Autonomous Robots 14, 33–49, 2003

Evaluation of Architectures for Mobile Robotics

ANDERS OREBÄCK AND HENRIK I. CHRISTENSEN
Centre for Autonomous Systems, Royal Institute of Technology, SE-100 44 Stockholm, Sweden
oreback@nada.kth.se
hic@nada.kth.se

Abstract. In this paper we make a comparative study of some successful software architectures for mobile robot systems. The objective is to gather experience for the future design of a new robot architecture. Three architectures are studied more closely, Saphira, TeamBots and BERRA. Qualities such as portability, ease of use, software characteristics, programming and run-time efficiency are evaluated. In order to get a true hands-on evaluation, all the architectures are implemented on a common hardware robot platform. A simple reference application is made with each of these systems. All the steps necessary to achieve this are discussed and compared. Run-time data are also gathered. Conclusions regarding the results are made, and a sketch for a new architecture is made based on these results.

Keywords: robots, architecture, comparison, service robot, software

1. Introduction

Mobile robotics is gradually gaining momentum both for commercial applications and as vehicles for research in a diverse set of domains. Initial design of robots used a sense-plan-act paradigm for the control of robots, e.g., Albus et al. (1987) and Barbera et al. (1984). Today it is widely recognized that the most efficient strategy for design of robots is the use of a hybrid deliberate architecture, where there is a successful coordination between deliberation and reactive control. Such architectures are used in most of the robots reported in the recent literature, e.g., RAP (Firby, 1989), Xavier (Simmons, 1994), Saphira (Konolige and Myers, 1996), and 3T (Bonasso et al., 1997).

Given a fairly general agreement on basic architectural principles, one might expect that a common software basis is available and that such a system is used for exchange of algorithms across laboratories and for technology transfer. This is unfortunately not the case. A quick review of the literature and the set of commercially available platforms reveals that a wide variety of different software architectures are available. Typical architectures include Mobility from RWI (Real World Interface, 1999), Saphira from IS-Robotics/SRI (Konolige and Myers, 1996), TeamBots (Balch, 2000), RAP (Firby, 1989), Xavier (Simmons, 1994), and BERRA (Lindström et al., 2000). The use of a single common architecture would, however, have a tremendous potential, and one is left wondering why such a plethora of architectures are available and what are the characteristics of these architectures?

What can be learned from each of these architectures and is there a possibility to perform a synthesis of a common architecture, or are the approaches adopted too different to allow for synthesis of a single unified architecture? In this paper the overall requirements for a mobile robot architecture are outlined in terms of dimensions to be considered, and basic requirements in terms of control, modularity, software engineering, and run-time performance. Using these requirements it is possible to consider to what extent the most commonly used architectures fit these requirements. In addition, it is possible to draw a number of conclusions with respect to the basic methods needed for a unified architecture.

2. The Hybrid Deliberate Architecture

Traditionally the architectures of the robot software-systems were of a hierarchical type, highly influenced by the AI research of the time. This meant a system having an elaborate model of the world, using sensors to update this model, and to draw conclusions based on the updated model. This is often called the sense-plan-act paradigm. The systems did not perform very well, partly because of the difficulty in the modeling of the world, partly because of relying too much on inadequate sensors.

In 1987 Rodney Brooks revolutionized the field by presenting an architecture based on purely reactive behaviors with little or no knowledge of the world, see e.g., Brooks (1987). However, the purely reactive scheme does not perform well when performing complex tasks. A hybrid approach has since then been common among researchers, e.g., Arkin (1990).

2.1. The Layers

The hybrid systems are usually modeled as having three layers; one deliberate, one reactive and one middle layer. The bottom reactive layer performs repetitive calculations on raw or extracted data. These calculations should be carried out in near real-time for safety-critical considerations. The modules in this layer are normally stateless. The top deliberate layer handles planning, localization, reasoning and interaction with human operators. The middle layer, often called either the sequencer layer, or supervisory layer, bridges the gap between the deliberate and the reactive layers.

The reactive layer of a hybrid system is often behavior based. This means that the subsystem consists of separate behaviors, where each behavior has one specified non-complex task. The behaviors represents a tight coupling from the sensors to the actuators. The reactive layer also consists of sensors and actuators. Sensor 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 deliberate architecture can be seen in Fig. 1.

2.2. Examples of Hybrid Deliberate Architectures

The concept of the hybrid deliberate architecture is generally attributed to Arkin (1986, 1987). The approach was implemented in the AuRA architecture. AuRA has a hierarchical system consisting of a mission planner, a spatial reasoner, and a plan sequencer coupled with a schema controller which forms the reactive system. A schema manager controls and monitors the behavioral processes during execution. 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. As we see, this layout matches very well that of Fig. 1.

The XAVIER system (Simmons, 1994; O'Sullivan et al., 1997) has been developed at CMU. The system is composed of four layers with specific functions: task planning, path planning, navigation, and obstacle

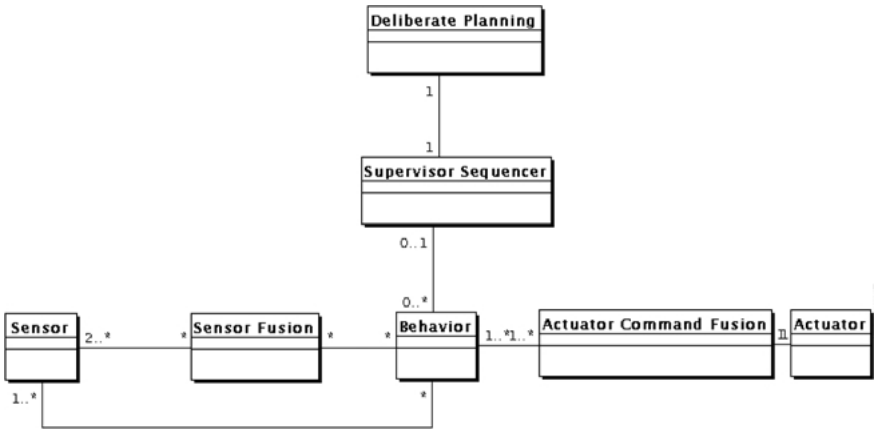

Figure 1. The hybrid deliberate architecture.

avoidance. The first two would fit into the top layer of the generic architectecture. Virtual sensors correspond to the sensor fusion module. The actuator command fusion is not needed since only the obstacle avoidance commands the actuators.

