research. A very large collection of classes and interfaces are available for developing new software.

One selection of these classes is called *Clay* which is a package of Java classes that can be combined to create behavior-based robot control systems. Clay takes advantage of Java syntax to facilitate combining, blending and abstraction of behaviors. Clay can be used to create simple reactive systems or complex hierarchical configurations with learning and memory. A basic interface is inherited by all robot classes. TeamBots is widely used in research and education. It has been ported to a number of robot platforms. TeamBots is primarily constructed for simulation, and it has an easy to use graphical interface for such purposes (see Figure 3.3).

The TeamBots architecture has no real deliberation layer but does contain methods for sequencing of tasks.

TeamBots is entirely built in JAVA and the source has been carefully grouped into a collection of fine grained classes. TeamBots can thus be run on basically all platforms that support JAVA. This includes most types of UNIX, MS Windows, as well as MacOS. Note that in order to run on a real robot, the actual hardware must be supported by device drivers. The device drivers are typically realized by using JNI (Java Native Interface) which wraps platform-specific functionality.

Many people consider Java to be inappropriate for a time critical system like that of a robot. It can be easily shown that a large portion of the time that the system spends in a control loop, consists of calls to the hardware. This is totally irrespective of the programming language used. There is also the possibility of using JNI to embed time critical code written in other languages.

Teambots could be called a general framework but does stipulate an architecture, especially regarding robot control. It is also a single process application which makes it non-distributable and less robust.

3.6 BERRA

BERRA (BEhavior based Robot Research Architecture) (Lindström et al. 2000), is an architecture with the primary design goals of scalability and flexibility. All components are heavy weight processes and can be transparently placed anywhere on the network. The implemented system makes heavy use of the Adaptive Communication Environment (ACE) (Schmidt 1994) package. By using this package, OS dependent system calls are wrapped, allowing for portability across a wide range of Operating Systems.

ACE also includes powerful patterns for client/server communication and service functions (Schmidt & Suda 1994) which are used in the system. The implemented system has been tested in a significant number of missions in the lab, where one room has been set up as an ordinary living room (Andersson et al. 1999). An abstract representation of the BERRA architecture is shown in Figure 3.4.

Sensors and sensor fusion modules are called *resources*. *Controllers* represent both the actuators and actuator command fusers. The middle layer is called the *Task Execution Layer*. BERRA has been tested on Solaris and Linux and it has be evaluated on a range of different platforms including Nomadic 200, Nomadic Scout, Nomadic XR4000, and ActivMedia Pioneer. Theoretically, all platforms supported by ACE can be considered. BERRA is written entirely in C++.

The BERRA system clearly separates the system into IPC, component architecture, and control. The multi-process behavior based scheme provides good performance but also makes it hard to debug. It is poorly documented which makes is hard to use and to extend.

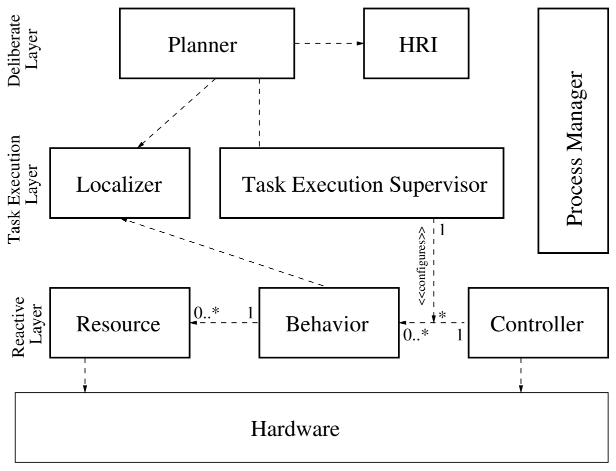


Figure 3.4: The BERRA system.

3.7 Smartsoft

Smartsoft (Schlegel & Wörz 1999, Schlegel 2004) is a component framework and architecture for robot systems. Smartsoft is in many ways very similar to BERRA. One interesting difference is that at the core of the framework are a number of communication patterns.

- AutoUpdate Timed
- AutoUpdate Newest
- Command
- Query
- Event
- Configuration

The *AutoUpdate* pattern is often referred to as *Push*. It is a subscription based concept used when clients want new data as soon as it is ready. Typical uses are when behaviors consume data from sensor-data servers.

In the *Timed* version, when clients issue a subscription, a value is passed along noting the desired time interval between updates. This can also be called synchronous mode.

In the *Newest* version, the clients want updates as soon as new data becomes available, no matter how long or short the time interval is. This can be called an asynchronous mode.

Query is like a method-call returning a value. The call can optionally have a parameter. This resembles what is often referred to as *Pull*.

Command is a simple non-returning (like void in C/C++) method-call.

In the *Event* pattern, the client asks the server to continuously evaluate a boolean expression, and to inform the client when the expression returns true.

Configuration is a rather SmartSoft-specific pattern where clients can set and query the internal state of a server.

The communication is socket-based but was later rewritten to use CORBA (section 4.3). The component framework provides very good communication but is too complicated and in a sense hierarchical. The components internal state is e.g. always controlled from the outside, something not everyone agrees with.

3.8 Player/Stage

Player (Gerkey et al. 2001, Gerkey et al. 2003) is a robot device server developed at the University of Southern California Robotics Research Labs. Player is a socket-based device server that allows control of a wide variety of robotic sensors and actuators. Player executes on a machine that is physically connected to a collection of such devices and offers a TCP socket interface to clients that wish to control them.

Clients connect to Player and communicate with the devices by exchanging messages with Player over a TCP socket. In this way, Player is similar to other device servers, such as the standard UNIX printer daemon lpd. Like those servers, Player can support multiple clients concurrently, each on a different socket. Because Player s external interface is simply a TCP socket, client programs can be written in any programming language that provides socket support.

In order to provide a uniform abstraction for a variety of devices, Player follows the UNIX model of treating devices as files. Thus the familiar file semantics hold for Player devices. For example, to begin receiving sensor readings, the client opens the appropriate device with read access; likewise, before controlling an actuator, the client must open the appropriate device with write access. In addition to the asynchronous data and command streams, there is a request/reply mechanism, akin to ioctl(), that clients can use to get and set configuration information for Player devices. Several clients can connect simultaneously to a device as there is no locking mechanism. Device interfaces are separated from device drivers.

Stage simulates a population of mobile robots moving in and sensing a two-dimensional bitmapped environment. Various sensor models are provided, including sonar, scanning laser range-finder, pan-tilt-zoom camera with color blob detection and odometry. Stage devices present a standard Player interface so few or no changes are required to move between simulation and hardware.

The Player/Stage project offers a device abstraction across robotics platforms be it in simulation or physically instantiated, however it does not impose any architectural constraints so it may, at least in principle, be used with many of the other systems described in this section.

The Player/Stage project has become very popular and widely used. The success is undoubtedly because of the open and unrestrictive nature of the system. There are however some drawbacks. It is using a client-server model and not a peer-to-peer model. The message formats are rather static. It also makes a number of assumptions about hardware which make it hard to port some platforms.

3.9 Lessons Learned

Quite a few lessons can learned from this study. Unless you really want to impose a architectural constraints, the system should be as flexible and general as possible. This also implies a peer-to-peer model without central servers.

Communication should be separated from the rest of the framework. The concept of communication patterns as used in Smartsoft is very attractive. Furthermore, hardware abstraction should be emphasized in order to ensure portability between hardware platforms and to ease the adding of sensors. Documentation is often a problem. This is of course related to ease of use. But even the best documentation can not make up for a complicated design.

Notable is that few systems have a fully fledged planner/deliberation system. This is likely because it is often the last part of a system that gets attention. It is also complex and requires specific domain knowledge. This knowledge is probably rare in robot laboratories today.

Regarding programming language, many systems are based on C++. Other languages are possible, but at least a C++ api would be good. Another observation is that real-time is rarely a major design criteria.

Chapter 4

Software Engineering Issues

Going from theory to practice, to actually implement a system, presents a whole range of problems and design decisions to be made. One has to choose operating system, programming language, and whatever tools and libraries to use. These issues will be discussed in this chapter.

4.1 Operating Systems

The operating system (OS), takes care of several important tasks in a computer system, such as

- handling of hardware,
- providing a file system,
- scheduling user programs,
- handling of security, and
- providing a number of services for user programs.

All of these items are essential and are present in any standard OS. Recommended reading on this matter is the classic book (Tanenbaum & S.Woodhull 1987,1997).

Basic Properties and Services

Here is a list of properties that may or not be supported by an OS,

- multi-tasking,
- multi-CPU support,
- multi-user,

- threading,
- network communication.

There are however a number of low-level services that are important for a robot application programmer. These include semaphores, signals, shared memory, pipes, fifos, and sockets. There are other considerations to take into account such as available software, tools and hardware support.

The properties and services needed from the OS in a robotic system of course depends on what type of system that one is constructing. Multi-tasking is needed e.g. where multiple behaviors will be executing concurrently. IPC (Inter Process Communication) is then also required. Network communication is needed for a distributed system.

Fortunately, all of the above properties are available in modern operating systems such as recent versions of Windows and dialects of UNIX. Granted, differences exist e.g. regarding processes and threads. A potential list includes:

- Windows XP/NT/2000
- Sun Solaris
- Linux
- QNX
- VxWorks

Real-Time

Operating systems can also be either real-time or non real-time. Strictly speaking, there are some that are considered to be in between (soft real-time), but that distinction is not made here. In the service robotics field, real-time operating systems are rarely used. This is because hard real-time properties are not really necessary, normal OS's provide satisfactory timely execution. A mobile robot does not really have to react to obstacles faster than in the order of tenths of a second (compare humans). But if the robot is equipped with a manipulator, real-time must be used. This is because it is needed to perform the fine control necessary to pick up objects, and for safety reasons. It is actually more dangerous to be hit by a robotic arm than by a robot moving around. A common misconception is that real-time has to do with speed. This is not true, it is a question of guaranteeing that a task will be executed within a given time-frame or not.

Linux

Linux is today the most widely used OS in robotics. There are a number of reasons for this.

- is based on the GNU tools and environment

- is open source with a free license
- good hardware support
- is well documented
- has no monetary cost
- has excellent performance and stability

Regarding hardware support, it may not always be the first OS to support new hardware, and uncommon hardware might not be supported at all. This issue is however constantly improving.

Linux is not real-time, although more and more real-time capabilities are brought into the stock kernel. Before the era of Linux, Microsoft DOS was many times used. Among the real-time OS's, QNX and vxWorks used to be the most popular. However, the trend moves towards using the Linux versions RTLinux and RTAI. Especially the latter, as the RTLinux is encumbered with patents.

4.2 Object Oriented Programming

Quite contrary to common belief, object-oriented programming is not very new. It first saw the light of day in the programming language Simula 67 developed at the Norwegian Computing Center, Oslo, Norway by Ole-Johan Dahl and Kristen Nygaard. Simula introduced important object-oriented programming concepts like classes and objects, inheritance, abstract data types, and dynamic binding. Alan Kay's group at Xerox PARC used Simula as a platform for their development of Smalltalk and Bjarne Stroustrup started his development of C++ (in the 1980s) by bringing the key concepts of Simula into the C programming language. Object-oriented programming is today becoming the dominant style for implementing complex programs with large numbers of interacting components. Among the multitude of object-oriented language are Eiffel, Eifeel and PROLOG. In particular the Internet-related Java (developed by Sun) has rapidly become widely used in recent years.

In robotics, C was the predominant language up until the mid nineties. The reason for this is probably the fact that robotics has been very closely related to programming hardware. Even today, device drivers and hardware API's are generally written in C. Obtaining the highest possible execution speed has also been desirable, and although not necessarily true, C was widely attributed as being the fastest option. As the ambition, size, and complexity of robotics projects grew, the need for properties like encapsulation, separation of concerns, and code reuse became more important. Since then, most members of the robotics community have been using C++ and lately we have seen systems being programmed in JAVA.

4.3 Communication

A key feature in a robot system is a reliable and efficient communication mechanism. Modules need to exchange data such as sensor data and events such as emergency stop commands. Runtime monitoring and error handling are also of great importance.

Robotic systems are often distributed over a number of hosts. This happens when the load is too high for the on-board processing power and off-board computers are needed. Even single vehicles are more and more equipped with more than one computer. Distribution is also integral in sensor networks and multi-robot systems. Another situation is during the development phase, where it is convenient to work at a desktop computer controlling a remote robot.

Communication Technologies

Operating systems provide various support for inter process communication (IPC). Sockets and shared memory are probably the most encountered mechanisms within robotics.

UNIX sockets Protocol for connections between processes on the same machine.

INET sockets Protocol for connections between processes on different machines.

Shared memory A communication over shared memory can be used between processes what can share a common memory block. This is preferably used when there are large chunks of information that needs to be transferred fast, for example images.

Other OS supported methods for communication are pipes, files, FIFOs, and signals. Note that also single process programs often need communication methods, e.g. in order to synchronize between threads.

There are however some problems associated with these protocols. One is that these methods operate on a rather low level and they are therefor error prone and not convenient to use directly. But the biggest problem is that different platforms use different byte-ordering, the so-called endian problem. This means that data cannot be sent across such platforms without conversion.

The first standard to address these shortcomings was *remote procedure calls* (RPC). Here the caller does not call the remote end directly, but calls a local *stub*. This stub linearizes (marshals) the data and sends to the remote part. Here another stub unmarshals the data and passes it on the the actual callee. Neither the caller nor the callee need to know, or indeed care about, that the communication is not local. RPC is in use in many systems, nowadays often in combination with XML 4.5. Light-weight versions can be found in the Windows NT operating system and commonly provide the communication in graphical user interfaces.

Middleware

A good definition of middleware is provided by (Emmerich 2000):

> "A layer between network OS's and applications that aims to resolve heterogeneity and distribution."

One can also say that middleware coordinates how parts of applications are connected and how they inter-operate. This enables and simplifies the integration of programs developed by different vendors. The most well known of these are CORBA, (D)COM, and Enterprise JavaBeans (EJB). They are disjunct but can be made to interact by various bridges available. .NET is a newly arrived contender from Microsoft.

Some middleware can also be called *component* technologies. Component-Based Software Engineering (CBSE) is a rather new approach towards system-building. The idea is that systems can be made by assembling building blocks, components. These components can either be commercial off-the-shelf (COTS) or produced in-house. Components have the following properties:

1. A component is a binary unit of deployment.

2. They implement one or more well-defined interfaces.

3. Should be re-usable between applications.

Similar concepts are also found in plugin-based architectures such as web-browsers and media-players.

Components are meant to be more easily re-used than classes or objects. Components should correspond to readily-understood entities, while classes are generally more fine-grained with dependencies requiring detailed knowledge. Moreover, the binary nature of components imply the following:

- They can be deployed without any familiarity with the source code, facilitating re-use.

- As new versions of a component is released, the user can just replace the entire binary.

- Components can be shipped by industrial companies without giving away source code, paving way for a market.

The hope is that software engineers would be able to browse and purchase components from catalogs, much in the same way as mechanical engineers can do with hardware parts. This is becoming true at places like www.componentsource.com.

A number of middleware technologies are explained below in some detail. One major source of reference for this section is (Szyperski 1998).

DCE

The *Distributed Computing Environment* (DCE) is a standard of the Open Software Foundation (The Open Group 1997). It implements full RPC between machines across heterogeneous platforms. It also uses version control of services. DCE is included here mostly for historic reasons, since it introduced many key aspects of modern component technologies. First and foremost the *interface definition language* (IDL).

IDL IDL separates the interface from the implementation. It to defines the interface that sits on the outside of the boundary, allowing the only communication that the object conducts with the outside world. The language enforces object orientation and supports robust exception handling. IDL is independent of programming language, but maps to other programming languages. For each remotely called procedure, IDL specifies the complete signature including the types of parameters and return values. Properties like the ranges of basic types are fixed in order to ensure crossing of machine boundaries. Both the client- and server-stubs mentioned earlier are generated from the same IDL file. IDL is an published ISO International Standard (ISO/IEC 1999).

UUIDs DCE also introduced the concept of *universally unique identifiers* (UUIDs). Names are constructed using an algorithm that guarantees uniqueness. These names are unreadable and meaningless for humans.

CORBA

CORBA(Common Object Request Broker Architecture) is a standard managed by the Object Management Group (OMG) (OMG 2004). See e.g. (Henning & Vinoski 1999) which is an authoritative book on CORBA and (Siegel 2001) which includes later development.

With CORBA, OMG set out to solve the problem of allowing for object-oriented systems implemented on different platforms in different languages to communicate. The target area was enterprise computing. Initially released in 1991, the open approach led to incompatibility between implementations. With CORBA 2.0 released in 1995, the standard included the Internet inter-orb protocol (IIOP). This rectified the situation and the CORBA story eventually turned into a success. More so with CORBA 3.0, important new additions have been made. Among these are integration with XML, the introduction of valuetypes and a component model. The latter means that CORBA is becoming a full-featured component technology. Valuetypes enables call-by-value instead of call-by-reference.

CORBA applications are made up of objects, units of running software that combine functionality and data. For each object type, an interface is defined in OMG IDL. The interface is a sort of contract that the server offers to clients that invoke it. OMG IDL maps to all of the popular programming languages C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and possibly more.

In CORBA, the Object Request Broker (ORB) takes care of all of the details involved in routing a request from client to object, and routing the response to its destination. On the sever side, the ORB also de-activates inactive objects, and re-activates them whenever a request comes in.

Any client that wants to invoke an operation on the object must use this IDL interface to specify the operation it wants to perform, and to marshal the arguments that it sends. When the invocation reaches the target object, the same interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them. The interface definition is then used to marshal the results for their trip back, and to unmarshal them when they reach their destination.

In order to invoke the remote object instance, the client must first obtain its object reference. There are several ways to do this, e.g. the Naming Service and the Trader Service (see section 4.4).

CORBA is not without criticism. It is by many regarded as a slow-moving beast with a steep learning-curve. There are many options and ways within CORBA to solve a given problem. Component versioning is a not fully solved problem in CORBA. The different implementations still suffer from minor incompatibilities and usually implement the standards to a variable degree. Many feel however that the situation has improved during the last years. Several CORBA implementations exist on a large number of platforms, many of them are free and open-source. Many IDEs (Integrated Development Environment) have standard templates for Corba based IPC.

COM and DCOM

COM is Microsoft's technology for component based software. COM stands for Component Object Model (Microsoft 2004). It is a *binary* standard and is language-independent with primary implementations in VB (Visual basic) and VC++ (Visual C++). Every COM interface has to support the method QueryInterface. By using this method, a client can get from any interface to any other interface supported by that component. By starting at the mandatory interface called IUnknown, initial entry can be made. This interface is also used to uniquely identify the component. COM uses a modified version of UUID called GUID (Globally Unique Identifiers). Reference counting is used for memory management. COM has it's own version of IDL, but using the standard development tools, the IDL is automatically generated. A COM interface can never be changed without incrementing the version number. Furthermore, old versions of interfaces must be preserved. This can be seen as drawbacks, but actually solves the versioning problem.

Distributed COM (DCOM) builds on top of COM and uses RPC in order to provide communication between processes on different machines.

The main niche for COM has been in desktop applications in relation to the Windows operating system. The Microsoft Office suite is a good example. However, a more for us relevant example is ABB's OperateIT.

Apart from Microsoft Windows, COM is available on the Macintosh and a number of other platforms. No free or open-source version seems to exist though. Development tools are very well advanced.

COM has evolved gradually with several bekindered technologies such as VBX, OLE, ActiveX and COM+. The strive for maintaining backwards compatibility means that many reduntant mechanisms within these technologies exist. Microsoft now recommends that developers use the .NET Framework (see section 4.3) rather than COM for new development.

Enterprise JavaBeans

Sun has created Enterprise JavaBeans (EJB) in order to provide a component technology for Java. This is mainly achieved by the introduction of the object serialization service and the remote method invocation (RMI). The garbage collection significant for Java has also been extended to work in a distributed setting. The versioning problem is addressed in a painstaking way.

JavaBeans has primarily been used in web applications. Development tools are somewhat available.

XML-RPC and SOAP

XML-RPC and SOAP are very similar and shares the same ancestry. They do remote procedure calling using XML as the encoding. It communicate over HTTP in order for it to go through firewalls. There are many compatible implementations that span many operating systems and programming languages.

XML-RPC came first but is now more like a subset of SOAP. The specifications is roughly five pages, and very easy to understand. It has been called "low-tech wire protocols based on the standards of the Internet" by Dave Winer, one of its authors.

SOAP extends XML-RPC by implementing user defined data types, the ability to specify the recipient, message specific processing control, and other features. XML-RPM messages can used with SOAP by embedding it in an envelope.

.NET

The .NET framework is a vast Microsoft initiative comprising several technologies. It is primarily designed to be used for so called *Web services*. A Web service is designed to be accessed directly by another service or software application. Web services are reusable pieces of software that interact over the network through XML and SOAP. Web services can be combined with each other and other applications to build so called *.NET experiences*.

The framework includes a new programming language called C#, which could be called a Microsoft fork from SUN's JAVA. It adds additional features related to component development.

A "common language runtime", which runs bytecodes in an Internal Language (IL) format. Code and objects written in one language can be compiled into the IL runtime. Providing that an IL compiler is developed for the language.

