Towards Component-Based Robotics*

Alex Brooks, Tobias Kaupp, Alexei Makarenko and Stefan Williams
ARC Centre of Excellence in Autonomous Systems (CAS)
Australian Centre for Field Robotics
University of Sydney, NSW 2006 AUSTRALIA
{*a.brooks, t.kaupp, a.makarenko, s.williams*}*@cas.edu.au*

Anders Orebäck
Centre for Autonomous Systems
Royal Institute of Technology
SE-100 44 Stockholm, SWEDEN
oreback@nada.kth.se

***Abstract*— This paper gives an overview of Component-Based Software Engineering (CBSE), motivates its application to the field of mobile robotics, and proposes a particular component model. CBSE is an approach to system-building that aims to shift the emphasis from programming to composing systems from a mixture of off-the-shelf and custom-built software components. This paper argues that robotics is particularly well-suited for and in need of component-based ideas. Furthermore, now is the right time for their introduction. The paper introduces Orca – an open-source component-based software engineering framework proposed for mobile robotics with an associated repository of free, reusable components for building mobile robotic systems.**

***Index Terms*— Distributed Robotics, Robotic Architectures, Component-Based Design, Software Re-use, Standardization**

I. Introduction

Component-Based Software Engineering (CBSE) is an approach that has arisen in the software engineering community in the last decade or so. It aims to shift the emphasis in system-building from traditional programming to composing software systems from a mixture of off-the-shelf and custom-built components [2][7][14][5][3].

For a long time, the manufacturing industry has allowed for replaceable parts by building them to a pre-defined standard. Similarly, CBSE offers developers the opportunity to source existing plug-in software components, rather than building everything from scratch. In addition, CBSE offers significant software engineering benefits by enforcing modular systems, which helps control dependencies, reduce maintenance costs and increase system flexibility and robustness.

The thesis of this paper is that the field of mobile robotics would benefit from adopting a component-based approach. A general component-based robotics framework should:

1) Be designed from the beginning with CBSE ideas in mind.
2) Adopt an open-source model: a requirement for collaboration between academic institutions.
3) Maintain an online repository of useful independently-deployable components, each with sufficient documentation.
4) Not be intimately tied to a particular transport mechanism such as CORBA, TCP/IP, RS232, or similar.
5) Not prescribe a particular architecture. The architecture should be defined instead by the set of components that are chosen and the manner in which they are composed.
6) Emphasise extensibility. Users should easily be able to extend the framework.
7) Allow for the building of systems ranging from single-vehicle architectures to distributed, including decentralised, systems. The latter requires that there be no unique central entity upon which all components rely.

This work is supported by the ARC Centre of Excellence programme, funded by the Australian Research Council (ARC) and the New South Wales State Government.

The remainder of this paper is organised as follows. Section II discusses related work. Section III defines components and component-based software engineering, and explains why they are useful. Section IV describes the application of CBSE to the field of mobile robotics, particularly (a) why robotics is in need of a component-based solution and (b) why robotics is especially well-suited technically to a component-based approach. Section V describes Orca – an open-source component-based software engineering framework proposed for mobile robotics. Section VI provides a detailed comparison of Orca with the Player framework. Finally Section VII concludes.

II. Related Work

Three major component models exist today, all of which are deemed unacceptable for mobile robotics. Microsoft's COM+ [6] is proprietary and not portable across operating systems, Sun's Enterprise JavaBeans [1] is restricted to Java, and the CORBA Component Model (CCM) [17] lacks sufficiently mature implementations.

The idea of applying CBSE principles to the field of mobile robotics is not new; work has been initiated elsewhere to introduce the concept of components [8] [11] [12]. To the authors' knowledge, there is no explicitly component-based framework that fulfils all the requirements in Section I, particularly the requirement for a repository of independently-deployable components.

Several existing open robotics projects do maintain large repositories of well-tested re-useable implementations of algorithms for mobile robotics. The two most popular are CARMEN [10] and Player [16].

CARMEN, the Carnegie Melon Robot Navigation Toolkit, communicates between components using IPC. It is designed specifically for single-robot systems.

While Player was not designed explicitly with CBSE principles in mind, it employs many component-based ideas. As the most successful mobile robotics re-use project to date, Player is arguably becoming the de-facto standard. Therefore it is compared in more detail against the current work in Section VI.