The architecture of the 3T (Bonasso et al., 1997) system corresponds very well to the generic architectecture. Principles for placing modules in the layers are based on four parameters, time, bandwidth, requirements from missions and modifiability. The reactive layer consists of skills which maps onto the behaviors in our generic model. The middle sequencing layer operates by using RAP (Firby, 1989).

The developers of the Saphira, see Section 5.1, architecture have not chosen to illustrate their system in terms of layers. Some sort of mapping to the generic model can anyway be made. Closest to the actuators are the *reactive behaviors* which would fit into our reactive layer. The sensors however, provide their data to modules that extract higher order data that are inserted into the *Local Perceptual Space LPS*. The LPS is used by other modules, regardless of their temporal or data abstraction level. As can be seen in Fig. 4, there are also goal and navigation behaviors. In the top we have the *PRS-lite* and a topological planner that would fit into the deliberate layer.

The BERRA architecture, see Section 5.3, just like 3T, closely resembles the generic model. The layering is however more of a descriptive issue and not as fundamental to the architecture as in 3T. 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*.

The TeamBots architecture, Section 5.2, has no real deliberate layer. One could place most of the Clay classes in the reactive layer. Clay also contains methods for sequencing of tasks. These would fit into the middle layer.

3. Dimensions of an Architecture

In order to define and design a common architecture for mobile robot applications it is essential to consider the key characteristics of the underlying system.

One can briefly summarize basic requirements as:

- Robot hardware abstraction
- Extendibility and scalability
- Limited run-time overhead
- Actuator control model
- Software characteristics
- Tools and methods
- Well documented

Each of these requirements are discussed in more detail below to clarify the basic requirements and provide a basis for evaluation of existing systems.

3.1. Robot Hardware Abstraction

Portability is a highly desirable design goal, since hardware is generally subject to change. A portable architecture 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 architecture should encapsulate these hardware specific commands into a generalized set of commands. 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.

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. See Fig. 2 for an example of hierarchical abstraction.

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. In all but a few circumstances, efficiency should be sacrificed since the software usually is more costly and outlives the hardware.

3.2. Extendibility and Scalability

Extendibility means here the support for adding new software modules as well as new hardware to a system. This is a very important aspect, since robotic systems in 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. Scalability

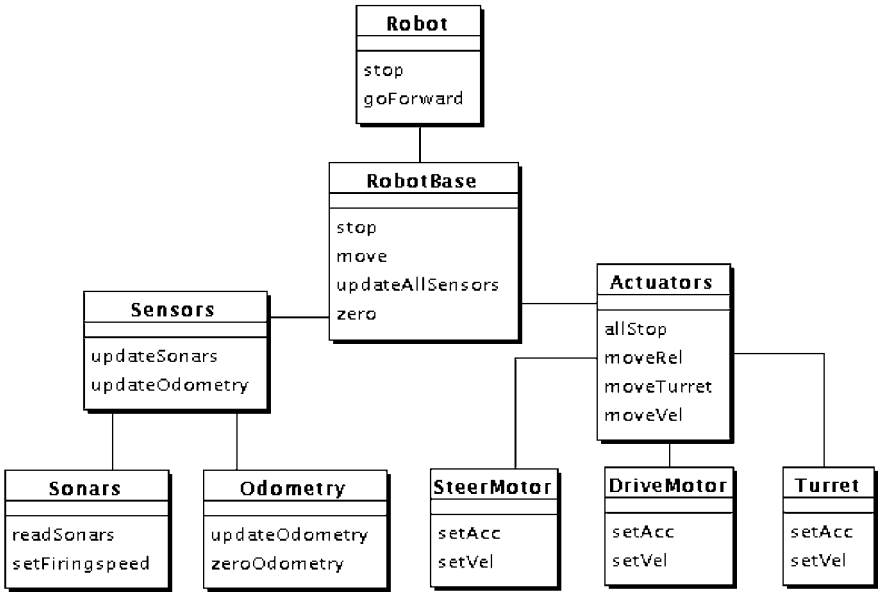

Figure 2. An example of hierarchical hardware abstraction.

is achieved by using efficient communication and well planned data flow. Avoiding bottlenecks and unnecessary constraints are also of importance. Scalability has to be addressed at the architectural level, by for instance providing means for distributing processes over several hosts.

3.3. Run-Time Overhead

The run-time environment is defined by a number of factors, including

1. memory requirements
2. cpu requirements
3. frequency
4. end-to-end latency.

In this context, frequency refers to the number of control-loops that the bottom layer runs per second. End-to-end latency refers to the maximum time it takes for a sensor reading to give impact on a hardware actuator command.

3.4. Actuator Control Model

In behavior based systems, the output from the individual behaviors need to be fused in order to produce one crisp actuator command. This can be done in a number of ways. Different kinds of simple superposition are however the most prevalent.

The actuator control model of the robot ultimately defines how the robot will move. A fast and smooth behavior is usually desirable. Unfortunately, this area has become a source of religious belief. Some argue for the behavior based approach, others for the fuzzy logic approach, and so on. Note though that most of these models do not exclude the others but can be used together in a system.

3.5. Software Characteristics

A robotic system must be robust and reliable and has to be prepared to handle unexpected situations. The architecture of the system must provide the framework for robust integration of required skills. For research and development purposes, an architecture should provide support for:

– A conceptual framework for reusability
– A clear distinction between levels of competence
– Simple integration of new components and devices
– Simple debugging
– Prototyping

The following are characteristics (Gabriel, 1993) that should be considered for a general software system.

The relative weights of these characteristics is a debated topic.

- Simplicity—the design must be simple, both in implementation and interface.
- Correctness—the design must be correct in all observable aspects.
- Consistency—the design must not be inconsistent.
- Completeness—the design must cover as many important situations as is practical.

Moreover, a good architecture is based on a formal theoretical ground. It is very important that the programmers who will be developing for the system, get a clear view of the architecture. Otherwise they may program the system away from the original idea, and the result may lose the cohesiveness it originally had. Optionally, users may be provided with an alternate simplified view.