The .NET technologies are very much confined to the Microsoft platforms. While the motto of JAVA is *write once, run anywhere*, the .NET clr is quite the opposite.

There is an open-source initiative called MONO which aims to provide some features of .NET to the Linux community.

Conclusions

Middleware lift the problem of distributed programming to a higher level of abstraction. This is indeed useful as robotics is inherently distributed. Using a technology that is standardized and used by a vast number of people, is preferred over constructing a separate communication toolkit for robotics.

Introducing component-based software engineering to robotics would certainly also be very beneficial. In fact, it could help to solve much of the problems stated in 1.4. First of all, complexity could be managed since CBSE enforces modularity at a functional level. By allowing domain experts to only concentrate on making "their" component without having to deal with intricacies of the rest of the system. Furthermore, components that internally use different algorithms but share the same interfaces can easily be interchanged. This facilitates comparison and verification.

However, none of the mentioned technologies can really be called a framework or a system architecture. No matter which technology is chosen, the available degrees of freedom are many. In order to produce something that can be called a component architecture, restrictions on how components can interact must be set as well as how components can be assembled and used.

COM is probably the most successful technology, largely due to the success of the Windows Operating system. A large number components are available commercially. The lack of free and open implementations is however a major drawback as well as poor platform and language support.

Based on JAVA, EJB is of course available on many platforms but then also excludes other languages. Since it is mainly designed for web applications, usage in a domain like robotics where (near) real-time behavior is needed, performance can be questioned.

The same can definitely be said about XML-RPC, soap and .NET. The latter also poses problems because it lacks support on many platforms and all the specifications are not open.

Some argue that using middleware results overhead and a decrease in efficiency. This is however not necessarily true and in any case, a high increase in productivity and stability should be expected. Problems that incur when communicating between different platforms, languages and operating systems are also taken care of. Middleware is today not applicable in embedded applications due to the large footprint. However, there is a concerted effort to drive down the footprints. At the same time hardware performance is pushing up. As it stands today, CORBA seems to offer the best solution. Especially as it is the only true vendor-neutral product.

4.4 Services

Components typically make use of external services in order to e.g. locate other components.

CORBA Services

OMG has defined a number of services. The following list does not include them all, rather a subset which contains the ones most relevant for robotics..

Naming Service

The Naming Service provides a basic service location mechanism for CORBA systems. It manages a hierarchy of name-to-object-reference mappings. Typically the server process hosting an object, binds an object reference with a name in the Naming Service by providing the name and object reference. Names can be grouped into a hierarchy using contexts. This is very similar to the directory structure in file systems. In this analogy, contexts are like folder-names. Clients can then query for names and navigate naming contexts.

More recently, CORBA Naming Service was subsumed/extended by the CORBA Interoperable Naming Service, a.k.a. INS. This inherits all the functionality from the original Naming Service specification in addition to addressing some its shortcomings. In particular, INS defines a standard way for clients and servers to locate the Naming Service itself. The INS service allows the ORB to be configured administratively to return object references from CORBA::ORB::resolveinitialreferences for non-locality constrained objects. The service also introduces the corbaloc IOR format, which can be used to bootstrap services not available at ORB installation time. The corbaloc format is a human-friendly format which resembles a www-address.

The Naming Service Mechanism can be seen as a telephone book's white pages,

Trading Service

The Trading Service can be likened to a telephone book's yellow pages. It is used when a client wants to locate another component that has some specific properties. In the robot context this could be a behavior wishing to locate a range-sensor. A server that wants to use a Trader Service to advertise its service is called an *exporter*. The exporter must first ensure that its service type is present in the service type repository of the target trader. A service type is basically a list of properties. The exporter then sends this list *filled in* to the trader. A client that wants to find an exporter is called an importer. The importer constructs a query in which provides the name of a service type and acceptable constraints on the values for the properties. Upon reception of this query, the trader returns any matches.

Event Service

The COS Event Service provides a means of distributing data from one server to one ore more clients. This can be done in either push or pull fashion.

The Implementation Repository

According to the CORBA specification, "The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects". As further explained in (Henning n.d.), the Implementation Repository performs the following three functions.

- Maintain a registry of known servers
- Record which server is currently running, and which port and host it uses.
- Starts servers on demand if they are registered with the Implementation Repository.

4.5 Document Markup Technologies

SGML and XML are languages that are used for defining markup languages. In other words, they are metalanguages that facilitate the definition of descriptive markup languages for the purpose of electronic information encoding and interchange. SGML and XML support the definition of markup languages that are hardware- and software-independent, as well as application-processing neutral.

SGML

Standard Generalized Markup Language (SGML) is an ISO standard (ISO 1986). The key idea underlying SGML is separating the representation of information structure and content from information processing specifications. Information objects modeled through an SGML markup language are named and described using attributes and sub-elements in terms of what they are, not in terms of how they are to be displayed or otherwise processed.

SGML is very large, powerful, and complex. It has been in heavy industrial and commercial use for over a decade, and there is a significant body of expertise and software to go with it.

XML

XML (Extensible Markup Language) (W3C 2004) is a dialect of SGML that is designed to enable 'generic SGML' to be served, received, and processed on the World Wide Web. XML originated in 1996, as a result of frustration with the deployment of SGML on the Internet. XML simplified the requirements for implementation, with the specific intention of enabling deployment of markup applications on the Internet. XML is a lightweight cut-down version of SGML which keeps enough of its functionality to make it useful but removes all the optional features which make SGML too complex to program for in a Web environment.

A DTD is a formal description in XML Declaration Syntax of a particular type of document. It sets out what names are to be used for the different types of element, where they may occur, and how they all fit together. A broad range of commercial and free software has been developed to assist users with markup implementation. DOM and SAX are corresponding APIs that are language independent and supported by numerous languages.

XML can and is used in many more areas than the Internet. Anywhere where data needs to be transfered or processed, XML is becoming the the preferred choice for many people. Within robotics there is a project called RoboML (www.roboml.org), that is an XML-based language for data representation and interchange in robotic applications. Combined with a set of communication protocols, it is aimed at providing a common interface for hardware and software robotic agents communicating via Internet networks.

XML would be an useful tool to describe and configure a robotic system. When it comes to using XML for transport one must be aware that the efficiency is quite low. There is however an association between XML and the CORBA Valuetype which can be used as transport object.

4.6 Lessons Learned

The most important findings in this chapter are

- Using *off-the-shelf* middleware is an effective option to handle communication.
- The component-based approach could help to solve many problems of robotics.
- Linux is a good choice for operating system.
- XML is an effective tool for certain areas of robotics.

These findings will be further discussed in Part II.

Chapter 5

Development

The development side of any computer system is of paramount importance. Design and structure need to be communicated between developers, and adequate tools should be used that aid the individual developers as well as supports collaboration. Documentation should be extensive and current. The learning curve for new developers should be as shallow as possible. Easy code maintenance should be planned for. All these requirements are of course easy to list, but it is often hard to find the time, resources and motivation to satisfy them. Programmers often prefers to dive straight into coding. Although it is not immediately apparent to the end-users, ease of maintenance also effects them. if bug-fixes, updates, and feature extensions are not being put out by the vendor on a regular basis, it might very well be because the software was not developed with maintenance in mind.

5.1 Levels of Programming

There are several good reasons to separate the design, development and system usage into different levels. These layers should reflect the levels of abstractions of a system and thus helps the initial design as well as makes it easier for novice developers in getting started. Maintenance is also greatly benefited.

The System Architect

The system architect is responsible for the global design decisions that make the foundation for all further development. This group usually consists of the initial developers of a system. For example in the type of system we are discussing, this means designing the infrastructure. If the architect can make the design comprehensible to other programmers, the architect does not necessarily have to do the actual coding. The system architect should have expertise in computer science and systems, but should also have very good knowledge of the domain for which the system is made (robotics in this case).

The Component Programmer

The component programmer creates new components with specific competence, e.g. device drivers, robotic behaviors, localizers. Components should typically be programmed by domain experts. Developing new components should not require deep knowledge in computer science, but a fair degree of programming skills could be expected.

The Application Programmer

The application programmer assembles and configures components into a working complete system. A graphical user interface could be provided at this level.

The End User

The end-user starts, executes, monitors and stops a working system. There should absolutely be no requirements of skills in programming or indeed advanced computing at this level. The job could be carried out using a graphical user interface, voice recognition, gestures, joysticks etc.

This group hardly exists for a research/academic robotic system, but never the less it should always be considered at the other levels. Of course if and when a system is offered commercially, this level is very important to guarantee success in the market.

5.2 Unified Modeling Language

The governing ideas and principles of a software system can be visualized in a graphic manner. A single graph can often clarify pages of explanatory text. UML(UML 2004), the Unified Modeling Language has become is a standard tool for both design and visualization.

UML deals with *things*, *relationships* and *diagrams*. Things are divided into structural, behavioral, grouping or annotational things. The structure of a class is portrayed as in Figure 5.1, where it can be seen that the class-name is shown in the top compartment, attributes in the middle, and operations in the bottom compartment. One kind of relationship is also shown in that diagram, inheritance.

Several types of diagrams can be made with UML.

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram

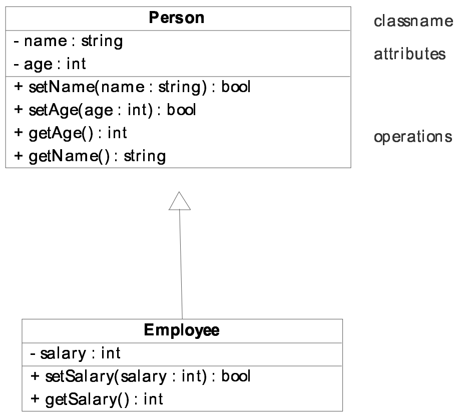


Figure 5.1: A UML class diagram.

7. Activity diagram
8. Component diagram
9. Deployment diagram

UML does have a syntax, but it is generally considered that one has a significant amount of freedom - the main goal is to convey ideas. Several profiles have been made to provide standard means for expressing the semantics of different areas, such as CORBA IDL. A good source for learning more about UML is (Booch et al. 1999). UML has the advantage that it is supported by several tools that allow design, re-engineering ,analysis and code-generation. Most of them cost rather much. There are a number of free open-source tools, but they are generally not as advanced.

There is one major pitfall associated with UML. That is the risk of putting too much into one diagram. Even trying to fit in all classes in a small system into one diagram will result in something messy and incomprehensible. So instead of lowering the learning curve of a system, it will effectively scare people away. This means that one should really try to limit the scope and to choose suitable abstraction levels when constructing UML diagrams.

5.3 The Development Process

Most projects are probably being executed without any particular development process in mind. This may be fine for very small projects, but not if the project is of greater size, involves a lot of people, or represents a financial stake. Using accepted methods and processes is also a prerequisite for some ISO certifications which in turn will provide benefits in a competitive market. The documentation obtained during the development process is also very useful during later maintenance and possible refactoring.

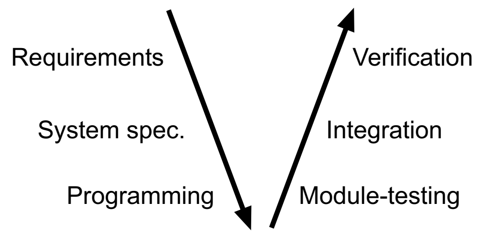


Figure 5.2: The V development approach.

The Waterfall Approach

This is considered to be the oldest method. It is basically a single-pass approach from requirements analysis to delivery. Results are frozen after every stage and nothing can be done in parallel. The name probably comes from the analogy that in a waterfall the water only flows in one direction. Of course this makes it a highly inflexible approach.

It is worth noting that some (Weisert 2003) argue that this method never really was proposed or advocated by anyone, but rather a construction by people who wanted to sell new methods and need old methods to miscredit. None the less, the essence of this approach is in use still today. At a large Swedish telecom company, binary patches are known to be applied by the engineers at the testing department, rather than feeding back the results upstream.

The V-Model

The V-model is said to be a refinement of the waterfall approach. A graphical view of the approach can be seen in Figure 5.2. There are several versions of the model with different labels across the axes. The meaning of the arrows also vary. Some mean that stages can be executed in parallel, others say that it is single-pass just as the waterfall approach. One good property is that testing and verification is assigned as high importance as the programming.

The Rational Unified Process

The Rational Unified Process, RUP (Jacobson et al. 1999), was created by *The Three Amigos*, i.e. Grady Booch, Ivar Jacobson and James Rumbaugh. A *software life cycle* is a cycle over four phases in the following order: inception, elaboration, construction, transition. A *phase* is the span of time between two major milestones in the software development process. Major milestones can be thought of as synchronization points where a well-defined set of objectives is met, decisions are met to move or not to into the next phase.

The essence of RUP is iteration. And the essence of iteration is that each iteration ends in a deliverable. A project plan is not a statement of what will be. Rather it is a statement of how risks will be managed.

5.4 The Open Source Model

The *open-source model* is not really a development model although many refer to it as such. It is rather a philosophy. It emulates the classical scientific research approach - to build on other peoples work and to share the results. There are however a few interesting methodologies that have emerged within the open-source communities.

There are a large number of very successful open source projects.

- Apache web server
- Mozilla Internet browser
- Linux operating system
- OpenOffice.org office suit
- The Gimp image manipulation

All these are widely used and are (except for Linux) available for several operating systems. Let us take a look at the background.

Background

In the beginning of the software development history, the cultural tradition was to share the software code (Stallman 1999). But when commercial software grew as an industry, the incentives to protect investments in development increased, and vendors introduced restrictions and non-disclosure agreements to protect their products from being copied.

This move into a proprietary software tradition was strongly opposed by Richard Stallman who was then a researcher at the MIT AI Lab. He supported the scientific research approach; knowledge must be shared and distributed freely in order for society to develop.

In 1984 Stallman started the GNU project with the aim to create a complete operating system that was completely based on free source code. He started a community for developers who all could contribute. They started with constructing the compiler and other necessary tools. The work on the kernel progressed rather slowly. Then in 1991 Linus Torwalds, a Finnish student, announced on the Internet that he had began working on a new UNIX-like kernel for the standard PC. He published the source code and invited others to help him in the development. By combining this kernel with the GNU software a complete operating system was obtained. (There is now an all GNU kernel called Hurd.) The GNU/Linux system spawned a quite a huge following which gave a big push to the GNU principles.

The term *Free Software* was coined by Stallman. The term itself has led to some confusions and Stallman uses the expression*free as in beer or free as in speech* to emphasize the dual meaning inherent in the word free. Free means freedom and does not have anything to do with price. The basic principles for Free Software are (Stallman 1999):

- Freedom to run the software.
- Freedom to modify the software for your own needs.
- Freedom to distribute copies of the software, either gratis or for a fee.
- Freedom to distribute modified versions of the software so that others can benefit from your improvements.

Later on these principles had to be formalized, in order for the growing Free Software community to be able to work by the same rules. The principles followed the expression *copyleft, all rights reversed* as opposed to copyright and stated the rules for the use of Free Software with the purpose of keeping software code free and not privatized. Different types of licenses for different purposes developed out of this copyleft principle, the most common is the GNU General Public Licence (GPL) 5.4. The very concept of Free Software code is of course controversial and opposes the strength of patent and copyright laws that has developed.

Some of the people in the Free Software community were concerned that the current vision was too strict and anti-commercial and contributed to alienate the Free Software community from the rest of the industry. They wanted to make the Free Software concept more broadly available. In 1997 they took on a new term, Open Source Software (OSI 2004), and at the same time got away from the ambiguity of the word free. Open Source was defined slightly different, one of the differences being that it takes a more relaxed view on mixing Open Source and proprietary solutions.

Open Source Methods

The Open Source approach is community based. In a development project anyone in the project community can contribute to the software. Each contribution is then generally reviewed by a council or board, or sometimes by one leading developer as in the case of Linux, before it is decided what goes into the actual software.

This process of peer review is what makes scientific results and therefore also many mean open source software robust and reliable. Another important benefit is control. Companies using OSS have control over their software and have the freedom to modify it after their own needs. They do not have to be locked in with a vendor and limited by what they chose to offer. The open mentality also implies an honesty about errors. If one developer has a problem with something going wrong he will turn to the community to get help to get it fixed. This has built in a quality feature in the OSS culture. In proprietary software development the companies usually do not want to admit any flaws in their products. The advocates of Open Source argue that software development has to be free to maximize development and innovation. By not sharing the knowledge the society will not evolve and there would only be individual gains.

This community approach could not have existed without the development of the Internet. In fact the whole Internet evolved much by the same drivers as the OSS approach. Several collaboration tools exist that aid developers cooperating over the Internet. The

Internet also provides means for distribution. This helps another cornerstone in the OSS approach, namely what is referred to as *Release early and often*. By making the software available users can download it and helping in ironing out bugs.

Eric S. Raymond describes in an article titled *The Cathedral and the Bazaar* (Raymond 2000), two distinct types of development. The first approach is like the building of a cathedral, structured planning, carefully crafted and controlled by a small gifted group or a single individual. This is the classical way to approach a large and complex project. But with Linux and other Open Source projects came another way to handle the situation. He describes it as a great babbling bazaar where different ideas meet and melt together.

One big benefit of using the Free/Open Source model is the fact that there are large number of freely available libraries that can be used. So by using Free software, time and money can be saved instead of building from scratch or buying code.

Licenses

One way to make a program Free is to put it in the public domain without copyright. But this also allows others to convert the program to proprietary software. By placing the software under a license the freedoms are protected. Note that the copyright must be retained by the programmer or transfered to some other entity. It is only the holder of the copyright that can change the license. Here is a list of the most common licenses. The complete licence texts can be seen at (Free Software Foundation, Inc. 2004).

GPL

The GNU General Public License (GPL) is the most pure and most used of the copyleft licenses. It is the most restrictive. Basically there can be no mixing with non-free code. If a piece of software relies on code under the GPL, then the whole software system must be put under the GPL. Any modified GPLd source must be made available.

LGPL

The GNU Lesser General Public License is used for some libraries. Theses libraries may be linked to by proprietary code. The use of this license is motivated when there are several proprietary alternatives. By allowing linking, the software has a greater chance of gaining wide use.

BSD

The BSD (Berkely Software Distribution) license is used for all software in the BSD UNIX dialects. It is very permissive and allows for making modifications without publishing them. The original license included an advertisement clause. Wherever software that included BSD code was advertised, a specific sentence had to be included. This clause has for practical reasons been removed. The BSD license (without the advertising clause) is not copyleft, but is compatible with the GPL.

5.5 Debugging

Software debugging is a valuable Tools for debugging has been around for a long time. However, there are several problems associated with current debugging tools. First there is the issue of learning how to operate them. It is not very easy which results in that programmers often do not use the tools. Second, many problems cannot not be solved using currently available tools. This is especially true when dealing with distributed or parallel software.

GDB

GDB, the GNU Debugger, is the most commonly used tool for debugging, at least on UNIX-flavored systems. The user executes his/her program *inside* GDB. It is possible to set break-points, examine variables and registers and even change values. Mostly, it is used for just finding what line in the source files causes a crash.

GDB supports C, C++, Fortran, Java, Chill, assembly, and Modula-2. There exist graphical user interfaces built on top of GDB. Examples are KDbg and DDD (Data Display Debugger). GDB is licensed under the GNU General Public License.

Valgrind

Valgrind is a fairly new tool that aids in finding memory-management problems in programs. When a program is run under Valgrind's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted. Thus, problems such as the use of uninitialized memory, reading/writing memory after it has been freed, and memory leaks can be effectively detected.

Problems like these can be difficult to find by other means, often lying undetected for long periods, then causing occasional, difficult-to-diagnose crashes. Valgrind can be used in combination with GDB. Valgrind is licensed under the GNU General Public License.

Logging

Logging is often used for debugging but can be used for other purposes as well. E.g. the output from a computational unit can be logged in order to validate the calculations. Another use is to record data for use in future simulations. These different cases requires different kinds of logging output. This needs to be considered when designing logging mechanisms.

There are a number of ready-made tools available for logging. In Java there is the widely used *Log4J*. Similar packages for C++ exists called log4cpp and log4cplus.

5.6 Documentation

Documentation is the Achilles heel of programming! Many great programs have been written that lacks even rudimentary documentation. This is mainly due to the fact that writing documentation is considered much more boring than writing code. Only in commercial contexts where people are hired strictly for that purpose, is the situation better. Another big problem is that of keeping the documentation current with programs and code. There are several levels of documentation, relating to the levels of programming 5.1.

DoxyGen

Doxygen is a documentation system for C++, Java, IDL (Corba, Microsoft and KDE-DCOP flavors) and C.

It can help you in three ways:

It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyper-linked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code. Doxygen can be configured to extract the code structure from undocumented source files. This can be very useful to quickly find your way in large source distributions. The relations between the various elements are be visualized by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically. You can even 'abuse' doxygen for creating normal documentation (as I did for this manual). Doxygen is developed under Linux, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, an executable for Windows 9x/NT is also available.

5.7 Version Control Systems

A version control system is crucial for software development. Especially when several developers are involved. Its function is to manage multiple versions of files. These versions are stored in a central repository. Whenever a developer joins a project, he/she *checks out* a private version which is stored locally. After modifications to the code have been done, the developer then *commits* the modified files along with a log message. The central repository is then updated.