III. Component-Based Software Engineering

Component-Based Software Engineering is said to be primarily concerned with three functions [7]:

1) Developing software from pre-produced parts
2) The ability to reuse those parts in other applications
3) Easily maintaining and customizing those parts to produce new functions and features

The remainder of this section defines components, contrasts them with objects, then outlines the benefits of component-based systems.

A. Components

While definitions of the term *component* vary, we adopt Szyperski's [14]. Collins-Cope identifies components as having the following four properties [3]:

1) A component is a binary (non-source-code) unit of deployment
2) A component implements (one or more) well-defined interfaces
3) A component provides access to an inter-related set of functionality
4) A component may have its behaviour customized in well-defined manners without access to the source code.

Components can be put together in various configurations to form a system. This flexibility comes from modularity, enforced by the contractual nature of the interfaces between components. Interfaces and their specifications are viewed in isolation from any specific component that may implement or use the interface. In addition, the contractual nature of interfaces allows components on either side of the interface to be developed in mutual ignorance. Designing each component and interface in isolation allows a component that implements an interface to be seamlessly replaced with a different component.

Examples of successful component-based systems include plugin-based architectures such as Netscape's web browser and Apple's QuickTime. Another example is Unix's pipe-and-filter composition where binary, independently-deployable components communicate via the framework provided by the Operating System [5].

B. Components Versus Objects

Objects are not candidates for components. Objects exist at run-time, whilst components are binaries that are deployed [3].

Objects can be incorporated into a system only with considerable glue code to perform functions such as moving data between objects, which must be provided by an experienced programmer. Objects are not atomic units of deployment; they may rely on the presence of other objects. In contrast, deploying components should simply be a matter of choosing them, possibly compiling them, configuring them, then pointing each one to the components with which it should communicate.

Additionally, classes are generally much more fine-grained than components. Components correspond to readily-understood entities, whereas classes are more likely to have purposes and details that are understood only by those who know the intricacies of the system [14]. As an example, a component that implements a particular algorithm or abstracts a particular piece of hardware may contain many classes. While the purpose and function of the algorithm or hardware is readily-understood, an understanding of the myriad of classes that might implement the component is likely to rely on substantial knowledge of the details of the system.

C. Benefits of Component-Based Systems

1) Modularity: Using component technology unavoidably leads to the adoption of principles of modularity. The result is a system with only controlled, explicit dependencies and the software engineering benefits that this brings such as a flexible, reconfigurable system, the ability to distribute the work easily amongst individuals and the ability to re-use modules across projects. The benefits of modular designs are well-known and the idea is not new. Indeed, it is certainly possible to design a modular system without the use of components. However, despite the best intentions, the vast majority of software solutions today are not modular [14].

2) Re-Usability: The primary motivation behind CBSE is re-use, reducing cost and time to market without compromising software quality. This is achieved through re-using existing components that are either sourced externally from third parties or developed in-house [3]. Szyperski highlights the fact that his definition includes an entirely non-technical aspect: "A software component can be deployed independently and is subject to composition by third parties" [14]. Due to the properties listed above, components can be deployed without satisfying complex and intricate dependencies, by third parties who are not familiar with the source code.

Code-copying, of functions or classes or snippets, is not re-use. The Re-use/Release Equivalency Principle states that the granule of re-use is the granule of release [9]. When a new software version is released, the user should be able to replace the entire binary. When code is copied, the copier owns the (different) code as soon as it is modified and therefore takes responsibility for the most costly part of the software development cycle, namely maintenance. Instead, re-use happens only when the re-user doesn't have to look at the source code (other than the public portions of header files). This is possible for libraries and components.

A result of these properties is that components are much more likely candidates than objects to be traded in markets of inter-changeable parts. Early proponents of object-oriented programming predicted markets where users could

browse and purchase from catalogs of re-useable software objects as easily as a mechanical engineer can purchase interchangeable car parts [4]. Unfortunately this vision has never come to fruition for objects, however there is evidence that components are succeeding where objects have failed in this regard: www.componentsource.com is an example of such a market.

D. Potential Criticisms

A potential objection to the use of components and standardization in general is that standard components are only useful if the rest of a system conforms to the same standards of interaction. If not, users still have to manually extract the parts they need to make use of any given component. This is true only when the proportion of users who adhere to the standard is small. Standards become powerful when a critical mass of users arise, such that people choose to conform due to the advantages that it brings, particularly the number of components with which they can interact.