3.6. Tools and Methods

Various tools can be used when constructing a software architectural system. As far as possible, standardized (by bodies such as ISO, ANSI, OMG, POSIX) tools should be used since proprietary tools tend to be short lived, and there is often skepticism among others to adopt them. As mentioned earlier, the hardware manufacturer provides the basic interface for accessing the hardware. This is likely a C language API. On top of this, functions or classes should be constructed to encapsulate and to separate the hardware specifics. These functions can in turn be called by higher level functions forming a powerful vocabulary.

With the advent of object oriented programming, portable and reusable modules can be programmed. This trend can also be seen in robotics. Earlier systems were almost exclusively programmed in the C language, with the exception of the AI community using LISP. Now, OO-based languages such as C++ and JAVA are the most popular tools. The next step will be to use component technologies such as CORBA (see below).

The governing ideas and principles of the architecture should be visualized in a graphic manner. A single graph can often clarify pages of explanatory text. UML (Booch et al., 1999), the Unified Modeling Language has become is a standard tool for both design and visualization. UML has the advantage that it is supported by tools that allow automatic synthesis, analysis and code-generation.

A key component in a robot system is a reliable and efficient communication mechanism. Modules need to exchange data and events. Run-time monitoring and error handling are also of great importance. Since in some systems, processes may be distributed on a number of computers, host-to-host communication should be transparent. This requires a broker object that can assist other objects in finding each other. A minimum requirement is to be able to activate and deactivate a module at run-time. Run-time reconfiguration of interconnections should also be possible. For scalability reasons, dynamic objects and process invocation on a *live-when-needed* basis is also highly desirable. Modern tools have factory patterns that aid in this. Data transfer can be initiated in either a *pull* or a *push* fashion.

In the past, several middle-ware standards such as sockets, Remote Procedure Calls (RPC), TCX (Fedor, 1993) or ACE (Schmidt, 1994) were developed. Recently, a number of new interesting frameworks have emerged. These lift the problem of distributed programming to a higher level of abstraction. The most well known of these are CORBA, ActiveX, and EJB. CORBA (Common Object Request Broker Architecture) (OMG, 2001) is a standard managed by the Object Management Group. Microsoft's ActiveX is a set of technologies based on COM (Component Object Model). Enterprise Java Beans (EJB), is a specification just like CORBA. All these technologies can interact with each other and they all adhere to the Interface Description Language (IDL). They provide component models and integration frameworks, which means that we can use them for the interaction as well as for the system descriptions.

3.7. Documentation

In order for an architecture to achieve success, that is used apart from locally, the documentation has to be rigorous. The following parts should be available in a proper documentation.

- The philosophy behind the architecture.
- A programmers guide.
- A users guide.
- An architecture reference manual that describes the API on each level.
- Code documentation.

The most difficult part of it is to keep the documentation current with the system. Documentation can be performed in a number of ways.

– Printed manuals.
– Web-pages.
– UML/class diagrams.
– Comments in the source.
– Javadoc, Doxygen

A combination of these is highly desirable. The JavaDoc/Doxygen utilities have the advantage of being embedded in the code. This way the documentation is always up to date, even if changes in the code still require corresponding changes in the comments. These documentation tools are excellent for code documentation but fails to capture the overall conceptual model for the system, consequently a combination of documentation strategies is needed.

4. Aim of Study

The aim of this study is to determine the characteristics of a range of common architectures according to the above dimensions. Conclusions can then be drawn that can aid in the design of new architectures.

One important issue to keep in mind, is that one must separate an architecture, from an implementation of the architecture. However, this kind of study can never be performed on a theoretical architecture, so what we really are comparing are implementations of architectures. Ideally, we should have a number of implementations of each one, in order to make more accurate judgments.

Another aim of this study is to provide guidance to readers who are about to choose between available robot systems. The following are examples of issues that have been addressed:

– Abstraction models
– Methods of control
– OS Support
– Language API
– Programmer efficiency
– Run-time efficiency
– Software overhead (size of code)
– Tools and methods
– Documentation

4.1. The Test-Case

The test-case considered was a simple service agent system. In this case, it meant that the operator could command the robot to navigate to anywhere in an office environment. For this purpose, the robot maintains a map of the area which is given a priori.

Figure 3. The Nomadics super scout.

The commands were issued either as keyboard instructions or, if possible, using speech commands. The map names were assigned human friendly names, such as ‘living room’, ‘dinner table’ etc. When possible, the robot would respond with synthesized speech. After arrival at the desired goal, the robot would be ready for another command. This scenario can easily be transferred into an interactive tour-guide robot. Then the modalities of speech input and output, as well as maintaining a more complex dialog, would be of greater importance.

All architectures were ported to one common robot platform, the Nomadic Super Scout (Nomadic, 2000), see Fig. 3. This is a small robot incorporating an industrial PC, Pentium 233 MHz, 64 MB memory and 6 GB hard disk. The scout has 16 Polaroid ultrasonic sensors and 6 tactile bumpers. A radio ethernet modem is available as well as facilities for audio I/O. The motherboard communicates to a controller-board through a serial port. The hardware control processor is a Motorola MC68332. Additionally, a TMS320C14 DSP is responsible for high-bandwidth motor control at a rate of 2 kHz. The scout is shipped with the RedHat Linux (currently 6.x) operating system and a C language API.

5. Architectures Considered

5.1. Saphira

Saphira (Konolige and Myers, 1996) is a robot control system developed at SRI International’s Artificial

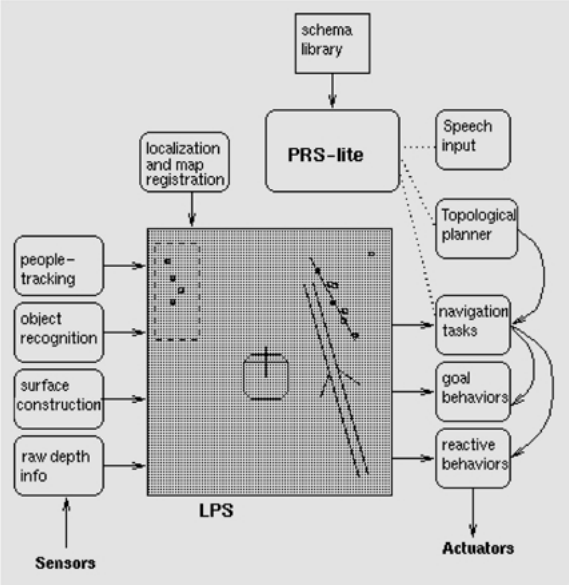

Figure 4. The Saphira architecture.