This way any changes can be tracked, and rollbacks can be carried out if the modifications later are regretted. Several developers can also work simultaneously on the same file. The system will automatically try to merge the files as they are checked in. As long as modifications have not been made on the same lines of code, there will be no problems. If conflicts arise they have to be sorted out manually.

Here a number of versioning tools are discussed.

RCS

The Revision Control System (RCS)(Barbera et al. 1984) was the first version control system and dates back to the eighties. RCS is rather simple and is hardly used nowadays. It is however still a valid alternative for single developer projects.

CVS

CVS is the most widely used versioning tool today. It builds upon RCS, providing additional features such as release management. It works well over the Internet. Several tools such as graphical interfaces exist such as *Cervisia*.

CVS works very well but has a number of drawbacks. One is that directories and file cannot be renamed. They have to be deleted and then added with a new name. Another problem is that several files and directories cannot be committed in one atomic operation. Log messages are stored per file instead of per commit.

Subversion

Subversion was started in order to address the drawbacks of CVS mentioned above. It can also use the HTTP-based WebDAV/DeltaV protocol for network communication and the Apache web server. It reached version 1.0 early 2004 and is still not considered mature by everyone.

Bitkeeper

Bitkeeper is a commercial product that received fame when Linus Torvalds started to use it for the maintenance of the Linux kernel source. In contrast to CVS and Subversion it is a distributed peer-to-peer system instead of client/server. The repository is stored in a database and is replicated among all peers. The name Bitkeeper refers to its ability to maintain data integrity. Other systems can have undetected file corruption.

Bitkeeper costs money but there is an alternative to use it at no cost. The condition is that all log messages are sent to a website where the messages are open to see for anyone.

5.8 Code Issues

There are number of issues that are important enough to be mentioned but do not warrant its own chapter.

Versioning

Version numbers of software releases can be handles in several ways. Commercial software often inflates version numbers in order to signal great enhancements that urges customers to upgrade. Open/Free software is on the other hand often too restrictive in terms of increasing version numbers.

Lately a standard has more or less emerged in this area. This scheme is e.g. used for the Linux kernel. The notation is *major level*, *minor level*, and *patch level*, like 2.4.27. A change in the major number means major changes. The minor number signifies added or changed functionality. The last number usually only means that bugs have been fixed. Only for a change in the patch level number can compatibility be assumed and that the API will not have been changed.

Language Bindings

Language bindings, or wrappers, means that a library can be used from a programming language other than the one used to build the library. This means that even though a project might decide to use a specific programming language, other languages can be used if language bindings are made available. The binding is said to be complete if every part of the API is wrapped by the language.

A common binding tool is the Java Native Interface(JNI) which helps to create Java bindings for a C library. Another useful tool is the Gnu Compiler for Java (GCJ). This makes it very easy to create Java bindings for C++ code.

Coding Guidelines

A project of decent size must decide on a set of guidelines for how the code will be styled. This will help programmers go into any code and quickly figure out what is going on. Mistakes in the code are also more easily spotted. The guidelines will dictate how classes, methods and variables should be named. This can e.g. be that class names should start with a capital letter, and that capital letters should be used as word separators. Of course these guidelines should be agreed upon early in a project before much code has been produced.

A related issue is that of namespaces. It should be decided early how namespace shall be used. For example whether a project should be put into one single namespace or if different parts should have its own or nested namespaces.

5.9 Lessons Learned

Here we summarize the most important aspects of the chapter.

- The concept of dividing programming into levels is very useful when constructing a programming framework.
- Modeling of system is preferably done using an UML tool.
- Using licenses and methods from the Open Source community brings many benefits.
- A version control tool is definitely invaluable.

Part II

A Design for Robotics Software

Chapter 6

Introduction

This part of the thesis shows in some detail the design and implementation of a proposed software framework for mobile robotics. Our main goal was to construct a generic framework for robotics that does not imply a specific architecture. This because we believe that one architecture can not be designed that fits any robot application. And had we restricted our work to a specific architecture, architecture research and experimenting would not be possible.

The findings as described in the first part of this thesis was used to design and implement a component framework (ORCA 2004). It was initially a part of an EU sponsored project (OROCOS 2002). The software has attracted roboticists from around the world, and has been successfully deployed in a number of projects. Additions to the framework has primarily been made at the Australian Centre for Field Robotics (ACFR). This will be indicated in the text where applicable.

6.1 Prerequisites

Before setting out to design a software system, a list of prerequisites must be defined. All to often, this step is overlooked. Either the list does not exist, or it includes everything imaginable. This is very important for the following reasons. Firstly, all people involved may not have the same goals or ideas as for what the design will be used for. Secondly, the choice of tools are a consequence of the prerequisites. E.g. CORBA might not be a good choice if the target is embedded hardware with low computing resources, while it may be an excellent choice on a standard PC. Thirdly, without clearly defining limitations, the unachievable goal of making a super general-purpose, good for everything kind of system, might be strived for, and thus never reached.

Domain and Application

The domain that the robots will be used in, is primarily a research environment with emphasis on flexibility, re-usability and scalability. Performance is not given as high a priority.

This contrasts to e.g. an industrial domain, where specialization, performance and production costs are highly important factors. Because of the chosen domain, a desirable property is that it should not be too hard to learn for a novice. However, a fairly high level of programming skill could be demanded from the programmer, corresponding to the education obtained by a master's thesis student.

Furthermore, the design is mostly aimed for robots operating in indoor environments, e.g. office, industrial or hospital settings. Applications in mind are of the type usually referred to as fetch-and-carry. This means that the robot should be able to navigate around and perform tasks according to commands in an interactive manner. Commands could be issued by a human operator e.g. via voice, web, PDA or keyboard. Example of tasks are:

1. take visitors on a tour of the environment
2. locate and bring back an object
3. patrolling
4. deliver messages
5. create a map of the environment

Platform

The chosen platforms are robots which typically consist of a base, a computer and sensors.

Base

The base has wheels and electric motors for drive and steer. A holonomic base is both common and desirable. The motors can be either of type synchro-drive as on the Nomadics Scout, or differential-drive as on the Nomad 200. On some robots, e.g. Nomad 200, a revolving turret is fitted on the base.

Computer

The computer is a PC or an industrial board. Specifications on CPU, RAM, disk etc are comparable to a normal desktop PC. The operating system (OS) is also of standard type. Linux is the obvious choice for reasons explained in 4.1. As manipulators are not considered in this design, real-time is not necessary.

Sensors

Standard sensors in service robotics include:

1. sonars
2. laserrangefinder
3. infrared (lidars)

4. cameras
5. bumpers

Sonars typically consist of 8 to 24 sensors each operating at a speed around 5 Hz. A laserrangefinder operates at speeds up to 500 kbs (USB interface). Bumpers operate by triggering discrete interrupts. Odometry requires at least 100 Hz in data resolution. Bandwidth requirements for the camera are much higher, typically around 50000 kbs and thus demands special attention regarding choice of communication mechanism.

License

The software will be made open-source (section 5.4) with a free license. This has impacts on the choice of tools that will be used since licenses can be incompatible. The choice was made to use LGPL (section 5.4) for libraries and GPL (section 5.4) for application code.

Programming Language

In general most OOP would be applicable for the design of a system. For applications such as robotics there is typically a need for I/O programming or access to APIs, which makes C or C++ an obvious choice. However, given a definition of generic APIs with language bindings, many different languages could be used, in particular for interfaces and reactive parts of a hybrid deliberative system.

Java might be an efficient choice for GUI programming, which functional languages such as Haskall and Lisp might be options for the deliberation part. However, C++ might be an operational choice for most of the functions considered.

6.2 Lessons from Part I

Here we recapitulate some of out conclusions from the first part of this thesis.

Architecture

We have seen that a number of traits have proven successful for systems, and that the lack of certain properties have raised hindrances. *Modularity*, *openness* and *extendibility* are indeed valuable assets. Another one is *simplicity*. There is a good rule of thumb called *KISS, Keep It Simple Stupid*.

No system has really gained wide use. We think this is because that most of them enforce too much a specific architecture. Player/Stage (section 3.8) is the exception and is probably the most popular around the world. It does suffer some drawbacks such as a client-server model. To create an artificial layering structure should also be avoided, as the placing of components can be ambiguous.

In order to enable an effective behavior based architecture, parallel execution of behaviors is necessary. This means we map the behaviors onto processes. This also increases vulnerability against crashes. Another (well-known) lesson is to avoid bottle-necks in order to facilitate scalability.

Providing a framework for hardware abstraction is important so that new hardware can be supported. One tricky part is to make assumptions about the hardware that are general but still non-complex. To enable interprocess communication and distribution of the system across several hosts, the communication framework needs to be powerful but simple.

Communicating patterns as those used in Smartsoft (section 3.7) are very useful in that they provide simple standardized methods for communication. Player (section 3.8) has a similar concept where devices are treated as UNIX files.

6.3 What We Are Aiming At

This part of the thesis where the design is described, is written in a bottom-up fashion. While a top-down approach would have been more pedagogical and better mirror the actual process, it presents difficulties. As every part builds upon, and is explained with terms described in lower levels.

To try to remedy this in some way, a simple graphical view (Figure 6.1) of a hypothetical implementation is presented here. This can be compared to the general hybrid architecture, Figure 2.1. By using our system and frameworks, this type of architecture should be easy to construct.

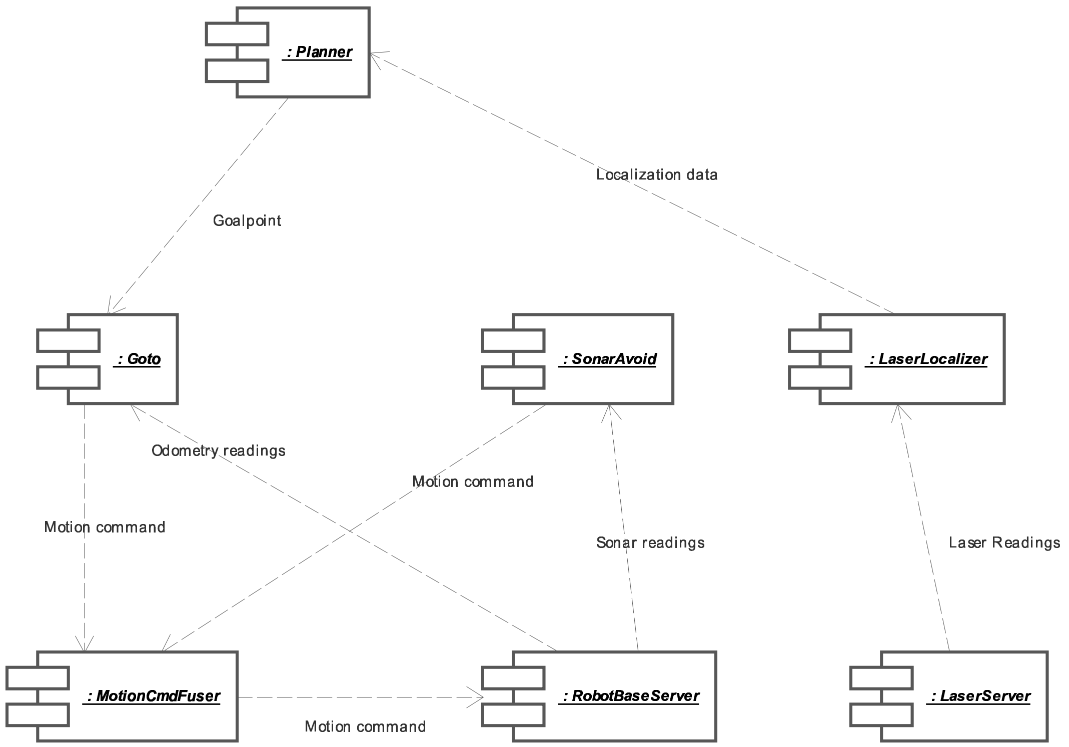


Figure 6.1: A simplified view of an example implementation. The arrows indicate principal data flow.

Chapter 7

Communication

7.1 Communication Toolkit

For reasons stated in section 4.3, CORBA has been chosen as communication technology. If the system had to be implemented on microcontrollers such as MC68332 with limited memory the choice of CORBA would have been less obvious. What remains is to choose the primary toolkit. This turns out to be quite a daunting task.

CORBA Implementations

The main factors to consider when making this choice is

1. popularity,
2. documentation,
3. adherence to CORBA specification,
4. available services.

There are quite a large number of toolkits available. Here is a list of the most popular.

1. ORBit
2. omniORB
3. TAO
4. JacORB
5. MICO
6. Numerous

ORBit is part of the GNU project and is supported by Redhat and Ximian. ORBit is used by the GNOME GUI desktop environment. This ORB is written in C which is also the primary target language. Mature bindings exist also for C++ and Python. At the time of writing, it is compliant to CORBA 2.4. ORBit2 is developed and released under GPL/LGPL.

TAO is based on the famous (and infamous) ACE (Adaptive Communication Environment) (Schmidt 1994). ACE is a framework that implements a number of standard communication patterns. It provides support for signals, events, IPC, concurrent execution and much more.

TAO is progressing quickly and is in wide use. It is however considered to have a large footprint in all aspects. The distribution is impressive but rather messy and hard to grasp. It provides a large number of services and supports almost all recent CORBA specifications. TAO is not GPL but released under a Free licence approved by OSI (OSI 2004).

OmniORB is a toolkit developed at AT&T Laboratories Cambridge. When AT&T closed down operation there in 2002, development ceased. OmniORB then became an independent project, but several of the original developers continue their involvement. It adheres to version 2.6 of the CORBA specification and is GPL.

JacORB is an ORB for JAVA. It is at 2.3 of the CORBA standard but also supports OBV. It is GPL.

MICO, acronym for MICO Is CORBA, is a rather popular implementation, especially in Germany, which is its native country.

TAO was finally chosen. Apart from the issues mentioned above, there was the fact of positive experience with using the toolkit ACE on which TAO is based.

Protocols

One serious drawback that choosing CORBA as transportation mechanism is the fact that basically all communication will take place using INET sockets. INET (Internet) sockets work between processes on different hosts as well as on a single host. This is of course practical when dealing with a distributed system. But if all communicating processes run on the same machine, there are more efficient methods of transport. Both UNIX sockets and shared memory provide more throughput and should be used when running bandwidth-intense communication on one host. For instance vision applications needs this, where large amount of video-data need to be transported fast between processes.

ISR (section 3.6) automatically identifies as a link is being setup, if the communicating processes reside on the same host or not. If they do, UNIX sockets are used, otherwise INET. This is totally transparent to the user. TAO provides something called *pluggable protocols*. This is meant to be something similar, and can switch between INET, UNIX, shared memory, SSL, UDP/IP and unreliable multicast. Our plan is to utilize this when the need arises.

Ideally, since CORBA is a standard, one should easily be able to switch toolkit later without having to make changes to the code, but in practice it is quite a bit of work. All toolkits vary slightly on the syntax.

7.2 Transportation Data-format

There are a number of options to choose from concerning what to actually transport across the wire. The choices are sometimes stipulated by the choice of communication toolkit. When communicating across different platforms, the endian problem must be handled. In Java, basically all class-objects can be transported thanks to the serialize method. When using pure socket-based schemes, strings or structs are common. XML is used in conjunction with SOAP (section 4.3).

Recent CORBA standards include a new construct called *valuetype*. Unlike traditional CORBA calls which are of type copy-by-reference, the valuetype is copy-by-value. This means that the data is copied across the wire. Valuetypes also have properties that makes them easy to use together with XML (OMG 2003). Some people find valuetypes hard to use (they resemble C++ autopointers). Nevertheless, the choice was made to use valuetypes for this project.

In order to be able to use the same IDL-file and underlying code regardless of what objects will be transported, single ancestor was made. It was called `OrcaObject` and has only a few basic operations.

```
valuetype OrcaObject {
    void dump();
    public ORCA::TimeValue creationTime;
    public string origin; // name of sender
    factory create_default(in string name);
    factory create(in string name, in ORCA::TimeValue tv);
  };
```

Based in this object, several standard objects suitable for robotics were constructed, such as `Pose2D`, `MotionCommand` and `SonarRangeReadings`.

7.3 Communication Patterns

A communication toolkit usually provides generic and often low-level communication mechanisms. So after choosing which toolkit to use, decisions have to made whether rules and mechanisms need to be constructed on top of that toolkit. There are a number of issues that come into play here. The advantages are

1. Freedom for developers is restricted
2. Easier to learn for novice users
3. The name of a remote calling method can usually be guessed
4. The remote calls can be reimplemented

The restriction on freedom may sound like a bad thing, but frameworks are actually all about the restricting of freedom. Szyperski puts it well: "An important role of a framework is its regulation of the interactions that the parts of the framework can engage in. By freezing certain design decisions in the framework, critical inter-operation aspects can be fixed. A framework can thus significantly speed the creation of specific solutions out of the somewhat semi-finished design provided by the framework." (Szyperski 1998)

One could of course choose not to add a level on top of the toolkit. In the CORBA case, this means that developers who construct new components just creates a new IDL-file (see section 4.3) and can choose whatever name and parameters he or she wishes. This will most likely lead to a disparate and heterogeneous system.

Callbacks are a common way to handle communication between components. Callbacks have however received a lot of criticism. One problem is that the server cannot know if a registered client is still alive at the time of delivering a callback. Another one is scalability (see e.g. (Henning & Vinoski 1999)).

The ORCA Patterns

After evaluating the patterns used in Smartsoft 3.7, a smaller subset was chosen. The names were also changed.

- Push (AutoUpdate)
- Query
- Send (Command)

The Event pattern was removed as seen redundant, the Push pattern could be used instead. The Configuration pattern was regarded as too much architecture specific, and could be implemented with Send and Query.

The patterns were implemented using C++ templates. The reason for this was to enable different classes to be transported using the patterns.

Push

The Push pattern is convenient to use when a component is sending data continuously to another component. This is often the case in a reactive loop where a sensor is updating a behavior. Figure 7.1 shows in UML the PushServer class. The methods `start()` and `push()` are called by the hosting object while subscribe and unsubscribe are called by the client over the network.

The client side proxy is shown in Figure 7.3. Here we recognize subscribe() and unsubscribe() that are directly routed to the server. The method push() is called by the server delivering new data. The method getUpdate() is called by the client in order to get the latest available update. It blocks if no update has been received yet.

The method getUpdateWait() is like getUpdate(), but blocks until new data has arrived. The method getUpdateWaitMax(const long msecs) is also like getUpdate(), but blocks until

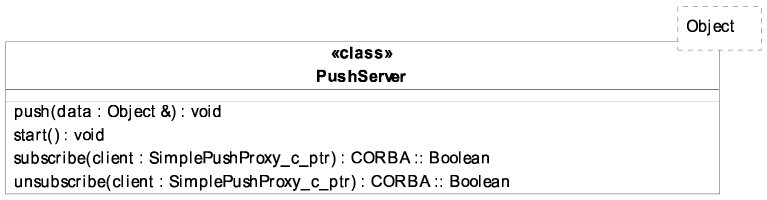

Object
«class»
PushServer
push(data : Object &) : void
start() : void
subscribe(client : SimplePushProxy_c_ptr) : CORBA :: Boolean
unsubscribe(client : SimplePushProxy_c_ptr) : CORBA :: Boolean

Figure 7.1: The PushServer class.

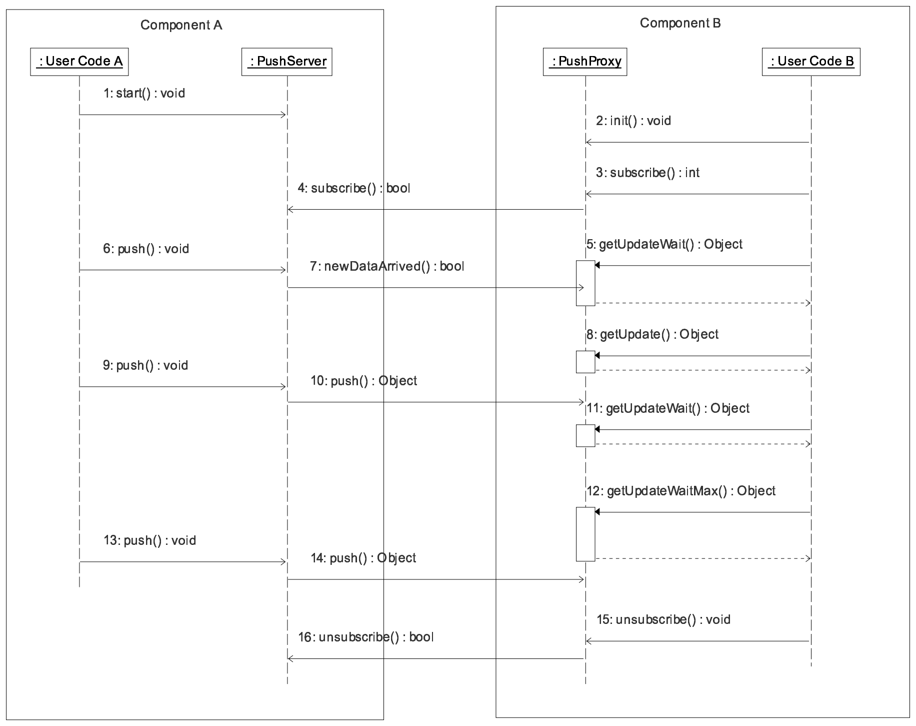


Figure 7.2: A sequence diagram showing the Push pattern.

new data has arrived or the specified time has elapsed. The method newDataArrived() can be used to check if new data has arrived from the server.