IV. Component-Based Robotics

A. Why Mobile Robotics Needs a Component-Based Solution

Despite substantial progress in mobile robotics research in the last decade, it still takes considerable effort to produce a robust, working mobile robotic system with a complete set of core competencies such as localization, mapping, path-planning, waypoint-following, obstacle avoidance, etc. The effort required to build such a system from scratch is certainly beyond what can be expected from a single PhD student. This creates a formidable barrier to entry for new research labs.

We would argue that the single largest obstacle to building a robust mobile robotic system today is the application-level software. Consider the other elements of a robotic system: Many different mobile platforms are available commercially, for both indoor and outdoor applications. A variety of sensors is available; lasers, sonars, inertial measurement units and GPS units can also be purchased off-the-shelf. The same is true for processors and wireless communication networks. Operating systems and software development tools can be downloaded for free. Even the algorithms are not hard to obtain, since solutions for all the core competencies listed above can be found in journals and conference proceedings.

The only thing that cannot be bought or downloaded is the application-level software. The claim of this paper is that a component-based approach offers a solution to this obstacle, by making software components available for download or purchase alongside everything else.

B. Why Mobile Robotics is Particularly Suited to a CBSE Approach

In addition to this need, we would argue that certain domain characteristics make mobile robotics particularly suited to a CBSE approach:

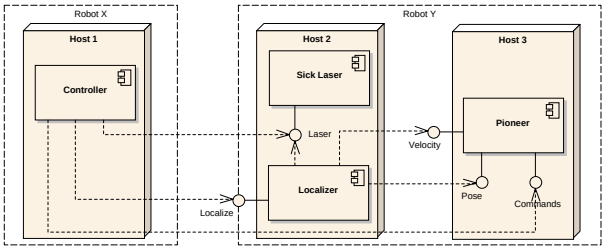


Fig. 1. A multi-robot Orca system. The software for Robot Y consists of three components, distributed over two hosts.

1) Inherent Complexity: Even a fairly simple robotic system, a single vehicle working in isolation, is complex. Correct operation requires the interaction of a number of sensors, actuators and algorithms. Many of these require their own threads of execution, and they are likely to need to communicate both synchronously and asynchronously. A mechanism is needed to manage this complexity from a software engineering standpoint.

2) A Requirement for Flexibility: Researchers need systems that are flexible enough to allow them to experiment with one particular aspect or algorithm, without their experimentation having large repercussions for the rest of the system. Researchers would also like to be able to compare different algorithms for doing a particular task while keeping the rest of the system constant. A prerequisite for this is software with strictly controlled dependencies.

3) Distributed Environments: Robotic systems are inherently distributed. When dealing with single vehicles it's often convenient to develop on workstations, controlling robots remotely. Often the processing power on a single robot is insufficient, requiring the use of off-board processing power. Code should not have to change if the off-board components are moved on-board. Increasingly, complex single vehicles are becoming multi-processor [15]. Multi-robot systems and sensor networks have many more distribution issues. Location transparency allows components to be distributed readily and easily, according to processing and bandwidth constraints.

4) Heterogeneity of Hardware and Operating Systems: Distributed robotic systems often require a mix of hardware platforms and operating systems.

The component-based approach advocated in this paper provides standard solutions to the challenges listed above, leaving researchers free to focus on their areas of interest.

V. Orca: A CBSE Framework For Mobile Robotics

Orca is an open-source CBSE framework designed for mobile robotics. An Orca component is a stand-alone process. It interacts with other components over a set of well-defined interfaces. Simply put, the Orca framework provides the means for defining and implementing these interfaces. A system consists of a set of interacting components, as illustrated in Figure 1. More formally, its aims are two-fold:

1) Provide a component model for the domain of mobile robotics. A component model "defines specific interaction and composition standards" [7]. It makes it easy to connect components (regardless of platform or location) and makes it likely that components developed independently will inter-operate.
2) Provide a component repository: a place where component re-use can occur.

In achieving these aims, Orca fulfils the requirements listed in Section I. The remainder of this section describes Orca's component repository, defines important elements of the framework, then describes the interaction and composition standards for Orca's components.

A. A Mobile Robotics Component Repository

This repository plays the role of the component catalog referred to in Section III, which implementers can peruse for the components required to build their system. It also plays the role of what Schmidt refers to as a "re-use magnet" [13], where innovation can be focussed. The Orca component repository is available online at http://orca-robotics.sourceforge.net.