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 (Fig. 4). 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). 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 server that manages the hardware, and Saphira which is a client to this server.

5.2. *TeamBots*

TeamBots (Balch, 2000) is a Java-based collection of application programs and Java packages for single- and multi-agent mobile robotics research. TeamBots is an example of a system with a high degree of granularity. 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 teaching. 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 Fig. 5).

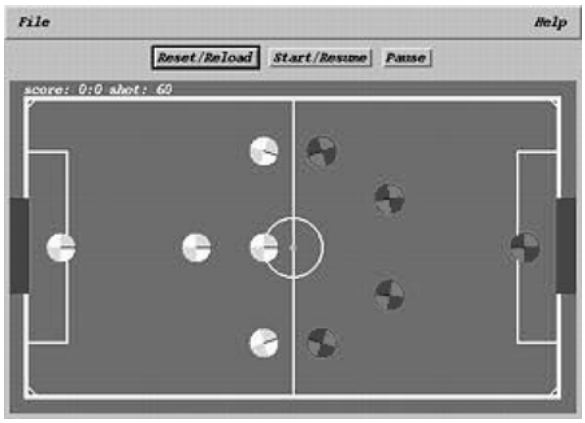

Figure 5. A TeamBots simulation.

5.3. *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 ACE (Adaptive Communication Environment) (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 and 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). See Fig. 6 for a diagram view of BERRA.

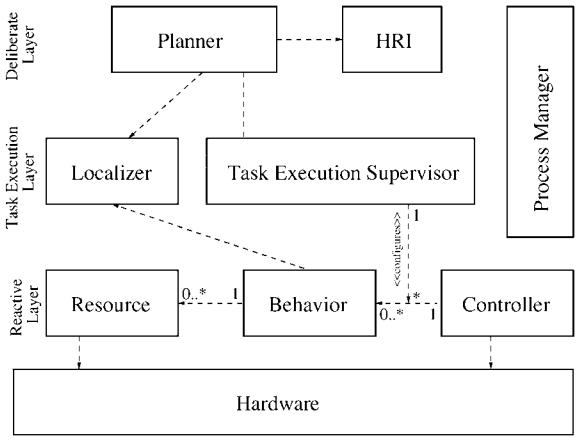

Figure 6. The BERRA architecture.

6. Evaluation of Software Characteristics

6.1. Programming the Test System

6.1.1. OS, Language Support and Std Libraries

6.1.1.1. Saphira. Saphira supports the most number of operating systems. Several flavors of UNIX and also MS windows is supported. The graphical user interface is based on Motif, so Motif libraries are required for developers. There are several 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* (Konolidge, 1997). It has a C-like syntax with semantics based on finite state machines. A part of Saphira is written in LISP. There are signs of speech feedback in the code, but it seems to have been dropped at some time, since we never managed to make the client communicate with the server.

6.1.1.2. TeamBots. TeamBots is entirely built in JAVA. The source has been carefully grouped into classes that form a fine grained collection of components. This means that TeamBots can 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 (a special Java Class).

6.1.1.3. BERRA. BERRA has been tested on Linux and Solaris. Theoretically, all platforms supported by ACE can be considered. The current version relies on ACE version 5.x. GNU CC version 2.95.x is recommended. In order to utilize speech, either Festival (Black and Taylor, 1997) or a DoubleTalk speech card is required. The speech recognition system is Esmeralda (Fink, 1999), which is a (non-free) system from the University of Bielefeldt. Some vision functions are based on Blitz++ (Veldhuizen, 1998). BERRA is written entirely in C++. Base classes and templates form the basis of the code structure.

6.1.2. Communication Facilities and Performance. Only one of the studied architectures (BERRA) are of the multi-process type and therefore make use of inter-process communication. BERRA bases all communication on sockets using ACE (Adaptive Communication Environment). BERRA supports both UNIX and INET socket protocols. TeamBots includes a communication package, and it is used for communication between robot instances.

6.1.3. Hardware Abstraction. Hardware abstraction in Saphira is performed in the robot server. This means that there is only one level of abstraction. Client processes can never address the lower level hardware.

In TeamBots, the hardware abstraction is very advanced. The interface *SimpleInterface* is implemented for all robots. It is also possible to address the hardware at a lower level. The same applies for control. The class *ControlSystemS* is a super-class for all control systems.

BERRA has basically one high-level abstraction. Methods exist for controlling the hardware at lower level, but this is achieved by parameterizing the high level commands using a difficult syntax.

6.1.4. Porting and Application Building

6.1.4.1. Saphira. Porting of the hardware level code for Saphira was a major task. We were supplied with the source code used for the Pioneer platform server. This consisted of a large number of C files with unclear inter-dependencies. Since we were not supplied with the source code of the Saphira client, direct observation of the robot was the only means of debugging.

Regarding the application, all of the relevant behaviors were already present in the system. The construction of the map and incorporating it into the LPS was also fairly straightforward. Localization then worked right out of the box. By using Colbert, a reasonable text based user interface was created. However, the system lacks route planning. The Colbert language might have

been used for this, but it was unclear to us how that could be done. This means that with Saphira we could only navigate to a point immediately accessible from the current location.

6.1.4.2. TeamBots. Porting of TeamBots to the Nomadic Scout was a straightforward process. There existed already a version for Nomadic Nomad150. The main difference is that the Scout has a differential-drive system whereas the Nomad150 is equipped with a synchro-drive.

Building of the application turned out to involve quite a lot of work. TeamBots is primarily constructed for pure reactive and non-interactive systems. Thus, we had to write high level code that could take commands from the operator and then activate the correct behaviors. Most of the relevant behaviors existed in the distribution, but a scheme of behavior states was added to enable switching between these. A simple route planner was also built. Something completely missing in the original TeamBots distribution are methods for localization.