Figure 7.2 shows a sequence diagram of the push pattern in operation. Note that in sequence 8, the getUpdate call returns the same object as in sequence 5. Note also that in sequence 12, the maximum time has expired and no object is returned.

The push pattern should be extended to take an additional parameter defining the interval at which the client would like to be updated. To implement this on the serverside is

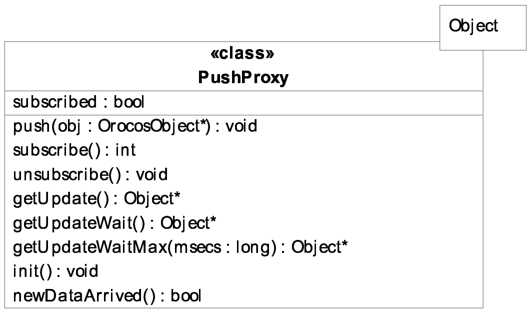


Figure 7.3: The PushProxy class.

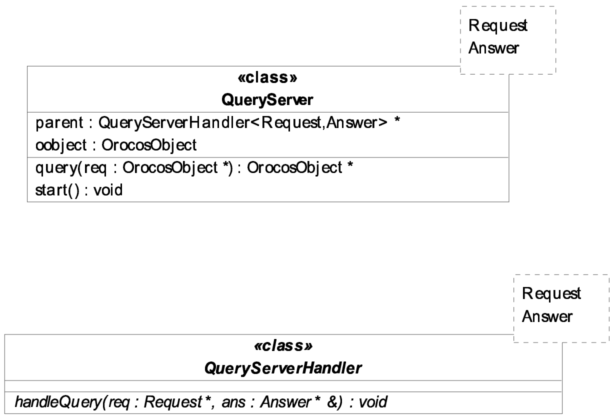

Request
Answer
«class»
QueryServer
parent : QueryServerHandler<Request,Answer> *
oobject : OrocosObject
query(req : OrocosObject *) : OrocosObject *
start() : void
Request
Answer
«class»
QueryServerHandler
handleQuery(req : Request *, ans : Answer * &) : void

Figure 7.4: The QueryServer and QueryServerHandler classes.

however not without problems. This since the control-loop runs at a more or less fixed frequency ultimately adapted to the hardware. A solution is to let the clients pick a multiple of this frequency. The clients must then be provided with this information beforehand.

Query

The Query pattern is used when a component needs to make a *one-time* request for data from to another component. It can of course be used several times, but for continuous updating, the Push pattern is a better choice. A UML class diagram of the QueryServer can be seen in Figure 7.4. The method query() is the only one visible by the client. As can be seen, it also takes an argument that can be used to parameterize the query. The server component implements the class QueryServerHandler to which the query is routed.

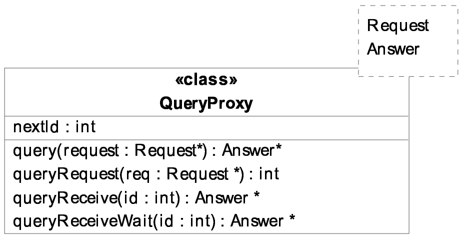

Request
Answer
«class»
QueryProxy
nextId : int
query(request : Request*) : Answer*
queryRequest(req : Request *) : int
queryReceive(id : int) : Answer *
queryReceiveWait(id : int) : Answer *

Figure 7.5: The QueryProxy class.

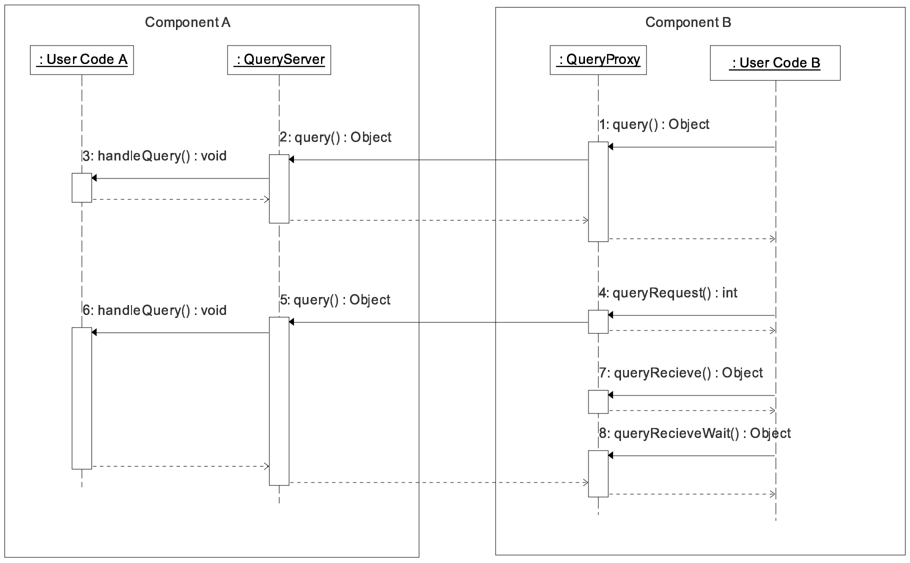
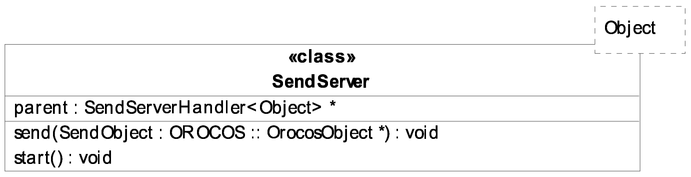


Figure 7.6: A sequence diagram showing the Query pattern.

On the client side (Figure 7.5), the query() method is straightforward, but the others need clarification. They are used for sending asynchronous queries. The method queryRequest() returns an ID that is than used for picking up the answer with queryReceive() or queryReceiveWait(). The latter one is blocking until the response is available.

Figure 7.6 shows a sequence diagram of the Query pattern. Sequence 1-3 shows the regular blocking query. Sequence 4-8 shows the non-blocking version. Note that sequence 7 returns without an answer, while sequence 8 waits until the answer is available.


Object
«class»
SendServer
parent : SendServerHandler<Object> *
send(SendObject : OROCOS :: OrocosObject *) : void
start() : void

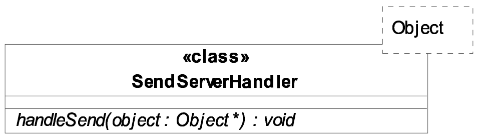


Figure 7.7: The SendServer and SendServerHandler classes.

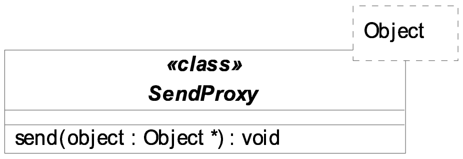


Figure 7.8: The SendProxy class.

Send

The Send pattern is a one-time sending of data from one component to another. The pattern does not include the receiver requesting the transfer afore hand. Send is the simplest pattern. On the server side (Figure 7.7), the component implements a SendServerHandler that receives the object. The client interface just consists of the method send(). Figure 7.7 shows the simple sequence diagram for the Send pattern.

One should note here that in the Send pattern, the client sends data to the server. Here the client-server terminology may seem confusing.

7.4 Synchronization

A problem that a developer who designs an asynchronous (push-based) behavior-based systems faces, arises when fusing the output from different behaviors (Figure 7.10). Usually the behaviors operate at different frequencies and thus push their data at different points in time and with varying interval.

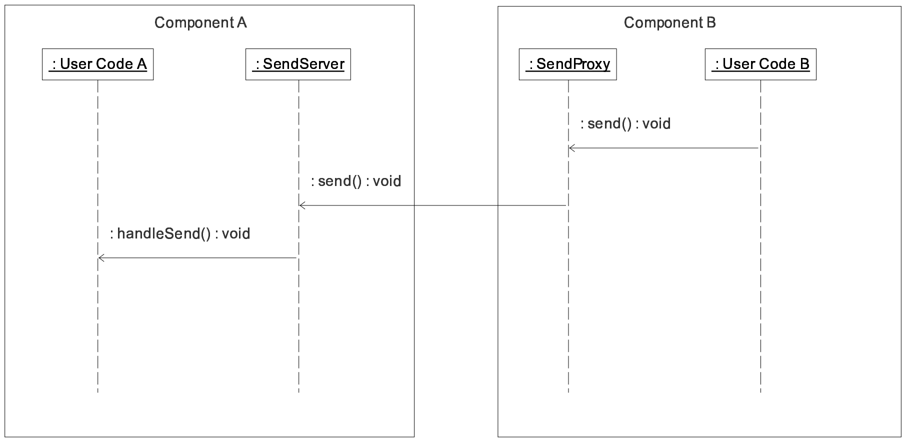


Figure 7.9: A sequence diagram showing the Send pattern.

This can be handles in different ways.

- Wait until all behaviors have delivered data, then fuse and send.
- For every incoming data, fuse with the latest available outputs from the others and then send.
- Have a global clock or scheduler, making it a synchronous system.

The first option is a good choice if the timings are modestly disparate. If the timing differences are greater, the second option may be more suitable. In ISR (section 3.6), this scheme is used rather successfully in all cases. The last option is of course more drastic. Abandoning the asynchronous model does have advantages but also adds to complexity performance loss.

To aid in this area, a Synchronizer class was designed (see Figure 7.11). The component developer instantiates an object of this class and calls the addSubject() method for each proxy that should be synchronized. The method `wait(subject)` is then used for waiting for a specific server. The method `waitAll()` is used for waiting until all servers have pushed new data. Both of these are blocking and will wait indefinitely. The method `waitAll(msecs)` can be used to define a maximum amount of time to wait.

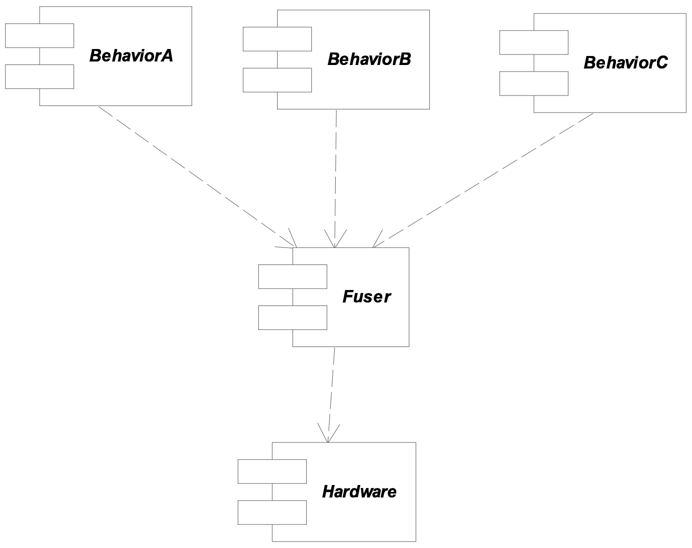


Figure 7.10: Synchronization situation.

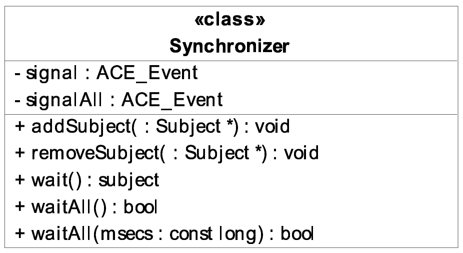

«class»
Synchronizer
- signal : ACE_Event
- signalAll : ACE_Event
+ addSubject(: Subject *) : void
+ removeSubject(: Subject *) : void
+ wait() : subject
+ waitAll() : bool
+ waitAll(msecs : const long) : bool

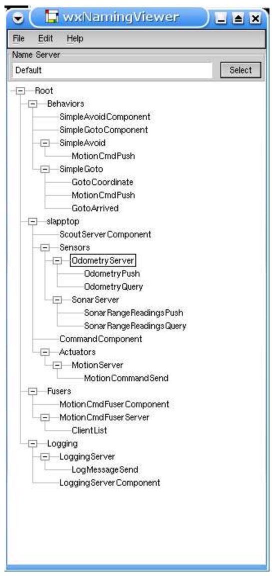
Figure 7.11: The Synchronization class.

7.5 Services

Components typically make use of external services in order to locate other components. Also other services exist, such as the Time, Scheduling and Security services.

Naming Service

As stated in section 4.4, the COS (CORBA Services) Naming Service is the service through which most clients of an ORB-based system locate other objects. It corresponds to a telephone books white pages, clients are stored and retrieved by their name. The data can be structured into hierarchies, very similar to the directory structure in file systems. The entity corresponding to directories are called contexts. Clients can query for names and even navigate these naming contexts. This gives you the choice on how to organize the



```
Root
  Behaviors
    SimpleAvoidComponent
    SimpleGotoComponent
    SimpleAvoid
      MotionCmdPush
    SimpleGoto
      GotoCoordinate
      MotionCmdPush
      GotoArrived
  slapptop
    ScoutServerComponent
    Sensors
      OdometryServer
        OdometryPush
        OdometryQuery
      SonarServer
        SonarRangeReadingsPush
        SonarRangeReadingsQuery
    CommandComponent
    Actuators
      MotionServer
        MotionCommandSend
  Fusers
    MotionCmdFuserComponent
    MotionCmdFuserServer
      ClientList
  Logging
    LoggingServer
      LogMessageSend
    LoggingServerComponent
```

Figure 7.12: A view of a running system of components as registered with a Naming Service.

structure. Here we found that a functional, rather that physical hierarchy is most often desired. Only where physical location is of importance should the physical view be used. For example, behavior components can be put in a top-level context, while sensor-server components should be placed in a context relating to the platform on which the sensor is located. This because the physical location of sensors are important. Figure 7.12 shows is a graphical view of a running ORCA system as registered with the naming-service.

Support for the Naming Service was considered fundamental, and was thus built into our system. This to the extent that actually no component can be started without the presence of a running Naming Service. Some people will undoubtedly object to this.

Trading Service

The COS Trading Service (see section 4.4) is a convenient way for components to find other components that offer a particular service. By using a Trader Service, specific names of other components need not be known which enhances flexibility. It also increases robustness since if a server has gone down, another server offering the same service can be located. It was concluded that the Trading Service is mainly useful in large complex systems with a large number of components. Support for it should be included but with a lower priority.

Event Service

The COS Event Service (section 4.4) seemed to be the perfect mechanism to implement the Push pattern (section 7.3). It turned out however that our choice of valuetypes (section 7.2) was not compatible with the current TAO implementation of the Event Service.

So the Push pattern was eventually rewritten from scratch. TAO also offers other flavors of the Event Service, notably the Real-time Event Service. There is also a Fault Tolerant Real-time Event Service. Such options are a natural choices for consideration in future work.

Logging Service

Mainly for debugging The CORBA Services includes a Telecom Log Service. It can be used for logging and monitoring the execution of a running system. This can also be used for pure debugging of applications. This is especially helpful, since finding bugs in distributed systems can be extremely difficult. In our system, we ended up implementing our own logging service which was integrated with the Log4cpp.

7.6 Summary

Here is a summary of the decisions made in this chapter.

- CORBA was chosen as communication technology.
- TAO was chosen as CORBA toolkit, primarily because it is up tp date regarding standards.
- The CORBA *valuetype* was selected as transportation data-format in order to enable copy-by-value.
- Three communication patterns were defined; *Push*, *Send* and *Query*.
- A Synchronization class was designed to assist when a component recieves data from several other components asynchronously.
- The Naming Service was chosen as primary mechanism for components locating other components.
- The Trader Service should be supported at a later stage.

Chapter 8

Ontology for Mobile Robotics

The data-types used for robotics play many important roles. The way they are chosen and organized

- structures and organizes *things* into a hierarchy,
- defines the common set of information that is passed around in the system,
- implicitly conveys the designers view of the world.

This chapter presents shortly related work, and then a chosen ontology is presented.

8.1 Related Work

In some scientific disciplines, tremendous work has been put into defining a cohesive standard of data-types. One example is the Image Understanding Environment (IUE), which is an object-oriented class library for use in computer vision. It is implemented in C++ and contains a large number of classes.

In robotics, so far no serious attempt has been made to form a standard set of types that the community can agree upon. Recently a number of initiatives has been started. One is called Kinematics (Bruyninckx 2003) and is developed in relation to the OROCOS project (OROCOS 2002). The aim is to develop application-independent libraries for the geometry, kinematics and dynamics of robots, which covers a large area of the robotics field.

Another newly started project is RETF (RETF 2003). It is a coalition of industry and academia that strives to produce and maintain specifications for reusable, interoperable building blocks for mobile robots. This also includes defining relevant data-types. It is also at a very early stage but seems to have stalled already.

JAUS (Joint Architecture for Unmanned Systems) is an American military project. It has been criticized for imposing architecture and protocols and to be very cumbersome to implement.

There are many successful libraries but definition of ontologoes for broad domains such as robotics have so far not been as successful. They often fail for a number of reasons. The number one problem is that they set the scope too wide, they try to cover every conceivable area. This results in very large libraries and documentation. This make users hesitate to install them and gets hard to manage. Getting many people from different sub-areas of robotics to agree on common terminology and classification is not also an easy task. So the standards often gets rewritten over and over again. And sometimes the projects do not reach the implementation stage at all.

8.2 A Proposed Ontology

The types normally needed for mobile robots can be divided into six areas: *Geometry* ,*World geometry* ,*Time* ,*Navigation* ,*Graph*, and *Estimators*. The rest of this chapter presents in a rather ad-hoc way a proposed ontology. The last two of the above areas are not addressed.

Choice of units is a very important factor. The obvious choice here is SI units, i.e. meters, radians etc. Traditionally other units have been common in robotics, such as millimeters, inches or tens of inches. This has caused a lot of headaches when conversions have had to be done.

Geometry

These address the standard mathematical geometric primitives. They form the basis for the subsequent types.

```
Geometry::

        typedef float Matrix22[2][2]
        typedef float Matrix33[3][3]
        typedef float Angle
        typedef float Distance
        typedef float Point2d[2]
        typedef float Point3d[3]

        typedef Angle Rotation2d
        typedef Angle Rotation3d[3]

        Line
        Plane
```

World Geometry

The World Geometry types differ from the Geometry types in the sense that they contain uncertainties. But just as with the Geometry types, they are primitives for the higher level

```
Position2d
{
    geometry::Point2d position
    Matrix22 uncertainty
}
Position3d
{
    geometry::Point3d position
    Matrix33 uncertainty
}

Orientation2d
{
    geometry::Rotation2d ori
    float uncertainty
}
Orientation3d
{
    geometry::Rotation3d ori
    Matrix33 uncertainty
}
WDistance
{
    geometry::Distance distance
    float uncertainty
}

Velocity

Acceleration

Translation
```

Time

The single Time type contains the SI unit second, as well as microsecond. Since these are used in UNIX operating systems, conversion will not be necessary.

```
TimeValue
```

```
{
    long seconds
    long useconds
}
```

Navigation

These reflect the higher level types that are composed of the types defined above. These can be implemented as classes or structs. These containers are then what is passed around in a system.

Some of these might need explanation. *Pose* is a common term for position estimation which is used e.g. for describing the position of a robot. The three dimensional variant is used for flying or underwater vehicles, but also for ground vehicles where altitude is regarded. A Pose is always used in conjunction with a timestamp. Rangereadings are the data provided by certain sensors such sonars and laserrangefinders.

A Goalpoint is used e.g. when instructing a robot to navigate to a point. A Goalpoint needs to have a name so referring to it is made easier for human operators. The Door is a Goalpoint with some added properties such as which way the door opens.

```
typedef sequence<worldgeometry::WDistance> RangeSequence

Pose2d
{
     geometry::Point2d position
     geometry::Rotation2d rotation
     time::TimeValue timestamp
     Matrix33 uncertainty
}
Pose3d
{
     geometry::Point3d position
     geometry::Rotation3d rotation
     time::TimeValue timestamp
     Matrix66 uncertainty
}

typedef sequence<Pose3d> PoseSequence

RangeReading
{
    RangeSequence range
    pose3d pose
}
```

```
MovingRangeReading : RangeReading
{
    PoseSequence poseSeq
}
GoalPoint
{
    Pose3d position
    string name
    worldgeometry::Orientation3d orientation
    worldgeometry::WDistance reachRadius
    boolean useOrientation
}
Door : GoalPoint
{
    worldgeometry::WDistance doorWidth
    worldgeometry::WDistance doorHeight
    worldgeometry::Orientation2d leafAnglge
    boolean doorLeafExists
    boolean rightSideHinges
    boolean opensOut
    boolean passable
}
```

8.3 Objects

As stated in section 7.2, CORBA valuetypes was chosen as format of the objects that are transported across components. While the previous section forms a theoretical guideline, this section describes the actual implementations of objects in the system.

The OrcaObject

A common superclass called **OrcaObject** is defined that is subclassed by all other object types. This simplified the implementation of the communication patterns 7.3. It was also inspired by the java.lang.Object which is the root of the class hierarchy in Java.

When constructing a superclass you generally need to think carefully because you want to keep it small and not change it frequently. The following attributes were included in the object:

origin A string containing the name of the component from which the data originates.

creationTime A timestamp relating to when the object was created.

copy_value() Creates and returns a copy of this object.

dump() Prints out the values.