All components in the repository are released under the Gnu General Public License (GPL), however the libraries that implement the framework are released under the Gnu Lesser General Public License (LGPL). This makes it possible to link against the libraries from closed-source components, and thus build systems that consist of a mixture of open- and closed-source components. This approach allows industry to maintain a competitive advantage while using Orca, and allows for the buying and selling of closed-source components.

B. Elements of the Orca Framework

Orca identifies the following aspects of the framework as important, and takes care to keep each decoupled from the others:

1) Objects: The abstract definitions of the units of data that can be passed (by value) between components. Examples include Point, Pose and LaserScan.

2) Communication Patterns: The abstract policies for how objects are sent between components. As an example, 'ServerPush' refers to a uni-directional channel between a server and a set of clients which is used to send objects asynchronously from the server to all the clients, as shown in Figure 2. 'ClientPush' is identical, but objects are sent from clients to the server. In this context, the term 'client-server' implies only that the connection is many-to-one.

3) Transport Mechanism: The method used to physically transport the objects, for example CORBA or raw TCP/IP sockets. Each communication pattern must be implemented at most once for each language and transport mechanism. There is no requirement that all patterns be implemented across all transport mechanisms: an incomplete set simply means that not all components are available for all transport mechanisms.

4) Components: The implementations of algorithms and hardware interfaces. The components can be implemented with knowledge only of the objects and the communication patterns. The implication of this is that the transport mechanism can be changed at compile-time.

A single item from each of the elements above is simple to implement, but a sizeable subset is a large task. Orca provides a set of re-useable implementations that is not only broad but can be expected to inter-operate.

The number of possibilities for each of the elements above is virtually infinite. Orca's approach is to define and implement those which are most likely to be useful, while ensuring that the framework is flexible enough that any of the items above can be added to. Users need only use the parts they like, and are free to make additions where they identify deficiencies. Keeping the elements of the framework as orthogonal as possible allows extensions to be made with minimal impact.

C. Component Interaction Standards

A component encapsulates some useful code, and interacts over a set of interfaces. An interface is completely defined by the combination of a communication pattern and an object type. Components can communicate if and only if the interfaces match. A port is an entity that can be instantiated to implement a specific interface. Ports may have additional parameters, such as how incoming objects are handled. These parameters are irrelevant to the outside world. Component code deals directly with the ports, without having to consider the complexity of network details.

An example may help to make component interactions clear. The following pseudocode sets up a ServerPush_Supplier port, which implements the ServerPush interface. It waits for clients to connect and pushes Pose2D objects:

```
ServerPush_Supplier(ObjectType) myPort
  ...
Pose2D pose(x, y, theta)
myPort.putData(pose)
```

The framework takes care of common communication-related issues that have proven useful in robotics, such as buffering on the receiver side. Receiving ports may have a buffer size of zero (meaning components must register a callback), a buffer size of one (termed a Proxy) or a buffer size of n. The following pseudocode sets up a ServerPush_Consumer with a buffer size of one:

```
ServerPush_ConsumerProxy(ObjectType) myPort
Pose2D pose()
myPort.getData(pose)
```

D. Component Composition

Composition standards define how components are pieced together into a system. The connectivity of an Orca-based system is defined by a set of XML configuration files, one per component. A component's configuration file has a section for each of that component's interfaces, defining how the interface will make itself available or how

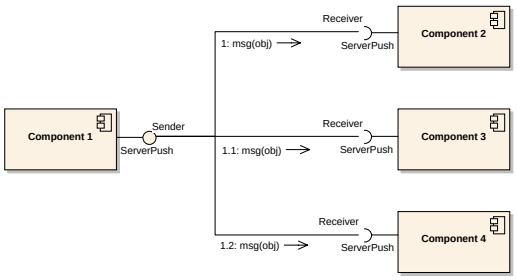


Fig. 2. The ServerPush pattern: a single server can push objects asynchronously to many clients. There must be one Supplier and n Consumers.

it will find another component's interfaces, as well as any necessary parameters for this connection. This information is likely to be specific to a particular transport mechanism. Each transport mechanism must implement a function to set up a connection based on the contents of this configuration file. Therefore switching transport mechanisms may mean adding to or changing the configuration file.

There is no particular constraint on how these connections should be specified. The configuration file can for example specify hard-wired connections, connections based on querying a centralised service, or perhaps dynamic connection establishment based on certain constraints.

These configuration files can also be used to customize the behaviour of components, fulfilling the requirement for the reuse of components that third parties be able to compose and customize them without ever having to look at the source code [3].