6.1.4.3. BERRA. BERRA previously ran on Nomad200 and Nomad4000. The abstraction for these platforms can accept calls from multiple clients. The Scout can only be accessed by one client. This posed a problem for BERRA since it consists of multiple processes, where several processes need to access the hardware. A major rewrite was performed so that only one process of BERRA would access and manipulate the robot hardware. This modification will make it easier to migrate BERRA to other platforms in the future. All code needed for the application was already present in the system.

6.1.5. Documentation. Saphira is clearly the best documented system. There are plenty of papers describing the philosophy behind the architecture. There is also a *Saphira Manual* that includes a user's guide, a programmer's guide and an API reference. The code however, is poorly documented. For TeamBots on the other hand, there is only the javadoc generated class documentation. The code is well documented. A book is currently being written. For BERRA, there are papers describing the design philosophy. All other documentation is web-based. There are quick guides for the user as well as for the programmer. Javadoc class descriptions are also available. The code is for the most part well documented.

6.1.6. Programmer Efficiency

6.1.6.1. Saphira. The great flexibility of programming that Saphira offers, is both a strength and a weakness. The vast amount of documentation and options can make a novice user weary. We believe that it is the intention of the authors that all application programming should be made with Colbert. This is a practical and easy language to learn. One 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. Compilation is however fast and the resulting file sizes are small.

6.1.6.2. TeamBots. TeamBots gives the option of either using or not using Clay. For the novice, the non-clay examples provided are easier to understand than the Clay ditto. It should be understood that Clay is to be preferred since it provides a large number of classes and a elegant behavior blending mechanism. JAVA provides a fairly fast compilation environment with moderate file-sizes. No special debugging arrangements are included.

6.1.6.3. BERRA. BERRA has the most number of features. It is a very complex system, and is considered a difficult system to learn. It consists of a very large number of directories and files. This makes it hard to grasp and to understand. Since it consists of several concurrent processes, the system is difficult to debug. Because of ACE, and the use of templates, compilation is time consuming. File sizes tend to grow large as well. Some provisions for debugging exists. A debug level can be set prior to starting each process. Adding new behaviors and sensors is though rather easy, since templates are provided for most functionalities.

6.2. Run-Time Evaluation

6.2.1. Saphira. Starting Saphira is easy. One first starts the robot server, followed by the Saphira client which then connects to the server. The operator can then start and stop behaviors and tasks directly in the graphical interface with the built-in in Colbert interpreter. The syntax for the text commands are however cumbersome. Libraries can also be dynamically loaded in run-time. The interface updates the position of the robot in a map of the environment, as well as other information, see Fig. 7.

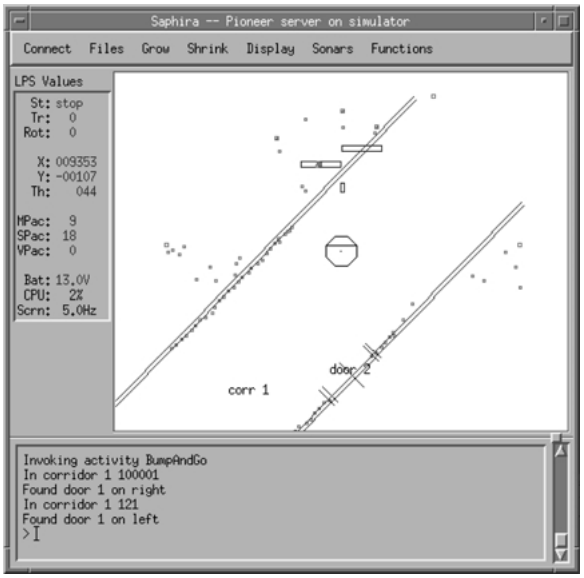

Figure 7. The Saphira graphical user interface.

Saphira does not demand great system resources, a little more then 10 MB is sufficient. The response time was however quite high. It took on average 0.6 seconds from a sensor reading until it gave a response to the actuators. The initial control design has a frequency of 10 Hz. It turns out that packets sent from the hardware have a limited size, so that only 8 sonar readings can be sent in one packet. However, this is complicated by the fact that the full update time of all the sonars takes about 500 milliseconds.

A major problem with Saphira was the inaccuracy of the localization system. It would inevitably loose track of its position after approximately 10 m. Thus the system must be augmented with methods for localization to provide a satisfactory performance.

6.2.2. TeamBots. The interface we developed is rather simple. After starting the system, the operator basically types in the name of the desired goal. The same speech recognition system used for BERRA was also ported to TeamBots, which gave an alternate method of command. No graphical user interface is available when running a real robot. One problem regarding the control of our TeamBots robot, turned out to be the tuning of parameters of the behaviors. These have a very high impact on how the robot behaves, and we could neither find a scientific, nor an intuitive way to tune these. Since the system lacked a localization module, the robot could not travel far before loosing its position. This because of the very accuracy of the odometry encoders of the Scout. The run-time system consumed about 8 MB of memory. The loop frequency was measured to 5 Hz. 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.

6.2.3. BERRA. BERRA is started by a shell-script that deploys the system processes in a cascade like manner. This works very well, as long as everything goes as planned. If however a process for some reason would die, the system is left in an undefined state and must be totally restarted. The latency from a sensor reading to a corresponding actuator command was measured to as low 0.17 seconds. This is very fast, but the fact that a full update of the sonars takes 500 ms has to be taken into account. The system demands 36 MB of run-time memory.

The functional aspects get a very high mark. When running BERRA, the scout can traverse the department for hours fulfilling navigation requests. Unfortunately, BERRA lacks a graphical user interface.

7. Conclusion on Evaluation

Although the compared systems are quite different, some conclusions can be drawn.

Hardware abstraction was handled well in all systems. It is evident that as long as the robot platforms do not deviate too much, one should go the extra mile in order to achieve portability. Run-time overhead is important to the scalability factor, but is not as important today as it has been. Modern hardware is fast and cheap. Optimization should only be done where it really is needed.

Two of the systems are not really designed for interactive tasks. In Saphira, the example programs typically perform a fixed task, for example patrolling an area. TeamBots has an even more *start and observe* methodology. In the kind of applications applicable to service robotics, human interaction and flexible task planning and switching is of great importance. However, the high level scripting language present in Saphira (Colbert) is an asset that greatly enhances productivity. The conclusion is that the deliberate layer should be formed in the

spirit of BERRA, with the addition of a task scripting language.