The *creationTime* attribute is itself an object called **TimeValue** which includes an attribute for seconds and one for microseconds, as defined in the previous section. Many other objects include additional timestamps which may seem strange. The motivation is that the creatationTime reflects when the valuetype was constructed, which in general is right before the object is transmitted. This can be used e.g. to measure communication latencies. The other timestamp can e.g. be used for representing the time at which a sensor-value is retrieved. Such a timestamp should be used when measuring the age of sensory data.

Navigation

Figure 8.1 shows the object hierarchy for navigation.

A **Point** correlates to the Point3D as defined above.

The **WorldCoordinate3D** is an implementation of the Position3D above. It includes a Point, an Gaussian uncertainty vector and short-hand functions for getting x, y and z.

The **WorldCoordinate6DOF** extends the above for six degrees of freedom. It adds a Point representing the 3D orientation, and short-hand functions for getting the pitch, roll, and yaw.

RobotPose is an object that can be used for describing a robot position. It includes the orientation of the robot, the translational velocity, the steering velocity and a timestamp from when the data was retrieved from the hardware.

Odometry contains a position x, y, steer angle, translational velocity, steering velocity, and a timestamp. Whether the position and steer angle should be included could be debated, but at least one robot platform supplies this by the hardware.

RangeReadings contains a range of values. The size of the range is given by nrOfReadings.

SonarRangeReadings extends the above object by adding a vector of poses as all scans cannot be regarded as fired from one place. It also adds a vector of timestamps for the same reason.

LaserRangeReadings only adds one timestamp as the readings are treated to originate from the same point in time and space.

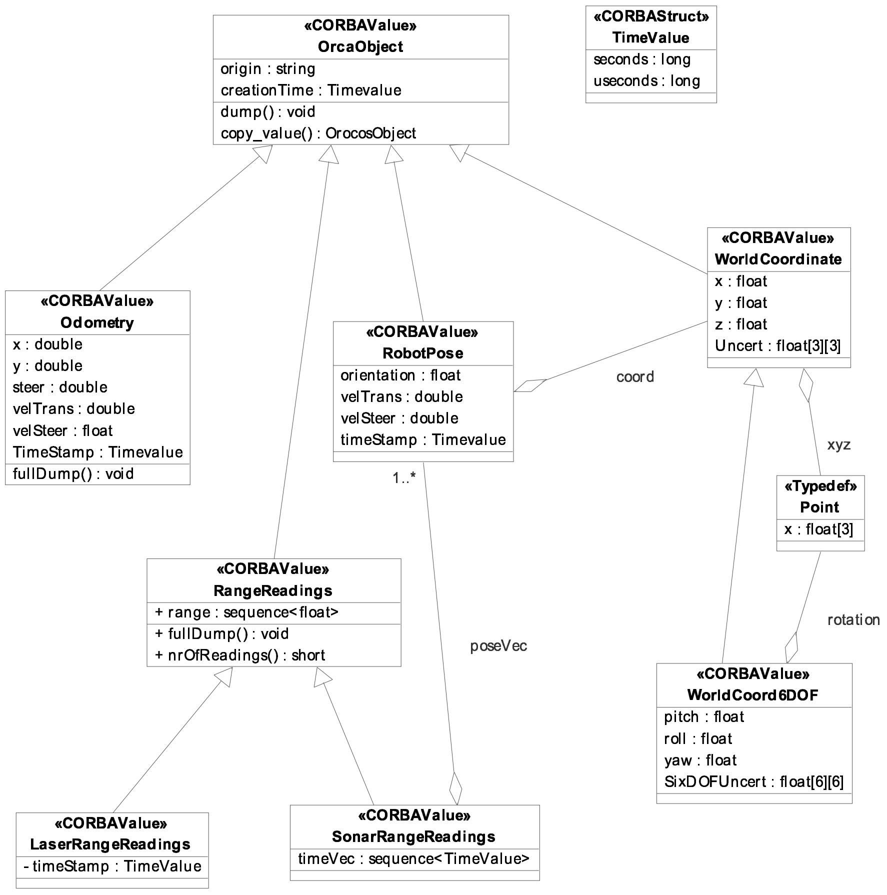


Figure 8.1: A view of object types used for navigation.

Hardware

Figure 8.2 shows the object hierarchy for hardware representation.

Robot is composed of actuators and sensors. Additionally it includes fields for name, vendor, model, radius, bumper radius, and name of hardware library.

Actuator includes fields for vendor, model, library, driver and port.

Drive extends the above object by adding type, maximum speed, maximum steering speed among others.

Sensor also includes vendor, model, library and port, but also update interval. There is also a field which is used to state the type of the sensor.

SonarRing extends the above object by adding the number of sonars and also other technical details.

LaserSensor adds technical details relating to that type of sensor.

Navigation

Figure 8.2 shows the object hierarchy for navigation.

GoalPoint contains a name, a WorldCoordinate. A field called reachRadius specifies how close to the point one must go in order to succeed in reaching it. A radius of zero would effectively make a robot move around it indefinitely. If an angle is given, it means the robot should face this direction after it has reached the goal. There is a tolerance to this called reachAngle.

Door extends the above object with fields relating to a door such as width, height, and leaf-angle. There are also a number of boolean values representing if the door is passable, which side the hinges are on etc.

MotionCommand can be used in two ways. Either the desired direction or the desired directional velocity is given. In both cases the desired speed and weight are given. Weight signals how much this motioncommand should be considered when fusing it with other motioncommands. This object can be of two more types, DONTCARE and FULLSTOP. The former means that this motioncommand should be disregarded (weight=0) and FULLSTOP of course means that the robot should stop immediately.

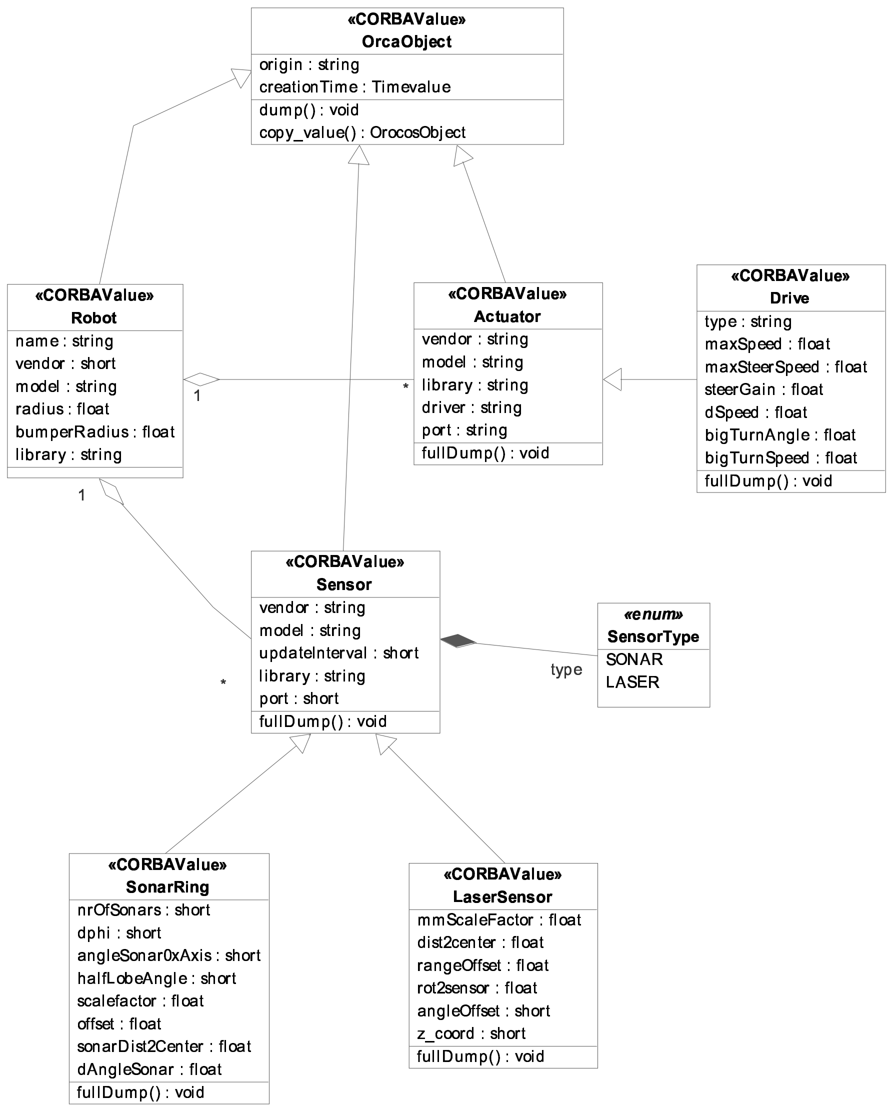


Figure 8.2: A view of object types used for representing hardware.

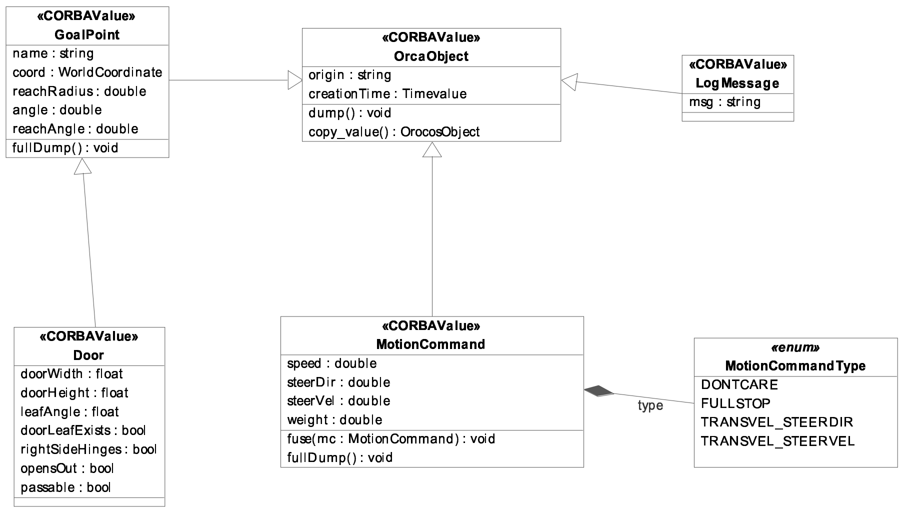


Figure 8.3: A view of object types for navigation.

Chapter 9

Component Model

In order to ease the development and interaction between components, a component model need to be defined. As stated previously, this is usually not something that the middle-ware software provides. According to (Wang et al. 2004), the solution is to define a *component middle-ware* with the following capabilities:

- Creates a standard "virtual boundary" around application component implementations that interact only via well-defined interfaces.
- Define standard container mechanisms needed to execute components in generic component servers.
- Specify the infrastructure needed to configure and deploy components throughout a distributed system.

Recent CORBA standards does however include a component model called CCM. There have not been any implementations available until December 2003, long before our system was designed. The above points have been addressed in out system, this is explained in this chapter.

9.1 Fundamental Properties

The classic division of components into clients and servers is not really applicable. A component can be both server and client at the same time. For example, a behavior component can be a client of a sensor-server and at the same time a server for an actuator-component. So a system of *equal peers* is what we want. Moreover, our philosophy is based on that the components in the system work very independently. This means that as far as possible, no central component governs explicitly the execution of other components. The motivation for this is

- Minimize bottlenecks or single points of failure (example of this can be found at the end of (Lucas 1999)).

- Reduce number of communications.
- Reduce resource allocation.

A system should in principle be able to run indefinitely without a restart. Furthermore, we often have a dynamic system where we will not know from the start which components may be started. What we do know is that we have limited resources in terms of memory, cpu etc. This leads us to a conservative approach regarding resource handling. Ideally, all components not needed should shut down completely in order to release all its resources. Alternatively, they should release as much resources as possible that are not in use and then be suspended.

There are however drawbacks to this scheme, one of which is efficiency. A central component that has a notion of global state could prepare other components that will come in use in the near future. This would matter if there is latency involved in initiating e.g. a piece of hardware. But with our scheme, a component may restart immediately after shutdown.

A trade-off would be to flag components that can be expected to be used frequently and that should not shut themselves down when inactive. This however adds to the complexity.

9.2 Granularity

When dividing a system into components, granularity becomes an issue. TeamBots (section 3.5) is an example of a system with fine granular structure, while ISR (section 3.6) is more coarse-grained as indeed is common in behavior based systems.

The latter scheme was chosen for our design for a number of reasons. Components are by definition binary (see section 4.3) and thus fairly coarse-grained with one well-defined purpose. In a behavior-based architecture, behaviors map naturally onto components. A behavior then is a full blown process and the operating system takes care of scheduling and parallel execution. And if such a process for some reason crashes only one activity will be effected.

9.3 The OrcaComponent

A standard component skeleton, or wrapper was created called OrcaComponent. This includes code that initializes all CORBA communication. It also registers the component with the Naming Service (section 7.5). Implementation-wise the OrcaComponent is a class which all components will inherit.

Another utility class called ComponentStarter is templated with the actual component. It takes care of parsing the configuration file, parameterizing the component code and then setting the component in an initial state.

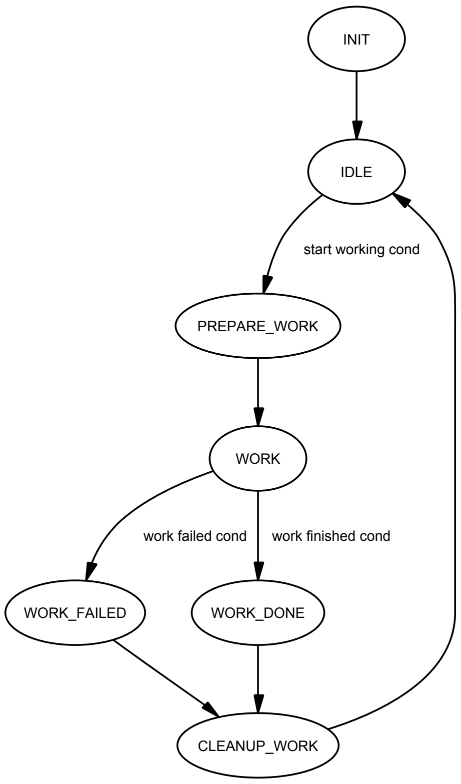


Figure 9.1: The state-machine used.

9.4 State-Machine

State-machines in various forms have always been present in robotic systems. There are usually states defined on different levels in a system. There is often one global state for one or more robots. Subsystems and components can also have its own internal states. This section deals with the latter.

For this project we decided to build upon successful experiences with the BERRA (section 3.6). The result can be seen in Figure 9.1. This state-machine has proven to work fine with behaviors as well as with sensor-servers. Changes between states are controlled by testing specific conditions. This testing is triggered by events such as clients connecting. Implementation-wise, each state corresponds to a method-call. The method-call `work()` returns a boolean. If it returns `true`, then the state WORK_DONE is entered. Otherwise the state WORK_FAILED is entered. The INIT and EXIT (missing from the picture) states are actually not explicitly implemented. The object's constructors and destructor can be used instead.

As stated in section 9.1, the high degree of independence of components, implies that

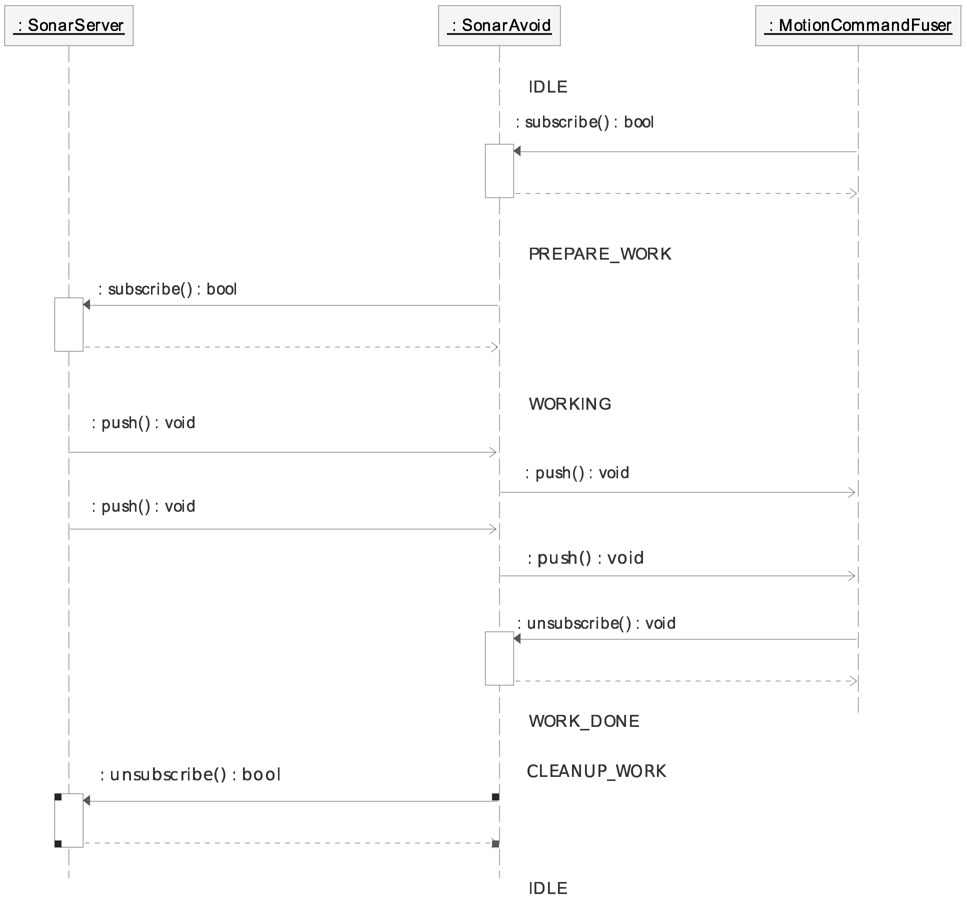


Figure 9.2: Statechanges (shown for SonarAvoid) occur automatically upon clients connecting

a component decides for itself when to change its internal state. This is for example in contrast to the principle in SmartSoft (section 3.7). Figure 9.2 shows how the behavior SonarAvoid, the middle component, changes state automatically when a client starts to subscribe to its output data. An connection thus often results in several other connections in a cascade-like manner.

9.5 Configuration

Configuration is another important topic. A system needs to be configured on different levels, on a global level as well as per component.

Initialization

This can be done in several ways:

- Ordinary text-files.
- Database.
- XML-files.
- Command line arguments.

Ordinary text-files is probably the most common way. One problem with those is that the parsing is error-prone and has to be custom made. Checking the format and syntax is also difficult. These problems are solved by using the XML format. While this is not the golden axe that many would have you believe, it does provide standards and tools that make it a good choice for use in configuration files. However, it is highly flexible and leaves great freedom to the designer.

Usage of databases for configuration within robotics is an interesting approach that have gained some popularity recently. This has many advantages but also disadvantages such as added complexity and reliance on an executing database program. For this project we chose to use XML.

Decisions have to made on what should be configurable. The obvious option is to hard-code the parameters into the source code. This implies of course a recompilation if an event must be changes, and should thus be avoided. But there is no reason not to hard-code values that can be expected to remain constant. Examples could be physical properties.

Since it can be hard to predict what parameters can be useful in the future, an open-ended format that can be extended is a good choice.

The following properties were identified to be configured by XML-files. There is one such file per component.

- Component name.
- Names of exported services
- Names of requested services

In the case of a component representing a hardware unit such as a sensor, the following items are added to the configuration file.

- Name of hardware library.

- Hardware type.
- Hardware vendor.
- Hardware model.
- Hardware parameters depending on hardware type.
- Physical parameters depending on hardware type.

Some of these parameters contradict what was said regarding constant values. But these parameters serve a dual purpose. Apart from configuring the hardware and server, clients may query the server for these values. By putting these in the configuration file according to a format, they can more easily be put in a well-defined structure that can be transported to the client.

A complete example of an XML configuration file can be found in Appendix A.

The parsing of the XML-file is done by converting it into a DOM (Document Object Model) tree.

Runtime Configuration

One very important issue is dynamic reconfiguration of interconnections at runtime. While some systems may not need this, it is nevertheless useful for error-recovery. A component may then reconnect to another component in the case of crash.

In a typical behavior-based architecture (section 2.2), different set of behaviors send motion-commands to the motion hardware depending on the current task. Therefor such systems need to be reconfigurable in runtime.

Our solution to this problem was to use the standard Send pattern (section 7.3) which is used to send a string containing the names of other components to which the receiver should connect.

Fancier solutions would be to send new configuration consisting of XML. Another feature would be to be able to configure the connections by using a graphical user interface.

9.6 Summary

In this chapter

- an OrcaComponent was defined in order to provide a standard container for components.
- a standard state-machine was specified for managing the internal state of components.
- configuration is carried out with the use of XML-files.
- runtime configuration was discussed.

Chapter 10

Hardware Abstraction

Portability across hardware platforms and devices is a highly desirable design goal. This obviously eases development when adding new devices such as sensors. Though generally not considered, at least not in academia, software implementation is often more costly than the hardware on which it runs. Obviously there is then great benefit from being able to move existing software on to new hardware. Remember also that hardware tend to change quite often, so that software often outlives hardware (if it is well written).