E. Design Considerations

A design choice that must be made concerns component granularity. Orca is not intended to be a panacea; engineering judgement is still required in this issue. Individuals are free to choose their own level of granularity. Using the principle of contractually-defined interfaces, as far as the rest of the system is concerned it is unimportant whether some given functionality is implemented in a set of smaller components or one monolithic multi-threaded component. All that matters is that the required interfaces are exposed.

In the robotics domain, another consideration is tightness of coupling. There is often a conflict between modularity, with the software engineering and re-use benefits that it brings, and the tight coupling required for efficiency and low latency, for example for tight control loops. Again, engineering judgement must be used.

F. Summary

Orca is the application of CBSE principles to the field of mobile robotics which fulfils all the requirements listed in Section I. In particular, Orca prescribes nothing about architectures. Extensibility is emphasized by keeping the various elements of the framework decoupled, making it easy to add to any one of them. Orca does not rely on any central entity and does not rely on any particular component, making distributed and decentralised systems possible.

Referring to the particular characteristics of mobile robotics outlined in Section IV:

1) Orca deals with complexity by allowing complex robotic systems to be reduced to a set of modular components.
2) Contractually defining the components' interfaces provides flexibility by strictly controlling dependencies and allowing for the re-configuration of systems and the replacement of components, even at run-time.
3) The particular challenges of distributed environments are addressed by providing component-developers with an array of off-the-shelf network-aware ports, objects, and components.
4) Heterogeneity of hardware and operating systems is addressed by providing abstract definitions of objects and communication patterns. By decoupling transport details from component code, the port of a transport mechanism to a different system can be applied across all components.

In addition to providing the means to produce components, Orca amplifies the usefulness of any particular component by providing a repository with which it can interact.

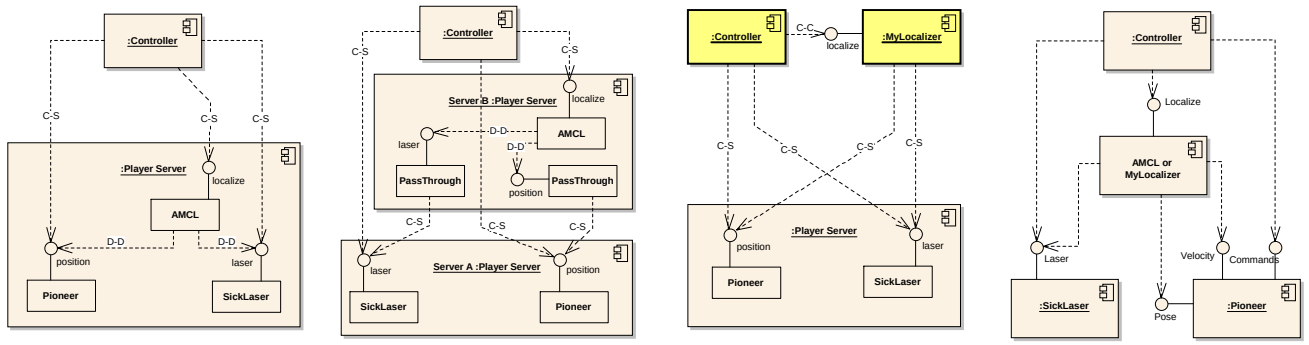
VI. Comparison With The Player Framework

Player [16] is the most successful mobile robotics re-use project to date. It was originally designed and developed as a hardware server. Its simplicity, reputation for reliability, good documentation and support from developers all contributed to its success. Player's design is similar to Netscape's plugin architecture, and is component-based in that sense: a monolithic server houses a set of modular devices. This has contributed to Player's successful re-use: devices conform to well-defined interfaces and correspond to well-understood algorithms or entities, allowing third parties to deploy and configure them without ever seeing the source code.

Player's most important constraint, which is fundamental to its design, is the delineation of server space from client space. This has several important consequences. Consider the single-robot system from Figure 1. Figure 3 shows how this might be built using either Orca or Player. Two differences are apparent. Firstly, Orca interfaces are more fine-grained than Player's, allowing finer control over configurations while adding some complexity. Secondly,and most importantly, there are several possible communication mechanisms in the Player model:

1) Inter-device communication, either within a Player server or between servers via a passthrough device
2) Client-Server communication
3) Inter-client communication

Inter-client communication is not addressed by Player at all, and must be invented by users. As such, software modules that rely on ad-hoc inter-client communication mechanisms do not conform to any particular standards, and are not re-useable in the same way as Player devices are. To make the 'MyLocalizer' and 'Controller'



(a) The setup using standard Player devices and a controller in client space.