Granularity is an important aspect of a system. The BERRA design features very large building blocks, essentially on the behavioral process level. TeamBots offers a very fine-grained system. This fine-grained scheme promotes flexibility, but it is at the expense of the novice or casual user. Saphira is somewhere in between, thanks again to the Colbert language. The conclusion here is that the medium level of granularity is a reasonable approach.

A graphical user interface, as the one Saphira includes, offers a valuable view of the system as well as a more joyful experience. This interface also lets the operator inspect some of the internal states and variables of the system, and this is a very convenient feature. A new system that wants to be widely adopted has to appeal to users of different skill levels, and a flexible graphical user interfaces is here essential.

Distributing processes over several hosts is a powerful way to provide scalability, extendibility and load balancing. Sensors could for instance be used that are situated in the environment or other agents. On the flip side, debugging becomes a difficult task (compare BERRA), as the overall system complexity grows. This course needs to be pursued, but careful selection of tools are here of utmost importance. As mentioned earlier, we think that the component technologies such as CORBA is interesting in this context. By replacing components, different actuator control models can also be deployed.

Only one of the architectures provides support for multi-agent systems. Obviously this area is getting more and more popular, and a unified architecture must cater for this very early in the design phase.

Saphira is the only architecture that mentions bindings for different languages. We think that this is an important feature as well.

In Table 1, a table outlining the findings in this chapter, as well as more information, is presented.

8. Discussion and Future Work

An initial question in this study was "why are architectures not shared across laboratories?" Our study of three architectures clearly demonstrates that the porting of different architectures in most cases is a major undertaking due to lack of adequate abstraction over the hardware platform. It would be desirable to have a Hardware Description Language (and associated abstraction model) that would allow easy interfacing to a variety of platforms. Initially this might be limited to the mobility part and sensors for the systems, but in due course such a facility should be extended to other parts as well. As seen in the evaluation, it is also a major task to provide similar facilities for behaviors and higher level functionalities due to lack of a common framework for specification of behaviors, supervisory system and task decomposition. Consequently, there is a need to define a common framework to provide the needed functionalities for porting of systems across platforms and laboratories. Below is a discussion of some of the issues to be considered in the design of such a basis.

8.1. Component Technology

As stated earlier, a highly modular design is desirable. Component technologies address many of the issues considered in this paper. One definition of *components* is *software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system* (Szyperski, 1998).

A component based solution 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.

The concept of interfaces as opposed to multiple levels of inheritance help to resolve problems known to object-oriented programmers.

Component based approaches have been initialized elsewhere within robotics. The National Institute of Standards and Technology (NIST), has been developing specifications of software components for intelligent control, a domain that includes robotics. In Messina et al. (1999), a three-stage component specification approach is proposed.

Step 1 (Generic categories for the specifications). Initially, a set of appropriate questions are defined that are relevant to the particular class of components. These questions might include:

- What problem is this component intended to solve?
- What is the input data?

Table 1. Evaluation table.

	Saphira 6.2	TeamBots 2.0	ISR BERRA 2.0
OS			
Linux	Yes	Yes	Yes
MS windows	Yes	Yes	No
Solaris	Yes	Yes	Yes
MacOS	No	Yes	No
IRIX	Yes	Java 1.2	No
FreeBSD	Yes	Java 1.2	No
NetBSD	Yes	Java 1.2	No
DEC OSF	Yes	Java 1.2	No
Supported platforms			
Nomadics	Yes (now)	Yes	Yes
Pioneer	Yes	No	No
Cye	No	Yes	No
Main prog. language	C	Java	C++
High-level language	Colbert (C-like)	Java API	C++ API
Software tools requirements	gcc, Motif	Java 1.2	gcc 2.95, ACE 5
Distribution			
Binary	Yes	Yes	No
Source	Limited	Yes	Yes
Installation			
Installation program	No	No	No
User friendliness	Good	Good	Poor
License			
Free	No	Yes	Yes
Open development	No	Yes	Yes
Free of charge	Limited	Yes	Yes
Graphics			
GUI	Yes	Yes (simulation)	Yes
Graphics programming	Yes (limited)	Yes	No
HRI			
Text	Yes	Yes	Yes
Speech	No	No	Yes
GUI	Yes	No	Yes
Palm	No	No	Yes
Multi agent support	No	Yes	No
Multi host (distributed)	No	No	Yes
Multi process	Threads	Threads	Multi-process
Data flow paradigm			
Push	Yes	No	Yes
Pull	Yes	Yes	Yes
Platform portability			
Hardware abstraction	Good	Very good	Very good
HW deps. in one file	Yes	Yes	Yes
Sensor extension cap.	No	Good (JNI)	Very good

(*Continued on next page.*)

Table 1. (*Continued*).

	Saphira 6.2	TeamBots 2.0	ISR BERRA 2.0
Functions			
Route planning	No (yes?)	No	Yes
Localization/accuracy	Yes/poor	No	Yes/very good
Sensor support			
Sonar	Yes	Yes	Yes
Laser	Yes	No	Yes
IR	No	No	Yes
Camera	Yes	Yes (newton)	Yes
Bumper	Yes	Yes	Yes
Documentation			
Manual	Yes	No	No
Webpages	No	Yes	Yes
Class diagrams	No	Yes	Yes
Book	No	Yes	No
Articles	Yes	No	Yes
Learning curve	Difficult	Easy	Difficult
Simulation	Yes	Yes	Limited
Behavior coordination	Fuzzy logic	Various	Vector/histogram addition
Stability	Very good	Very good	Very good
Timing aspects			
Realtime support	No	No	No
Sensor Actuator latency	0.6	0.40	0.17
Frequency	2	2.5	6
Bandwidth			
Sensor to behavior	4 KB/s	OS limitation	OS limitation
CodeSize			
Filesize, dummy behavior	200 B	100 B	1.7 MB
Runtime memory size			
Dummy behavior	24 B	40 B	3.5 MB
Complete system	11 MB	45 MB	36 MB
Harddisk footprint	8.5 MB	12 MB	250 MB

- What is the output data?
- How robust is the component?
- What kind of computing hardware is required?
- How does the component perform against available benchmarks? etc.