10.1 Basics

A portable system should provide abstraction of hardware such as sensors and actuators. Although robot manufacturers have different hardware solutions, the fundamental basis often remains similar. At the lowest level, one or two motors control the drive and steering. Often a sonar ring and bumpers are present.

The manufacturer usually provides an API that lets the programmer use slightly higher level commands such as to move in absolute or velocity mode.The system should encapsulate these hardware specific commands into a generalized set of commands.

Abstractions should be made at different levels. For instance, on a synchro-drive system, a move command at a higher level will be transformed into separate low-level commands for the drive-motor and the steer-motor. This way, a programmer can choose the level suitable for his application programming.

Ideally the hardware characteristics should be kept in a single file in the software source. Then this file would be the only place where changes have to be made when moving the system to new hardware.

This abstraction at the same time makes it more difficult to exploit special purpose sensors and hardware and there is thus a balance between abstraction and efficiency.

10.2 A Hardware Abstraction API

This section describes our design regarding hardware abstraction. The guiding principles have been to make as simple and practical as possible. Where possible, other patterns from the framework should be reused. Following this principle, we managed to use the aforementioned communication patterns all the way down to the hardware library.

The Hardware Container

When setting out to partition the hardware of a robot into components, one immediately runs into problems. This is due to the fact that most off-the-shelf robots consists of a base containing locomotion as well sensor hardware. Often these are controlled by the same electronics, accessed as a single device, e.g. via a serial port. This means that the hardware modules cannot be separated into different binary components since not more than one process can access a port at one time.

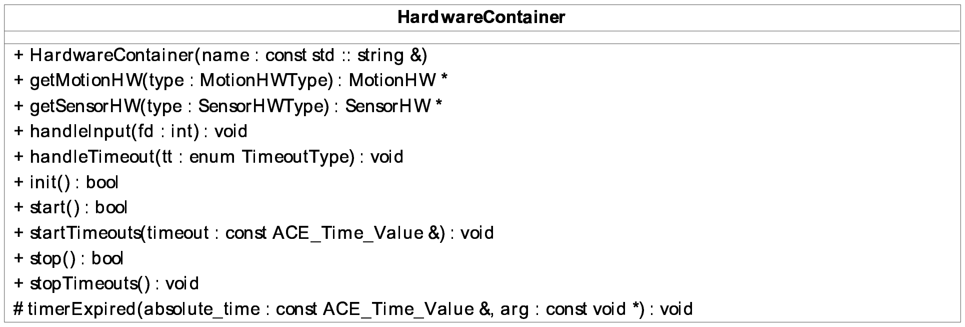
On e.g. the Nomad 200, the hardware can actually be accessed by different processes simultaneously but this is very inefficient because the they will then poll the hardware individually. And as the the polling command is the same for all hardware, and the fact that hardware access takes a very long time (in the order of tenths of seconds), much time is wasted.

This means that the hardware that is controlled by the same port should be bundled together into one component. This does violate the designers urge to separate according to abstraction, but is nevertheless necessary. Often the robot is equipped with additional sensors. These can readily be handled by separate components.

Some would argue to place all the hardware of a robot into one single component, a *Robot-Server*. But that leads to other problems. E.g. what constitutes a robot? Is a sensor placed on a robot totally different from a similar sensor placed on the wall, or on another robot? That schemes also means that the source code of the component has to be modified when placing a new sensor on the robot. This adds to complexity and modification is known to introduce bugs.

Our approach is to define a generic *hardware container*, Figure 10.1. This hardware container can contain a number of sensor and motion containers, see Figure 10.2. A hardware container handles only one hardware library. The hardware container handles initiating, starting and stopping the underlying hardware that it is responsible for.

This HardwareContainer fills the needs very well for representing a robot-base as those described above. It is sub-classed for every specific robot-base, sensor or other type of hardware. The HardwareContainer is then wrapped in a component, e.g. a laserrangefinder server or a robot-base server.



```
HardwareContainer

+ HardwareContainer(name : const std :: string &)
+ getMotionHW(type : MotionHWType) : MotionHW *
+ getSensorHW(type : SensorHWType) : SensorHW *
+ handleInput(fd : int) : void
+ handleTimeout(tt : enum TimeoutType) : void
+ init() : bool
+ start() : bool
+ startTimeouts(timeout : const ACE_Time_Value &) : void
+ stop() : bool
+ stopTimeouts() : void
# timerExpired(absolute_time : const ACE_Time_Value &, arg : const void *) : void
```

Figure 10.1: The HardwareContainer shown at some detail.

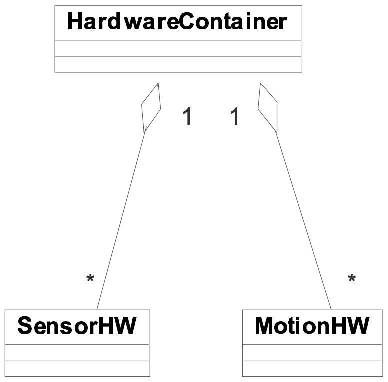


Figure 10.2: The HardwareContainer contains sensors and/or motion hardware.

Sensors

Sensors are represented by the SensorHW base-class. It is in turn sub-classed for every major type of sensor, such as laserrangefinders, sonars etc (Figure 10.4).

If the sensor is of standalone type, it is put in it's own HardwareContainer. If it belongs to a robot such as the Nomad200, it is put in that robot's HardwareContainer.

A SensorHW class typically provides a PushServer and/or a QueryServer (see section 7.3), as shown in Figure 10.5. These servers are instantiated at the component level, according to the configuration. This is because their presence are optional and the name can be set at runtime.

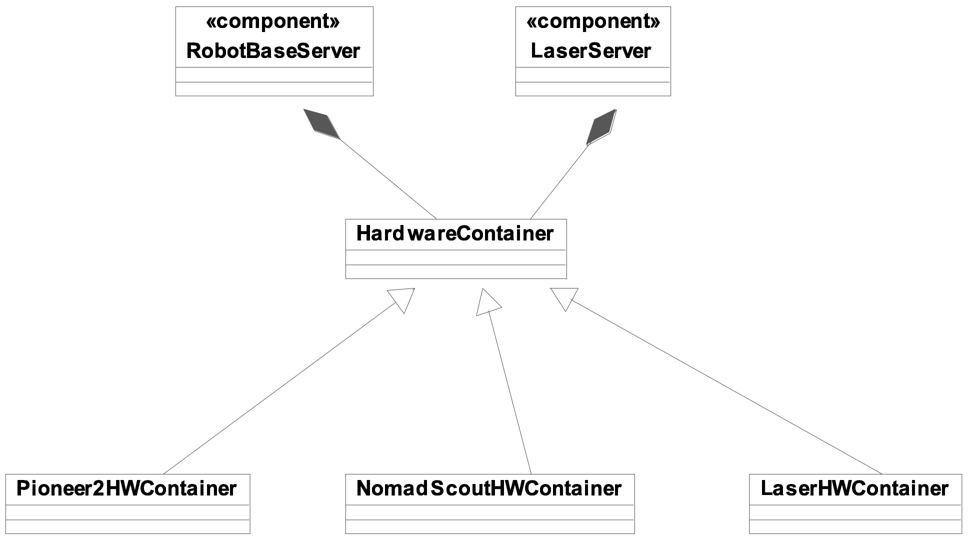


Figure 10.3: The HardwareContainer with neighboring classes.

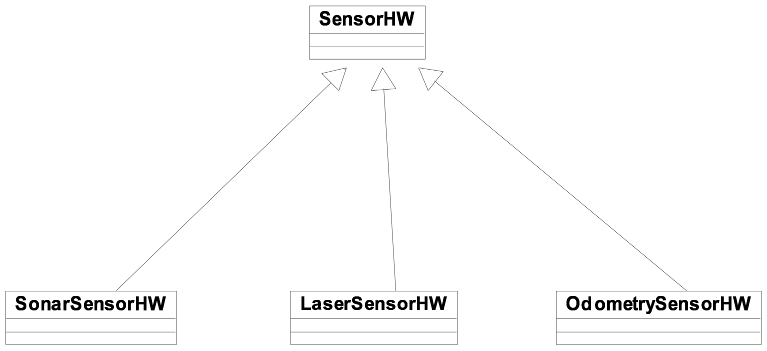


Figure 10.4: A SensorHW class is sub-classed for different types of sensors.

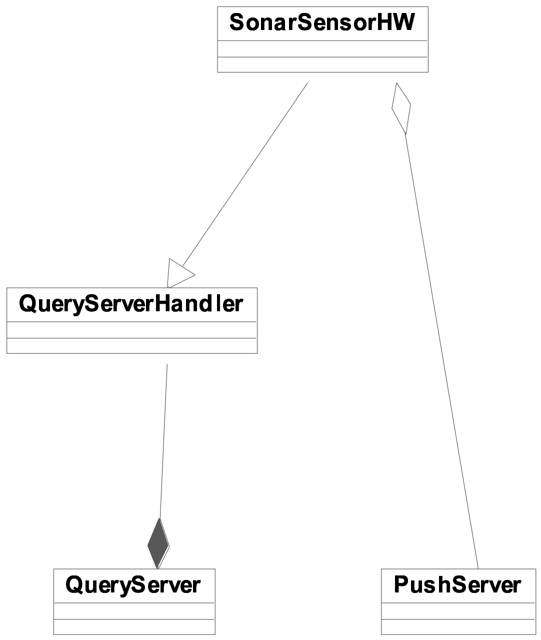


Figure 10.5: A sensor typically provides a PushServer and a QueryServer.

Motion Hardware

The MotionHW class is used for actuators that locomotes a robot. The class may need to be sub-classed for specific hardware (Figure 10.6). This is also instantiated at component level according to configuration.

Usually only a SendServer, accepting motion-commands, is associated with with this class (Figure 10.7). Note that a crisp motion-command is required, as no blending is carried out. A standalone component that fuses motion-commands should be used if that need exists.

10.3 Concluding Remarks

Some clients may need to know specifics about a hardware server such as maximum speed, range etc. As stated in section 9.5, this data can be made available in configuration files and then supplied by the server via another QueryServer. This has so far not been called for in current implementations, but can easily be added.

The implemented RobotBaseServer is, as said earlier, designed to be as generic as possible. It can be used without modification for a number of robots. Its main job is to parse the configuration file and instantiate servers which are tied to the hardware library that is linked in dynamically at runtime. The only thing the developer needs to do is to

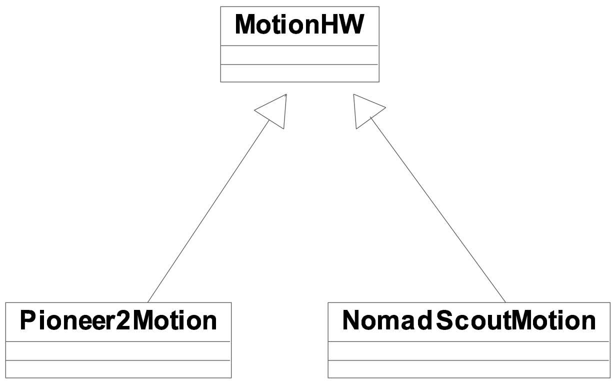


Figure 10.6: A MotionHW can be sub-classed for a specific platform.

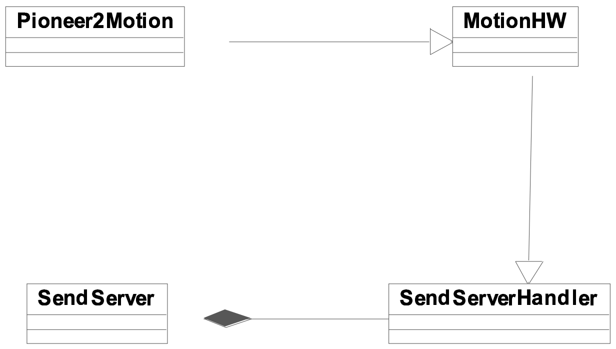


Figure 10.7: A MotionHW can be sub-classed for a specific platform.

supply a hardware specific configuration-file and a hardware library that is built according to the above presented HardwareContainer.

The RobotBaseServer class is however too much hard-coded in our view. E.g. it has to be modified in case new (major) types of sensors as well as new objects are needed. The initial idea was to make a perfectly generic component that could instantiate whatever classes was named in the configuration-file. This however failed. The main reason was the choice of programming language C++. Had a language that supports reflection(Green 2004), such as Java, been chosen, it would have had greater chance of success.

Recently, a bridge was developed by ACFR, that lets us interface directly to Player/Stage (see section 3.8) supported hardware. This gives us access to hundreds of different hardware devices.

10.4 Summary

In this chapter we introduced some new ideas regarding hardware abstraction,

- a HardwareContainer class was defined that can represent everything from a robot-base to a laser-scanner.
- a SensorHW class was used to represent any sensor device.
- a MotionHW class for representing actuators.

Chapter 11

Architecture and Implementation

This section describes an architecture and implementation that uses the proposed component-framework. According to IEEE 1471, architecture is *the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution*.

11.1 Components

Here we describe a select number of implemented components. Remember that a component runs in its own process and is executable. For each component there is a corresponding configuration file (section 9.5. This file contains the names that should be used for exported interfaces. It also includes the names of the interfaces (or servers) that the component should connect to. This makes these things totally configurable at runtime. This also means that the names of interfaces found in the following pages are just examples and could easily be changed.

The following presentation of components are divided into behavior, hardware, deliberate and other components. Trying to label components like this can be hard and even unfortunate. This is because it implicitly enforces an architectural view. Remember that components should be regarded as equal peers from the frameworks point of view. In that sense there are no differences between the categories. These categories are used here in reference to the commonly used hybrid architecture (section 2.3).

Behavior Components

The fundamental building blocks of a behavior-based system are of course the behaviors. As stated earlier (see section 2.3), one or more behaviors can execute concurrently. If more than one behavior that effects the same hardware executes, their output needs to be fused since the hardware needs one crisp motion-command.

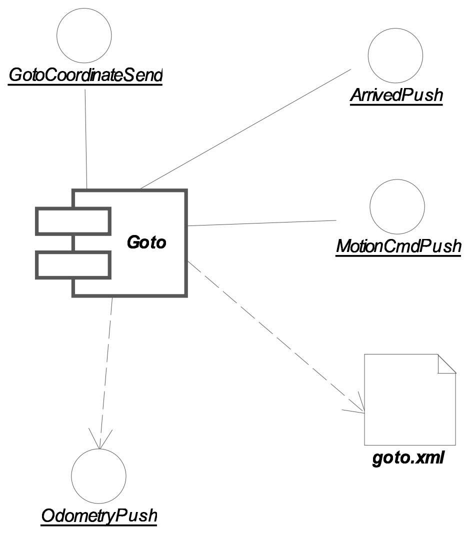


Figure 11.1: The Goto component.

Goto

A Goto behavior takes as parameter a goal-point in 2D- or 3Dspace and strives to drive the robot to that goal. In order to be able to this, it needs to have information on the current location of the robot on a regular basis. This information can e.g. be obtained from localization components or odometry servers.

An implementation of a Goto behavior can be seen in Figure 11.1. This component exports three interfaces. GotoCoordinateSend receives the goal-point to which the robot should move towards. Remember that the flow of data is from a SendProxy to a SendServer (section 7.3).

The second exported interface is named MotionCmdPush and is a PushServer that sends a motion-command for every motion-control loop lap. The last interface is called ArrivedPush. Interested parties can subscribe to this PushServer in order to learn when the robot has arrived at the goal.

This particular Goto component depends on another component that exports a odometry data via a PushServer.

Avoid

The task of an Avoid behavior is to have the robot avoiding driving into obstacles. An Avoid behavior can avoid stationary as well as moving obstacles depending on the implementation. If there are no obstacles in the vicinity, this behavior should do nothing. But in the presence of obstacles, its job is to have the robot keep a safe distance between the robot and the obstacles. In its simplest form, it will just stop the robot if an obstacle is

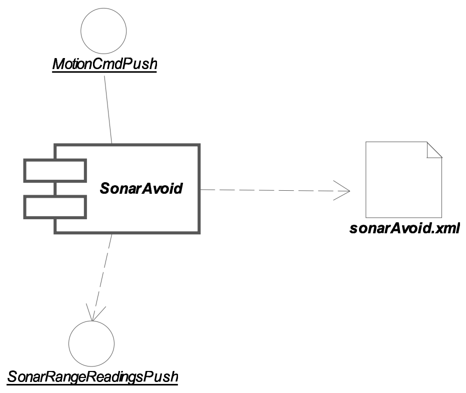


Figure 11.2: The Avoid component.

too close. More advanced algorithms can have the robot steer around obstacles in smooth curves. Avoid behaviors are almost always used in conjunction with other behaviors, such as Goto.

Avoid behaviors use data from sensors in order to detect obstacles. Several different kinds of sensors can be used. Sonars, laser, cameras and infra-red are the most common sensors.

The textbooks are virtually filled with algorithms concerning the fine art of obstacle-avoiding. Common methods use potential field histograms (PFH) or vector field histograms (VFH). See e.g. (Borenstein & Koren 1991). Though it seems hard to find an algorithm that gives smooth trajectories in every situation.

Figure 11.2 shows an Avoid behavior that utilizes sonar range-readings for detection. It exports an interface that provides a push-service. This service will push a motion-command for every lap of its calculation loop.
The ORCA distribution contains two Avoid behaviors, one uses laser-readings, the other uses sonar-readings. At ACFR, an Avoid behavior based on VFH+ is being developed.

Hardware Components

A Hardware component encapsulates a piece of hardware such as a sensor or a robot-base 10.2. They are a bit different than other components in a number of ways:

- They wrap a HardwareContainer.
- They seldom import (need) other other components in order to execute.
- They dynamically link to a hardware library.
- They seldom do anything else than instantiate servers which are then handed over to the hardware library.

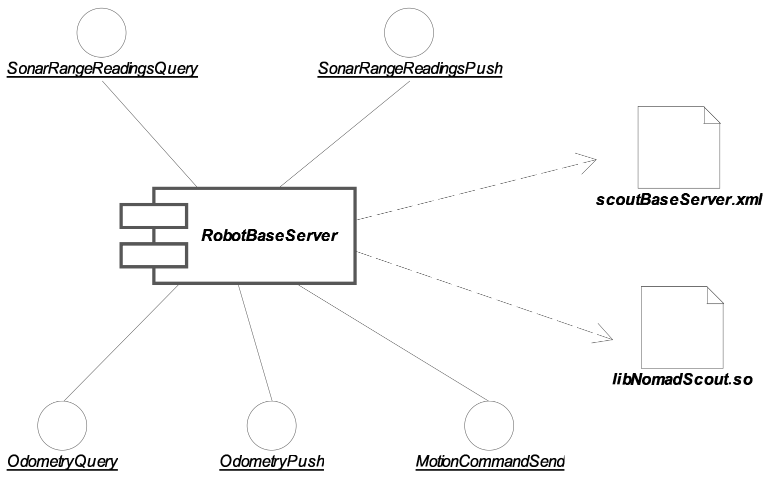


Figure 11.3: The RobotBaseServer component linked with the library libNomadScout.so.

RobotBaseServer

The RobotBaseServer is a fairly generic wrapper for several types of robot-bases (see section 10.3). The current implementation can handle sonars and odometry in any combination with Push and Query as stated in the configuration file. It also handles motion actuators using the Send pattern. So far hardware libraries and configuration files have been constructed for three different robots, SuperScout, Pioneer2 and Peoplebot.

As seen in Figure 11.3, when started with the configuration file for the Nomadics SuperScout, it exports five interfaces. Since the scout has sonars and odometry sensors, PushServers and QueryServers for sonars and odometry are started. And since the scout can move, a SendServer accepting motion-commands is there as well.

LaserServer

The LaserServer component wraps a laserrangefinder sensor. It can be used with different brands as long as a hardware library exists that supports the HardwareContainer class. It exports Push and Query for the LaserRangeReadings object.

Deliberate Components

To the category deliberate components belong planning, reasoning, human-robot interfacing etc (see section 2.3). For example in ISR (section 3.6), one module simply called Planner takes care of communicating with humans in different ways, mission-planning, path-planning and a range of other tasks. This was clearly a mistake since the source code has become sp cluttered and complicated that no one can debug it. The solution is of course to split the tasks into several components.

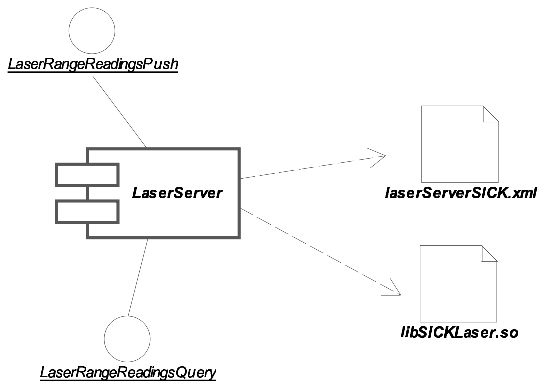


Figure 11.4: The LaserServer component.

Command

For this prototype implementation we so far only have very simple somewhat deliberate component, see Figure 11.5. It asks for typed in coordinates and then interacts with other components to in order to take the robots to those coordinates and then stop. Then it asks for new coordinates from the user and the task is repeated.