(b) Moving AMCL to a different host requires extra PassThrough devices.

(c) Using one's own localizer in client space requires the use of client-client communication.

(d) The Orca method: only one type of communication is required.

Fig. 3. Various configurations for a system with hardware servers, a localizer (either Player's Adaptive Monte Carlo Localizer (AMCL) or a hand-written localizer) and a central controller, using either (a)-(c) Player or (d) Orca. The map (required for AMCL) is omitted for clarity. For Player, communication links are labelled Client-Server (C-S), Device-Device (D-D) or Client-Client (C-C). The use of non-standard Client-Client communication in (c) affects the re-useability of both the 'MyLocalizer' and 'Controller' implementations.

modules independently re-useable under the Player model, MyLocalizer must be ported to server space. This requires significant effort, a possible re-design to fit the Player Device model, and code changes to the Controller.

Rather than developing in client space then porting to server space, one might develop directly in server space. In practice this doesn't happen due to the extra complexity of conforming to the Player device model, and the problems of developing within a monolithic server where all components must be started and stopped together.

The Player model works well when its assumptions, namely that there are server-only modules and client-only modules, are correct. In contrast, all Orca components are equal. Components that act both as servers and clients are much more natural in this model. Since there is only one inter-component communication mechanism, the localizer in Figure 3 can be replaced or moved to another host easily, without requiring changes to other parts of the system.

The consequence of this design decision is that all Orca components are equally re-useable. The differences between Orca and Player become more apparent as system complexity scales from the simple vehicle in Figure 3 to complex distributed systems.

VII. Conclusion

This paper has provided an outline of Component-Based Software Engineering techniques, and motivated their application to the field of mobile robotics. It has described Orca: a specific approach to the adoption of component-based techniques, and a component repository.

Work is ongoing to improve the framework and develop new components. There are currently several indoor and outdoor projects underway that implement robotic systems using Orca components.

References

[1] D. Blevins. Overview of the enterprise JavaBeans component model. In *Component-based software engineering : putting the pieces together*, chapter 33. Addison-Wesley, 2001.

[2] Castek. Component-based development: The concepts, technology and methodology. Castek Company's white paper, available at www.castek.com, 2000.

[3] M. Collins-Cope. Component based development and advanced OO design. White paper, Ratio Group Ltd., 2001.

[4] B. J. Cox. Planning the software industrial revolution. *IEEE Software*, 1990.

[5] D. D'Souza and A. Wills. *Objects, Components, and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[6] T. Ewald. Overview of COM+. In *Component-based software engineering : putting the pieces together*, chapter 32. Addison-Wesley, 2001.

[7] G. T. Heineman and W. T. Councill, editors. *Component-based software engineering : putting the pieces together*. Addison-Wesley, Boston, 2001.

[8] A. Mallet, S. Fleury, and H. Bruyninckx. A specification of generic robotics software components: Future evolutions of genom in the orocos context. In *International Conference on Intelligent Robotics and Systems*, Lausanne (Switzerland), oct 2002. IEEE.

[9] R.C. Martin. Granularity. *C++ Report*, 1996.

[10] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In *IEEE/RSJ Intl. Workshop on Intelligent Robots and Systems*, 2003.

[11] I.A.D. Nesnas, R. Volpe, T. Estlin, H. Das, R. Petras, and D. Mutz. Toward developing reusable software components for robotic applications. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, 2001.

[12] A. Oreback. Components in intelligent robotics. Technical report, Royal Institute of Technology, Stockholm, Sweden, 1999.

[13] D. Schmidt. Why software reuse has failed and how to make it work for you. C++ Report magazine, January 1999.

[14] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2002.

[15] C. Urmson, J. Anhalt, M. Clark, T. Galatali, J. P. Gonzalez, J. Gowdy, A. Gutierrez, S. Harbaugh, M. Johnson-Roberson, H. Kato, P. L Koon, K. Peterson, B. K. Smith, S. Spiker, E. Tryzelaar, and W. Whittaker. High speed navigation of unrehearsed terrain: Red team technology for grand challenge 2004. Technical Report TR-04-37, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, June 2004.

[16] R. Vaughan, B. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2003.

[17] N. Wang, D. Schmidt, and C. O'Ryan. Overview of the CORBA component model. In *Component-based software engineering : putting the pieces together*, chapter 31. Addison-Wesley, 2001.