Step 2 (Natural language instantiation of the specification). In this step, the questions from step one are answered for the particular component. The natural language description can then be made public. Search engines can locate candidate components and users can browse through the descriptions as they would through a hardware components catalog.

Step 3 (Formal language instantiation of the specification). At the final stage, the component is described in a formal language in order to provide an unambiguous specification and to support simulation and verification.

OPEN-R (Fujita and Kageyama, 1997) is an acronym for Open architecture for Robot entertainment. It is a proposed standard for interfaces of hardware and software components in entertainment

robotics. Researchers at SONY have developed this in order to promote research in intelligent robotics by providing off-the-shelf components and basic robot systems. The aim for the standard is to achieve system extension and reconfiguration capabilities for mechanical, electrical, and software components. The following features are suggested:

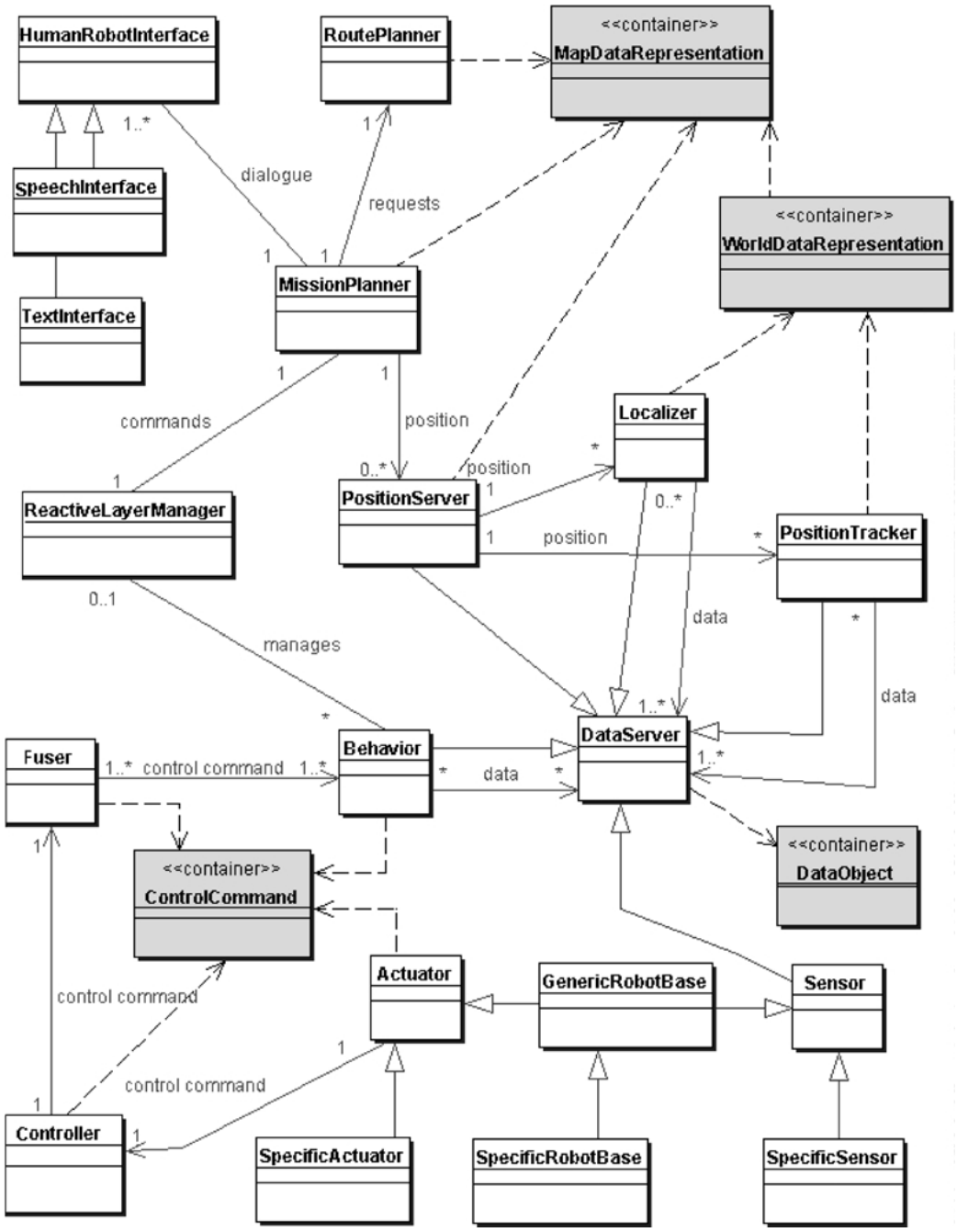

Figure 8. A proposed generic design.

1. common interfaces for various components such as sensors and actuators.
2. mechanisms for obtaining information on functions of components and their configurations.
3. a three-layered architecture for hardware adaption, system services, and applications.

The software platform is based on Apertos, an object-oriented real-time distributed operating system. In Fujita et al. (1999), a reconfigurable robot platform is presented. By using configurable physical components, reconfiguration of the robot style (e.g., legs or wheels), as well as hot plug-ins, are enabled.

8.2. Deliberate Components

One common mistake we wish to avoid is the one to combine the functions of a user interface, a mission planner, and a route planner (BERRA and partly Saphira). All these are kept separate in the new architecture. In the case we have more than one mode of operator input, we make sure that we isolate even those components.

Another important part of a mobile robot is that of localization. Algorithms used for finding the current pose and for tracking the pose are usually not related. Therefore, those functions should be divided into different modules. Apart from that, one simple server should be available that can supply clients with the current pose.

Representations of data are often overlooked in the architecture literature. A new system must put equal emphasis on this as on the traditional "*boxes and arrows*". Two examples of representations are map data and sensor data.

The new system will also support multiple agents. Basically all modules should be addressable from the outside. Some people argue that a component such as a sensor, should only be reached from outside the robot by going through a central deliberate component. However, we have seen that specially constructed chains of command introduce unwanted complexity.

8.3. Reactive Components

Another common mistake is to integrate the actuator controllers with the fusing mechanisms (BERRA, Saphira). We will not repeat this mistake. The hardware abstraction is based on the recommendations from Section 3.1. All sensors have the same interfaces and the same goes for all actuators.