To carry out its mission, it first sends the coordinates to a component exporting the GotoCoordinateSend interface, typically a Goto component (section 11.1). It also starts subscribing to that same component's ArrivedPush server. Then it sends a list of clients to a component exporting the ClientListSend interface, which in this case will be a MotionCmdFuser. It will then wait for a message arriving via the ArrivedPush interface. Then the procedure is repeated.

Other Components

There can be components that are hard to fit into the above categories. E.g. helper components carrying out transformations or general computations. Here only one such component is studied.

MotionCommandFuser

In a behavior-based system, several behaviors can run concurrently. But a hardware actuator can only accept one crisp command. Therefor the motion-commands from the behaviors must be fused into one. This is the job if the MotionCmdFuser (see Figure 11.6). This component will be tied to a certain hardware component according to the configuration file. But since the running behaviors will vary during execution, these are not known initially. Instead the component exports an interface that accepts a list containing the names of the the behaviors. Upon receiving such a list, it will contact these components and start a sub-

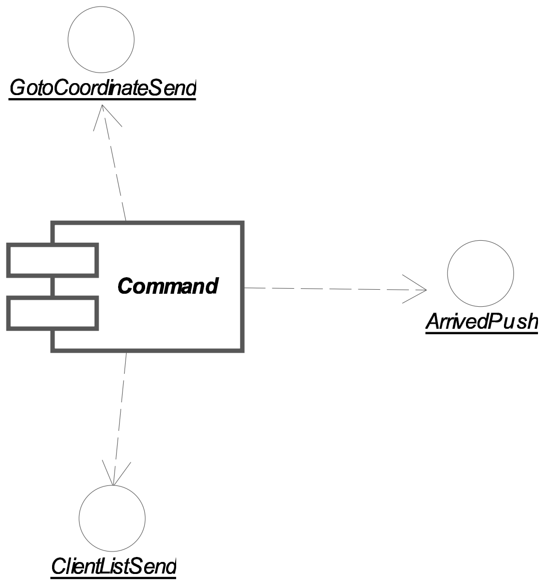


Figure 11.5: The Command component.

scription. The motion-commands received are fused and sent to the hardware component. It uses the synchronization feature (see section 7.4).

Missing Components

This chapter only presents a very simple robotics system. To able to do anything useful, a number of components need to be added.

Localizer

For all but the simplest mission, a robot needs to know it's location. This location should have an margin of error within a couple of centimeters. The odometry sensor present on many robots tend to drift quite rapidly. Therefor more elaborate means need to be deployed. Many methods use other types of sensors such as sonar, laser or cameras. Some fuse information from several sensors to get a more accurate results. Different filters such as the Kalman filters are also used.

One technique is to record so called landmarks in the environment. Landmarks are segments that are easily distinguished and the sensor picks up on several runs. Corners are e.g. good landmarks for a sonar-sensor. The landmarks are stored in a map and later compared to the sensor-readings obtained when trying to localize.The map that the robot is using for localization, should have common references with the human operators. Otherwise commanding a robot to a certain place is very hard.

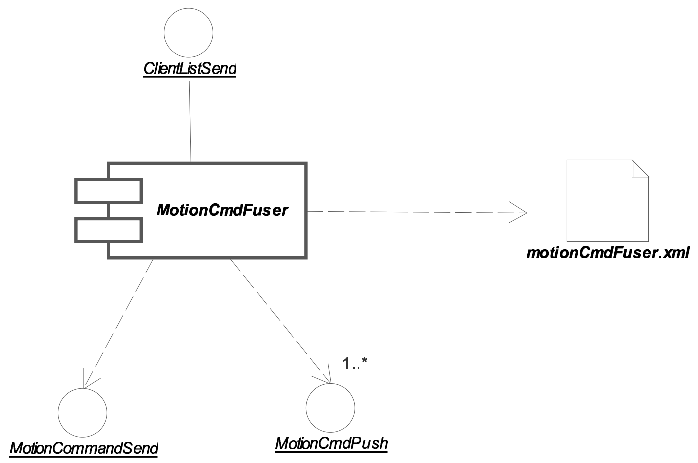


Figure 11.6: The MotionCmdFuser component.

Recently there has been a large amounts of research on simultaneous localization and mapping (SLAM). This is especially useful in missions operating in unknown, unmapped environments.

Localization can actually be split between several components. E.g. one position-server gathering and fusing data from one or more localizer and position-trackers. Localizing algorithms are typically very computationally expensive.
At ACFR, work is going on to add a SLAM localizer to the system.

Planners

A mission-planner is needed if the robot should be able to perform reasonably complex tasks. It handles the global state of the system. The task at hand is broken down to a sequence of global states. Each global state is associated with a certain set of running behaviors. E.g. the state Goto (aka GoPoint aka GotoGoal) is usually mapped to the simultaneous running of the Goto and Avoid behaviors. A mission-planner should also be able to detect and compensate for errors.

Path-planning is also necessary if the robot should be able to navigate e.g. between rooms. There are many ways to do this. One popular algorithm is called A* (see e.g. (Hwang & Ahuja 1992)).
A path planner which uses occupancy grid representation is being developed at ACFR.

Human-Robot Interface

Although not strictly necessary, a multi-modal human-robot interface is a vital component in robotic system. Commonly speech, gestures and web-based interactions are used.

Having the robot provide feedback is almost more important. Speech synthesis is an obvious choice here.

11.2 Deployment

Running the System

The system is presently started in a very crude way. The components are started one by one in a terminal window. The configuration file is given on the command-line. First of all the CORBA NamingService must be started. This can however be left running between sessions. The TAO NamingService uses multicast which is non-standard but provides a very convenient way for other components to locate it. One can expect trouble if several people run similar systems with NamingServices running on the same port.

Collaboration diagrams of the phases in a typical run of the system are shown on the following pages. Figure 11.7 shows the initialization phase which starts from the Command component. In Figure 11.8 all the reactive components are active and data is being pushed repeatedly. This phase continues until the goal is reached. In Figure 11.9 the goal is reached which results in the robot being stopped. All connections are shut down.

Simulation

The simulator from Nomadics called NServer can be used to simulate the SuperScout robot with the system. This also provides a graphical view of the robot as well as sonar and infrared readings. Obstacles can also be added to the map. A screenshot can be seen in Figure 11.10.

As briefly mentioned in section 10.3, we can now use hardware libraries that is developed for the project *Player/Stage*. This support also includes simulation for a large amount of hardware.

Graphical User Interface

Graphical user interfaces (GUI) can look very differently. They are sometimes developed without any specific user category in mind. However, they should really be different depending on the intended use. A programmer who mainly wants to do debugging has other needs than an end-user who wants to operate on and monitor a system. A solution would be to have a highly configurable interface where most of the views can be hidden.

Another aspect is that adding a GUI may decrement the performance or alter the behavior of a system, since added load and communication might have an impact.

Currently a GUI exists that can display entities of the world in different views. It connects to the ordinary PushServers of hardware components. Figure 11.11 shows a local

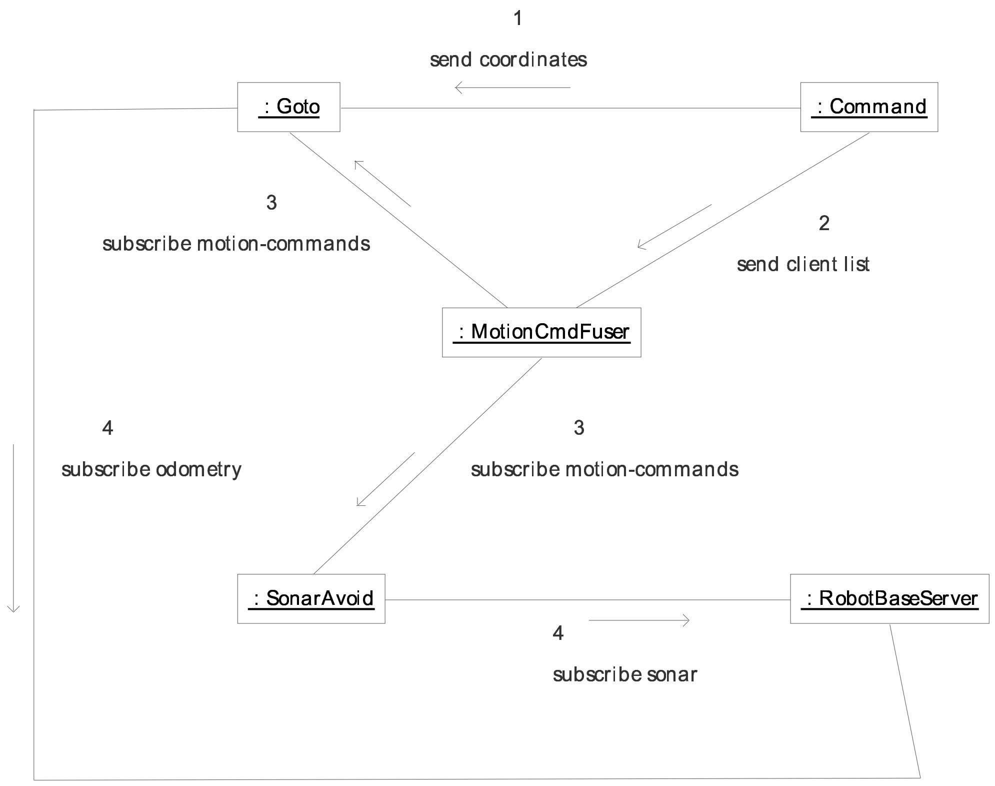


Figure 11.7: Collaboration diagram showing the initialization phase in a typical run.

view of a platform displaying its sonar scan (yellow), laser scan (blue), speed (vertical red arrow) and turn-rate (red arc arrow). This GUI is based on the graphics toolkit QT. This GUI has been made by ACFR.

Plans are to construct a GUI in which which re-wiring of component interconnections can be made dynamically.

Performance

A scientific evaluation of the performance has not yet been made. An undergraduate project has been started to measure this, but so far no numbers are available.

There are however some conclusions that can be drawn. Of the several projects using the framework, none have so far expressed concerns about latencies. This is also the case when many components are active at the same time. This means that the communication overhead does not introduce latencies comparable to the time interval with which the hardware is probed.

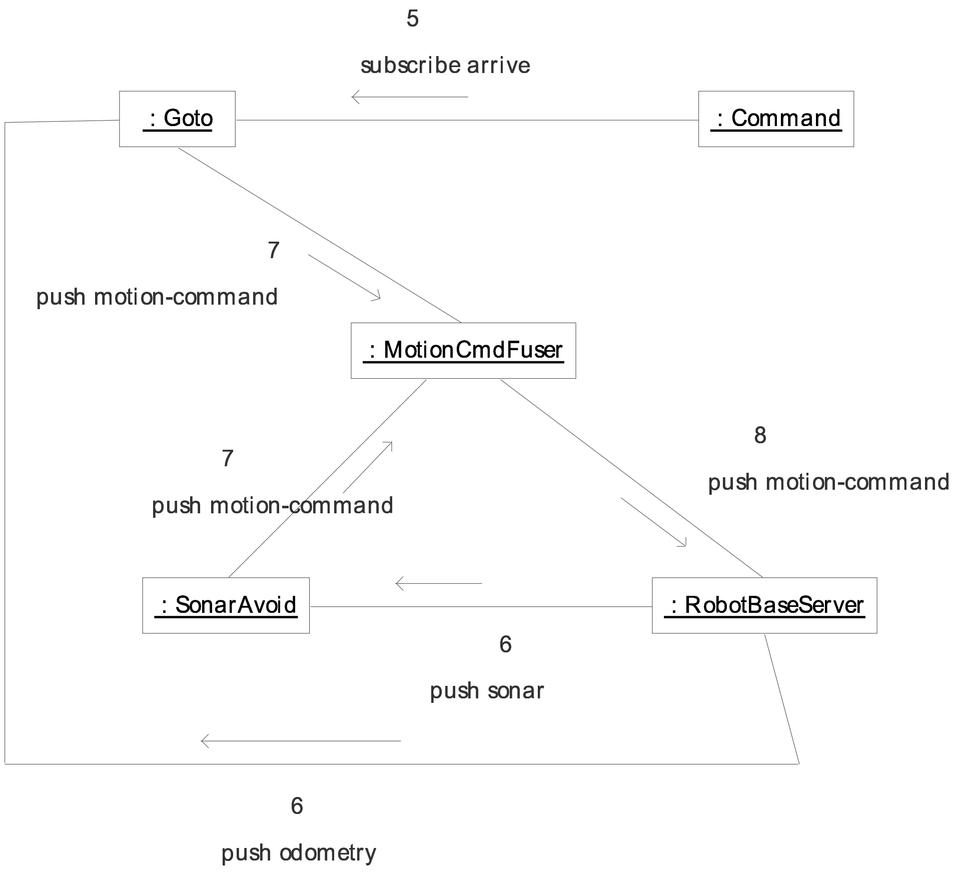


Figure 11.8: Collaboration diagram showing the running phase in a typical run.

In (Wenfeng et al. 2004), 40 milliseconds is measured between sensor responses on average. The ORBit people have done some comparisons on different CORBA toolkits, which clocked 10,000 simple operation invocations and TAO finished after 8.81 seconds (GNOME 1999).

One empiric result is that when the Query pattern is used extensively, the load of the system can become high. This is rectified by switching to the Push pattern.

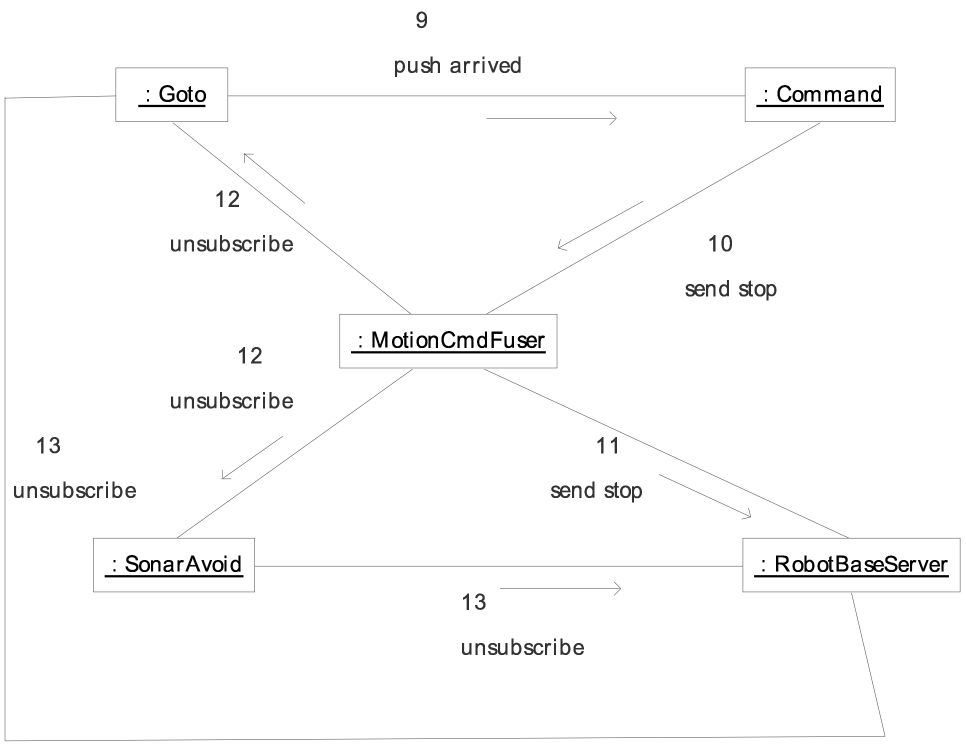


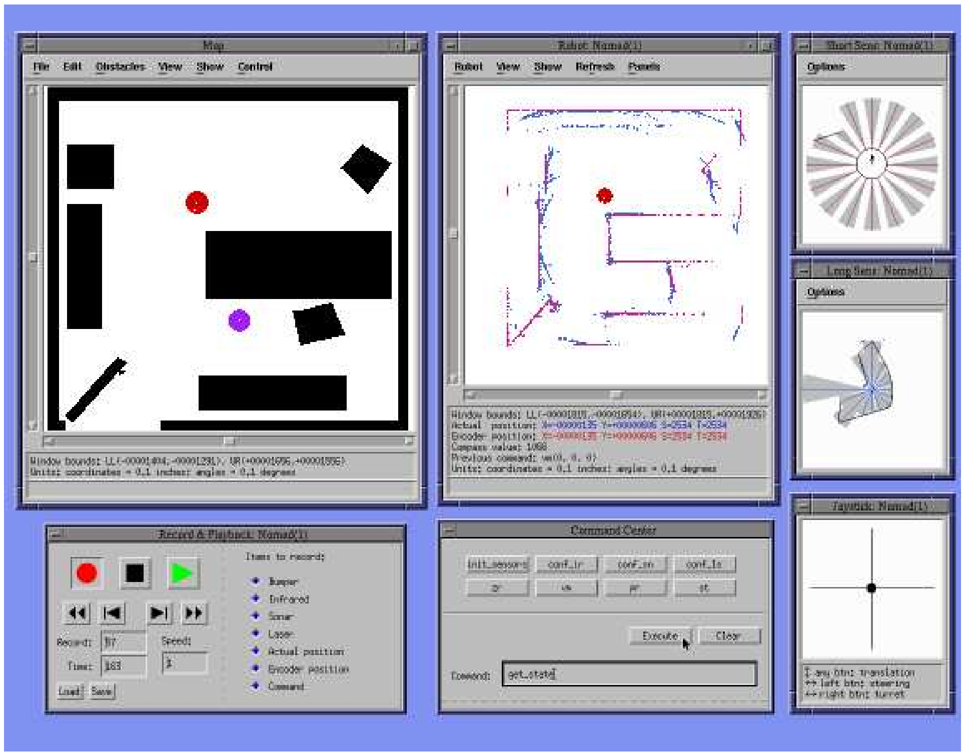
Figure 11.9: Collaboration diagram showing the closing phase in a typical run.

11.3 Development

Ease of development is of paramount importance if a a piece of software should widely used. The following features are present to aid the developer.

- One standardized component model.
- A small number of, but versatile, communication patterns.
- Helper classes for state-machine, XML-parsing, synchronizing etc.
- Most (but not all) CORBA calls are hidden.

Development is currently being done to will hide more CORBA specific code and to let the component developer only have to deal with one class file.



```
Window bounds: LL(-00001404,-00001281), UR(+00001696,+00001596)
Units: coordinates = 0.1 inches; angles = 0.1 degrees
```

```
Window bounds: LL(-00001815,-00001894), UR(+00001815,+00001926)
Actual  position: X=-0000135 Y=+0000606 S=2534 T=2534
Encoder position: X=-0000135 Y=+0000606 S=2534 T=2534
Compass value: 1088
Previous command: vm(0, 0, 0)
Units: coordinates = 0.1 inches; angles = 0.1 degrees
```

```
↕ any btn: translation
↔ left btn: steering
↔ right btn: turret
```

Figure 11.10: A screenshot of the Nomadics Scout simulator.

Toolkits and Libraries

Here is a summary of the toolkits and libraries that the framework currently depends upon.

TAO is the CORBA implementation we use.

Log4Cpp is a toolkit to assist in logging.

XMLwrapp a C++ library for working with XML data.

libxml2 is an XML C parser on which XMLwrap is built.

Doxygen Doxygen for generating documentation.

Since ACFR added support for Player/Stage, it is required for some hardware support and for use of the Stage simulator. QT is also needed for the ACFR graphical user interface.

XMLwrapp is a very nice toolkit that unfortunately has not gained a wide user-base. ORCA is currently in the process of switching to the more popular toolkit Xerces which is under the Apache umbrella. Log4Cpp will probably be exchanged for log4cplus since that toolkit is better maintained.

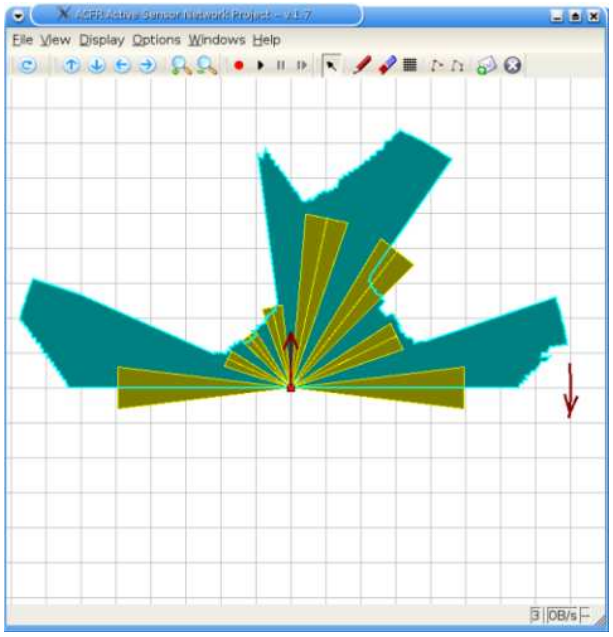

ACFR Active Sensor Network Project – v1.7
File View Display Options Windows Help
3 0B/s

Figure 11.11: A GUI showing sonar scan (yellow), laser scan (blue), speed (vertical red arrow) and turn-rate (red arc arrow)

Documentation

The documentation for ORCA is all available at the website. This includes instructions dealing with getting and installing the ORCA software and other required toolkits. There is also a guide on how to quickly get the software running. The available developer information consists of webpages, examples, and presentations. The code documentation generated with Doxygen can also be browsed, as well as the CVS repository. There is not yet something that could be called a manual or handbook.