8.4. Future Work

The first step of the design phase is to construct an object model. The next step is to identify a set of interfaces. These interfaces should be reused wherever possible throughout the system. The design of these interfaces should stabilize quite quickly, as an interface can never be changed.

The high-level functional building-blocks necessary in a robot architecture have been identified in Fig. 8. Although this is an UML-diagram showing classes, base-classes and inheritance, it should not be regarded as a blueprint for a new system. The class diagram was chosen in lack of better ways to visualize the different modules.

The base-class *DataServer* in Fig. 8 should really be an interface described in IDL.

One great risk when designing a component framework is that it will be hard to maintain layers and levels of competence. The system may well grow into a complex soup of components.

9. Summary

We started in this paper by discussing basic requirements for a software architecture for mobile intelligent robots. In order to gather comparative data, a reference application was programmed by using three different available architectures. The requirements were checked against the three implementations. In the end, conclusions were drawn that reflect upon the results of the study. A number of principles are formulated that will guide in the design of a new architecture.

Acknowledgments

This research has been sponsored by the Swedish Foundation for Strategic Research through its Centre for Autonomous Systems and the EU through the "Open Robot Control Systems—OROCOS" IST-2000-31064. This support is gratefully acknowledged.

References

Albus, J., McCain, H., and 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., and Christensen, H. 1999. ISR: An intelligent service robot. In *Intelligent Sensor Based Robotics*, Springer Verlag: Heidelberg.

Arkin, R.C. 1986. Path planning for a vision-based autonomous robot. In *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, also Ph.D. Dissertation, Department of Computer and Information Science.

Arkin, R.C. 1990. Integrating behavioral, perceptual, and world knowledge in reactive navigation. In *Robotics and Autonomous Systems*, Vol. 6, pp. 105–122.

Balch, T. 2000. TeamBots. Available at world wide web, www.teambots.org.

Barbera, A., Albus, J., Fitzgerald, M., and Haynes, L. 1984. RCS: The NBS real-time control system. In *Robots 8 Conference and Exposition*, Detroit, MI.

Black, A. and Taylor, P. 1997. Festival speech synthesis system: System Documentation. Human Communication Research Centre, University of Edinburgh, 1.1 edition.

Bonasso, R.P. et al. 1997. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificiall Intelligence*, 9(2): 237–256.

Booch, G., Rumnaugh, J., and Jacobsen, I. 1999. *The Unified Modeling Language User Guide*, Object Technology Series, Addison-Wesley: Reading, MA.

Brooks, R. 1987. A hardware retargetable distributed layered architecture for mobile robot control. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Fedor, C. 1993. TCX—An interprocess communication system for building robotic architectures. Carnegie Mellon University, Pittsburg, Pennsylvania.

Fink, G.A. 1999. Developing HMM-based recognizers with ESMERALDA. In *Lecture Notes in Artificial Intelligence*, Vol. 1692, Springer: Berlin, pp. 229–234.

Firby, R.J. 1989. Adaptive execution in complex dynamic worlds. Ph.D. Thesis, Yale University.

Fujita, M. and Kageyama, K. 1997. An open architecture for robot entertainment. In *Proceedings of the First International Conference on Autonomous Agents*, pp. 435–442.

Fujita, M., Kitano, H., and Kageyama, K. 1999. A reconfigurable robot platform. *Robotics and Autonomous Systems*, 29(2/3): 119–132.

Gabriel, R.P. 1993. Lisp: Good news bad news. Available at world wide web, http://www.ai.mit.edu/docs/articles/good-news/good-news.html.

Konolidge, K. 1997. COLBERT: A language for reactive control in Saphira. In *German Conference on Artificial Intellgence*, Freiburg.

Konolige, K. and Myers, K. 1996. *The Saphira Architecture for Autonomous Mobile Robots*, SRI International.

Lindström, M., Orebäck, A., and Christensen, H. 2000. Berra: A research architecture for service robots. In *International Conference on Robotics and Automation*.

Messina et al. 1999. Component specifications for robotics integration. *Autonomous Robots*, 6(3): 247–264.

Nomadic. 2000. Nomadic Technlogies Inc. Available at world wide web, www.robots.com.

OMG. 2001. CORBA. Available at world wide web, www.corba.org.

O'Sullivan, J., Haigh, K.Z., and Armstrong, G.D. 1997. *Xavier Manual*, internal manual.

Real World Interface. 1999. Mobility robot integration. Available at world wide web, Tech sheet, www.isr.com/rwi/.

Saffiotti, A., Ruspini, E., and Konolige, K. 1993. Blending reactivity and goal-directedness in a fuzzy controller. In *Second International Conference on Fuzzy Systems*, San Francisco, CA, Vol. 14, pp. 134–139.

Schmidt, D.C. 1994. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *11th and 12th Sun Users Group Conference*.

Schmidt, D.C. and Suda, T. 1994. The service configurator framework. In *IEEE Second International Workshop on Configurable Distributed Systems*.

Simmons, R. 1994. Structured control for autonomous robots. In *IEEE Transactions on Robotics and Automation*.

Szyperski, C. 1998. *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley: Reading, MA.

Veldhuizen, T.L. 1998. Arrays in Blitz^{++}. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Springer-Verlag: Berlin.

Anders Orebäck is carrying out his Ph.D. studies at the Centre for Autonomous Systems (CAS) at the Royal Institute of Technology (KTH), Stockholm, Sweden. He has primarily been working with architectural and integration issues within the field of service robotics. He was one of the lead designers of the ISR (a.k.a BERRA) system developed at CAS. He earned his M.S. in Electrical Engineering at KTH. Before pursuing his Ph.D. studies, he worked with VR and other emerging technologies in a small startup in the entertainment industry.

Henrik I. Christensen is a professor of computer science specializing in autonomous systems at the Royal Institute of Technology

(KTH), Stockholm Sweden. In addition he is the director of the Centre for Autonomous Systems (CAS) at KTH. He also serves as the founding chairman of the European Robotics Network—EURON. He received M.Sc. and Ph.D. degrees from Aalborg University in 1987 and 1989, respectively. His has published more than 120 contribution on robotics and computational vision. In addition he serves on the editorial board of several journals incl. IJRR, IEEE PAMI, IJPRAI and AI magazine.