11.4 Problems and Unhandled Issues

Threading Problems

When building distributed multi-threaded systems, some problems tend to be hard to avoid. One of these is threading. It is very hard to predict exactly what parts of the code that may end up running at concurrently in different threads. The problem is usually solved by introducing locks all over the code. This can still lead to all kinds of problems, but also makes it more hard to understand the code. Efficiency may also be reduced.

Error Handling

The handling of errors is severely lacking the present system. Here are some ideas taken from the TODO list:

- Implement something like an alarm-central where components can send their errors and others can subscribe to them.
- Introduce MissionNumber/ID in all communications so that errors can be mapped to specific missions.
- A server must be able to "survive" a client disconnecting preferably no matter how it is disconnected. This is not the case presently.

Security

Security issues are very seldomly addressed in research software, and this software is no exception. Commercial systems could be expected to provide security at different levels. Requirements would include authentication between components and encryption of communication channels. More advanced features include access control at different levels.

CORBA provides specifications for a Security Service. It is a security reference model that provides the overall framework for CORBA security. TAO contains an implementation of this service.

It is doubtful whether security needs to be considered in the short run. The data that is available in a robotic environment is hardly of any interest to any exterior party. The only realistic scenario is if a malicious person connects to robot components with the indent of taking over control or to cause damage to the system. Since security measures generally increase complexity and decrease performance, we have decided to postpone implementation until the need arises.

Automatic Startup

The way the system is started currently is not satisfactory. By using the CORBA service called ImR (see section 4.4), a component will be started as soon as another component requests to use it.

11.5 Summary

The ORCA framework has gained some maturity and is being used in several projects, see e.g. (Wenfeng et al. 2004).

Available Components

ORCA comes with a number of ready-made components, and more are developed all the time. The following were available before ACFR began developing for ORCA.

Goto goto goal behavior.

SonarAvoid obstacle avoidance behavior using sonars.

LaserAvoid obstacle avoidance behavior using laser.

RobotBaseServer component representing a robot base.

LaserServer server component for a laser-range finder.

Command a component for interacting with the system.

MotionCommandFuser a component that blends motioncommands for sending to an actuator.

LogServer a component to which log messages can be sent and processed.

Additionally, here follows a partial list of components currently being developed at ACFR. They are all in various stages of development.

GaussMapViewer a simple viewer that outputs a Gaussian point map.

GUI a Graphical User Interface.

LaserViewer a simple example that displays laser ranges.

LocalizerViewer a simple viewer that displays 2-dimensional pose data.

OgPathPlanner a path planner which uses occupancy grid representation.

RandomWalk drives robot randomly while avoiding obstacles.

SlamLocalizer simultaneous localization and map building.

TeleopControl a remote control interface.

TruthLocalizer localizer that provides pose using Stage.

VFH+ obstacle avoidance.

Higher level mission planning is still not present.

Hardware Support

Here is a list of the currently supported hardware:

- Nomadics Scout
- XR4000
- SICK laser scanner
- Pioneer 2
- PeopleBot

While the above are supported natively, the Player/Stage bridge developed at ACFR means that a large number of other hardware can be used as well.

Chapter 12

Summary and Future Research

This chapter aims to summarize the thesis, and to also presents topics for further research.

12.1 Summary

This thesis dealt with software systems and architectures for mobile robots. In the introduction, the biggest problems of robotics were defined as the lack of re-use and sharing resulting in a barrier into robotics research. The first part laid a foundation by studying various aspects that are necessary to take into account when developing such a system. First we briefly reviewed the evolution of robot architectures. Then we looked at a number of systems, both contemporary and from the past. We evaluated these in order to find what properties have showed to be successful, and what should be improved. Some

- modularity
- openness and extendability
- sound hardware model
- simple but powerful communication
- ease of use

We also concluded that enforcing a certain architecture restricts the versatility and hinders the system of becoming widely adopted. We also found that a well designed hardware abstraction invites developers to produce drivers. Furthermore, the traditional division of clients and servers should be replaced by a framework of common peers. The use of patterns in the interaction between components were found to be highly effective.

Then we studied some software engineering issues such as operating systems and programming techniques. Different communication technologies were investigated, with an emphasis on middleware. Component-based software engineering was found to be a valuable tool that could help resolving the problems defined in the introduction. The development process was also looked into. We defined the different programming roles and and touched on system modeling. The open source model was then discussed, followed by debugging, documentation and versioning systems.

Part two of the thesis presented a proposed software framework for mobile robotics. First a number of prerequisites such as application domain and hardware platform. We then discussed our choice of communication technology along with a proposed set of communication patterns. The following chapter discussed and presented the types and objects that are communicated in the system. Next, the concept of component model was explained and the model used in our system was presented. The importance of hardware abstraction was discussed in the following chapter along with the scheme used in the proposed framework.

The last chapter an implemented system using the proposed framework. The different components in the implementation were presented along with their deployment. Finally a number of unhandled issues were discussed.

12.2 Future Research

Robotics has as a research field matured significantly. There are however topics that need to be addressed. One such topic is that of standardization

Standardization

The software engineering community have been addressing the issues of standardization and reuse for several years. Many people mean that it is high time for the robotics community to adopt the knowledge, methods, and tools created. To re-iterate, a standard would simplify the following:

- exchange of software parts between labs, allowing specialists to focus on their particular field.
- comparison of different solutions.
- startup in robot research.

Standardization in a field is a sign of maturity but there are also negative aspects. Standardization means that at a particular point in time, a line has been drawn. This means that further evolution is if not stopped, but hindered in ways. But standardization can be done at different levels. It can be at a high level describing e.g. common terminology and data structures. The other extreme, where components from different systems actually can inter-operate implies of course a much more detailed and therefor more restrictive standard.

One obstacle towards standardization is the fact that there are a number of robotic fields with different demands, terminologies and traditions. There are e.g. different communities for mobile robots, manipulators and humanoids. The task of getting all these researchers to agree on common standards is not an easy task.

Recently, a number of exciting robot software projects have emerged. Apart from the one described in this thesis, there is Marie, CARMEN and Miro. The Player/Stage project is also preparing to enter a second generation. The software of these projects are not compatible but there are many similarities and their designers share much of the same ideas. During the summer and autumn of 2004, discussions have been carried out on mailing-lists on to which extent common standards can be adopted. This collaboration is being coordinated by Herman Bruyninckx, who manages OROCOS.

More on standardization in robotics can be found in (Vaughan et al. 2003), (Roy et al. 2003), (Nesnas et al. 2003), (Huang et al. 2003) and (Hattig et al. 2003).

Hardware Range

A related issue to standardization is that a framework only can be used on a limited range of hardware. In this context we refer to the computing resources such as CPU and memory. Those dealing with e.g. service robots usually have a standard PC that offers the possibility to use affective but large footprint software such CORBA. But this kind of technology is not usable on small embedded systems. And since the former do not want to settle on a least common denominator, but rather fully exploit available resources, constructing standard mechanisms that can be deployed on both cased is very hard.

Separating the libraries from the API is one way to do it. Or in other words, separating the policy from the mechanism. This is most likely the best way to go, but there are disadvantages. It drastically increases complexity and means that two or more parts have to be maintained and synchronized. This scheme usually also implies the use of `#ifdefs` which is said to be the road to disaster. Wrappers in general results in code that is hard to maintain. The question is if our current programming tools are enough to solve this problem?

At ACFR, the CORBA specific code has actually been separated in order to create a truly transport neutral framework. This was implemented by using the wrapping method.

Mobile Manipulation

It is more and more common to place a manipulator arm on a movable robot base. This introduces a number of problems. One that is relevant to this thesis is the fact that the manipulator needs to be controlled by a computer running a real-time operating system, while the base-software in most cased will be controlled from a non real-time OS. These systems need to communicate, especially so if the manipulator is to be operated in synchronization with the base. This cannot be done readily without compromising the real-time properties. Solutions for this have been proposed e.g. in (Petersson 2002), but further research is necessary. The long-term solution is likely though that all robotics systems gravitate towards using real-time operating systems.

Appendix

Appendix A

Component Configuration

An example XML configuration file.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE component SYSTEM "component04.dtd">
<component>
    <componentName>RobotBaseServerComponent</componentName>
    <rootContextName>default</rootContextName>
    <context id="sensors" name="Sensors">
        <context id="sonar" name="SonarServer">
            <server id="sonarquery">
                <NameService>
                    <use>true</use>
                    <nsName>SonarRangeReadingsQuery</nsName>
                </NameService>
                <TraderService>
                    <use>false</use>
                    <serviceType></serviceType>
                </TraderService>
            </server>
            <server id="sonarpush">
                <NameService>
                    <use>true</use>
                    <nsName>SonarRangeReadingsPush</nsName>
                </NameService>
                <TraderService>
                    <use>false</use>
                    <serviceType></serviceType>
                </TraderService>
            </server>
        </context>
        <context id="odometry" name="OdometryServer">
            <server id="odometryquery">
                <NameService>
                    <use>true</use>
                    <nsName>OdometryQuery</nsName>
                </NameService>
                <TraderService>
                    <use>false</use>
                    <serviceType></serviceType>
                </TraderService>
            </server>
            <server id="odometrypush">
                <NameService>
                    <use>true</use>
                    <nsName>OdometryPush</nsName>
                </NameService>
                <TraderService>
                    <use>false</use>
                    <serviceType></serviceType>
                </TraderService>
            </server>
        </context>
    </context>
    <context id="actuators" name="Actuators">
        <context id="motion" name="MotionServer">
            <server id="motionsend">
                <NameService>
                    <use>true</use>
                    <nsName>MotionCommandSend</nsName>
                </NameService>
                <TraderService>
                    <use>false</use>
                    <serviceType></serviceType>
                </TraderService>
            </server>
        </context>
    </context>
    <hardware>
        <library>libOrocosNomadScout.so</library>
        <robot>
            <name>Dewey</name>
            <vendor>Nomadics</vendor>
            <model>SuperScout</model>
            <radius>190</radius>
            <bumper_radius>205</bumper_radius>
        </robot>
        <sensors>
            <sensor id="sonarring1">
                <sensor_type>
                    <sonar-ring number_of_sonars="16" dphi="225"
                    angle_sonar0_x_axis="0"  half_lobe_angle="125"
                    scalefactor="0.0435" offset="-12.0" />
                </sensor_type>
                <vendor>Polaroid</vendor>
                <update_interval>0.2</update_interval>
            </sensor>
        </sensors>
        <actuators>
            <actuator id="actuator1">
                <actuator_type>
                    <drive>
                        <drive_type>
                            <differential/>
                        </drive_type>
                        <maxspeed>1000</maxspeed>
                        <max_steerspeed>1</max_steerspeed>
                        <steergain>0.5</steergain>
                        <dspeed>50</dspeed>
                        <big_turnangle>1</big_turnangle>
                        <big_turnspeed>12.5</big_turnspeed>
                    </drive>
                </actuator_type>
            </actuator>
        </actuators>
    </hardware>
</component>
```

Bibliography

Albus, J., McCain, H. & Lumi, R. (1987), Nbs standard reference model for telerobot control system architecture (nasrem), Technical Report 1235, National Bureau of Standards, Gaithersburg, MD.

Andersson, M., Orebäck, A., Lindström, M. & Christensen, H. (1999), *I*ntelligent Sensor Based Robotics, Springer Verlag, Heidelberg, chapter ISR: An Intelligent Service Robot.

Arkin, R. C. (1986), Path planning for a vision-based autonomous robot, *i*n 'Proceedings of the SPIE Conference on Mobile Robots'.

Arkin, R. C. (1987), Towards cosmopolitan robots: Intelligent navigation in extended man-made environments, Technical Report COINS 87-80, Ph.D. Dissertation, Dep. of Computer and Information Science.

Arkin, R. C. (1989), 'Motor schema-based mobile robot navigation', *I*nternational Journal of Robotics Research **8**(4), 92–112.

Arkin, R. C. (1990), Integrating behavioral, perceptual, and world knowledge in reactive navigation, *i*n 'Robotics and Autonomous Systems, Vol. 6, pp. 105-22'.

Arkin, R. C. (1998), *B*ehavior-Based Robotics, The MIT Press, Cambridge, Massachusetts, London, England.

Asimov, I. (1950), *I, Robot*, Spectra.

Balch, T. (2000), 'Teambots'. www.teambots.org.

Barbera, A., Albus, J., Fitzgerald, M. & Haynes, L. (1984), Rcs: The nbs real-time control system, *i*n 'Robots 8 Conference and Exposition', Detroit, MI.

Booch, G., Rumnaugh, J. & Jacobsen, I. (1999), *T*he Unified Modeling Language User Guide, Object Technology Series, Addison-Wesley.

Borenstein, J. & Koren, Y. (1991), 'The vector field histogram - fast obstacle avoidance for mobile robots', *I*EEE Transactions on Robotics and Automation **7**(3), 278–288.

Brooks, R. (1986), A robust layered control system for a mobile robot, *i*n 'Proceedings of the IEEE International Conference on Robotics and Automation', Vol. RA-2, pp. 14–23.

Brooks, R. A., Breazeal, C., Marjanovic, M., Scassellati, B. & Williamson, M. (1999), The cog project: Building a humanoid robot, *i*n C. Nehaniv, ed., 'find this !!!!', Vol. Computation for Metaphors, Analogy, and Agents of *L*ecture Notes in Artificial Intelligence, Springer, New York, pp. 52–87.

Bruyninckx, H. (2003), Kinematics and dynamics: Internal interfaces. draft.

Emmerich, W. (2000), *E*ngineering Distributed Objects, Wiley.

Engelberger, J. F. (1989), *R*obotics in Service, MIT Press.

Firby, R. J. (1989), Adaptive Execution in Complex Dynamic Worlds, PhD thesis, YALE.

Free Software Foundation, Inc. (2004), 'Licenses'. http://www.gnu.org/licenses/licenses.html.

Gerkey, B. P., Vaughan, R. T., Støy, K., Howard, A., Sukhatme, G. S. & Mataric, M. J. (2001), Most valuable player: A robot device server for distributed control, *i*n 'Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)', Wailea, Hawaii, pp. 1226–1231.

Gerkey, B., Vaughan, R. T. & Howard, A. (2003), The player/stage project: Tools for multi-robot and distributed sensor systems, *i*n 'Proceedings of the 11th International Conference on Advanced Robotics (ICAR'03)', Coimbra, Portugal, pp. 317–323.

GNOME (1999), 'Corba applications in gnome'. http://developer.gnome.org/doc/whitepapers/ORBit/about-orbit.html.

Green, D. (2004), 'The reflection api', http://java.sun.com/docs/books/tutorial/reflect/.

Hattig, M., Horswill, I. & Butler, J. (2003), Roadmap for mobile robot specifications, *i*n 'International Conference on Intelligent Robot Systems (IROS2003)', IEEE/RSJ, Las Vegas, Nevada.

Henning, M. (n.d.), 'Binding, migration, and scalability in corba'.

Henning, M. & Vinoski, S. (1999), *A*dvanced CORBA Programming with C++, Addison Wesley.

Huang, H.-M., Albus, J., Kotora, J. & Liu, R. (2003), Robotic architecture standards framework in the defense domain with illustrations using the nist 4d/rcs reference architecture, *i*n 'International Conference on Intelligent Robot Systems (IROS2003)', IEEE/RSJ, Las Vegas, Nevada.

Hwang, Y. K. & Ahuja, N. (1992), 'Gross motion planning a survey', *A*CM Computing Surveys **2**4(3), 219–291.

ISO (1986), 'Standard Generalized Markup Language (SGML)', Published ISO Standard. ISO 8879:1986.

ISO/IEC (1999), 'Interface Definition Language', Published ISO Standard. ISO/IEC 14750:1999.

Jacobson, I., Booch, G. & Rumbaugh, J. (1999), *T*he Unified Software Development Process, Addison Wesley.

Konolige, K. (1997), Colbert: A language for reactive control in saphira, *i*n 'German Conference on Artificial Intellgence', Freiburg.

Konolige, K. & Myers, K. (1996), 'The saphira architecture for autonomous mobile robots', SRI International.

Kortenkamp, D., Bonasso, R. P. & Murphy, R., eds (1998), *A*rtificial Intelligence and Mobile Robots - Case Studies of Successful Robot Systems, AAAI Press / The MIT Press.

Lindström, M., Orebäck, A. & Christensen, H. (2000), Berra: A research architecture for service robots, *i*n 'International Conference on Robotics and Automation', IEEE.

Lucas, G. (1999), ‘Star wars: Episode 1 - the phantom menace’, Motion Picture.

Maurer, M., Behringer, R., Dickmanns, D., Hildebrandt, T., Thomanek, F., Schiehlen, J. & Dickmanns, E. D. (1995), VaMoRs-P: an advanced platform for visual autonomous road vehicle guidance, *i*n ‘Proc. SPIE Vol. 2352, p. 239-248, Mobile Robots IX, William J. Wolfe; Wendell H. Chun; Eds.’, pp. 239–248.

Microsoft (2004), ‘Component object model’. http://www.microsoft.com/com.

Nesnas, I., Wright, A., Bajracharya, M., Simmons, R. & Estlin, T. (2003), Claraty and challenges of developing interoperable robotic software, *i*n ‘International Conference on Intelligent Robot Systems (IROS2003)’, IEEE/RSJ, Las Vegas, Nevada.

OMG (2003), ‘XML/Valuetype Language Mapping, v1.1’. www.omg.org.

OMG (2004), ‘Object Management Group’. www.omg.org.

ORCA (2004). orca-robotics.sourceforge.net.

Orebäck, A. & Christensen, H. I. (2003), ‘Evaluation of architectures for mobile robotics’, *A*utonomous Robots **1**4, 33–49.

OROCOS (2002). www.orocos.org.

OSI (2004), ‘Open source initiative’. http://www.opensource.org/.

O’Sullivan, J., Haigh, K. Z. & Armstrong, G. D. (1997), Xavier manual. Internal Manual.

Petersson, L. (2002), A Framework for Integration of Processes in Autonomous Systems, PhD thesis, Royal Institute of Technology.

Raymond, E. S. (2000), ‘The cathedral and the bazaar’. http://www.catb.org/˜esr/writings/cathedral-bazaar.

RETF (2003), ‘Robotics Engineering Task Force’. http://www.robo-etf.org.

Roy, N., Montemerlo, M. & Thrun, S. (2003), Perspectives on standardization in mobile robot programming, *i*n ‘International Conference on Intelligent Robot Systems (IROS2003)’, IEEE/RSJ, Las Vegas, Nevada.

Saffiotti, A., Ruspini, E. & Konolige, K. (1993), Blending reactivity and goal-directedness in a fuzzy controller, *i*n ‘Second International Conference on Fuzzy Systems’, Vol. 14, IEEE, San Francisco, CA, pp. 134–139.

Schlegel, C. (2004), Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach, PhD thesis, Universität Ulm.

Schlegel, C. & Wörz, R. (1999), The software framework smartsoft for implementing sensorimotor systems, *i*n ‘Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 99’, IEEE, Kyongju, Korea, pp. 1610–1616.

Schmidt, D. C. (1994), The adaptive communication environment: Object-oriented network programming components for developing client/server applications, *i*n ‘11th and 12th Sun Users Group Conference’.

Schmidt, D. C. & Suda, T. (1994), The service configurator framework, *i*n ‘IEEE Second International Workshop on Configurable Distributed Systems’.

Siegel, J. (2001), *Q*uick CORBA 3, OMG Press.

Simmons, R. (1994), Structured control for autonomous robots, *i*n 'IEEE Transactions on Robotics and Automation', Vol. 10, pp. 34–43.

Simmons, R. & Apfelbaum, D. (1998), A task description language for robot control, *i*n 'Conference on Intelligent Robotics and Systems', Vancouver Canada.

Stallman, R. (1999), The gnu operating system and the free software movement, *i*n C. DiBona, S. Ockman & M. Stone, eds, 'Open Sources', O'Rilley.

Szyperski, C. (1998), *C*omponent Software: Beyond Object-Oriented Programming, Addison-Wesley.

Tanenbaum, A. S. & S.Woodhull, A. (1987,1997), *O*perating Systems Design and Implementation, 1,2 edn, Prentice-Hall, Inc.

The Open Group (1997), 'OSF Distributed Computing Environment (DCE) '. http://www.opengroup.org/dce.

UML (2004). http://www.uml.org.

Vaughan, R. T., Gerkey, B. & Howard, A. (2003), On device abstractions for portable, resuable robot code, *i*n 'International Conference on Intelligent Robot Systems (IROS2003)', IEEE/RSJ, Las Vegas, Nevada, pp. 2121–2427.

W3C (2004), 'Extensible Markup Language (XML)'. http://www.w3.org/XML.

Wang, N., Rodrigues, C., Balasubramanian, K. & Schmidt, D. C. (2004), 'Tutorial on the corba component model (ccm)'.

Weisert, C. (2003), 'There's no such thing as the Waterfall Approach! (and there never was)'. http://www.idinews.com/waterfall.html.

Wenfeng, L., Dingfang, C., Christensen, H. I. & Orebäck, A. (2004), An architecture for indoor navigation, *i*n 'Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on', Vol. 2, IEEE, pp. 1783– 1788.