Reusable Software Components for Robots using Fuzzy Abstractions

Robert Smith
r2.smith@qut.edu.au

Glenn Smith
gp.smith@qut.edu.au

Aster Wardhani
a.wardhani@qut.edu.au

Centre for IT Innovation, Faculty of IT
Queensland University of Technology

Abstract

Mobile robots today, while varying greatly in design, often have a large number of similarities in terms of their tasks and goals. Navigation, obstacle avoidance, and vision are all examples. In turn, robots of similar design, but with varying configurations, should be able to share the bulk of their controlling software. Any changes required should be minimal and ideally only to specify new hardware configurations. However, it is difficult to achieve such flexibility, mainly due to the enormous variety of robot hardware available and the huge number of possible configurations. Monolithic controllers that can handle such variety are impossible to build.

This paper will investigate these portability problems, as well as techniques to manage common abstractions for user-designed components. The challenge is in creating new methods for robot software to support a diverse variety of robots, while also being easily upgraded and extended. These methods can then provide new ways to support the operational and functional reuse of the same high-level components across a variety of robots.

1. Introduction

Robotic control software has come along way since early sense, plan and execute systems [17] in the early 80s. Faster and more robust reactive styles were introduced [6], but were unable to execute complex plans. Deliberative systems that combine reactive and sense-plan-execute approaches have been built with varying degrees of success ([8], [11], [4], [2] and [16]). Decision-making and execution of robot activities are complex tasks to manage.

Hardware limitations and diversity have resulted in very specialised and inflexible software. Autonomous robots were traditionally faced with limitations in size, battery power, CPU speed, and memory. Hence controlling software was written to run as efficiently as possible to maximise the power of such hardware. The software had to focus on extracting the most from the hardware. This required very specialised solutions. In the past typical approaches to robot software construction produce monolithic systems using ‘brute force’ methods. This approach has made extensions, upgrades, and software reuse difficult.

Today, embedded processors are far more powerful. Autonomous robots with embedded CPUs are able to compute and react faster then ever before. This increased capacity reduces the need for low-level specialisation of software in order to gain critical speed optimisations. The increased capacity allows satisfactory computational speed to be achieved even when using more generic software.

This work leverages on the ability to use more generic software to address the need for more flexible and reusable robot code in the face of diverse hardware configurations. This is achieved via the specification and implementation of a framework that supports abstractions of robot hardware. The framework is constructed using component-based software techniques and the use of fuzzy logic enables a flexible and versatile manipulation of robot abstractions.

2. Issues

The most notable challenge with robot abstraction is that of the sheer diversity of robot hardware design. Secondly, the problem of moving code between systems is difficult as the translation is at best tedious or at worst impossible, even when the robot hardware can support all required functions of the code.

2.1. Diversity

The diversity among robots is extraordinary - both in their design (eg. hardware, size, shape) and their configuration (eg. orientation, position, facing). The matrix of possibilities creates a diversity that is unmanageable using existing techniques. There is very little standardisation between robots from different vendors. Even robots of the same type can be easily altered and reconfigured - so there are no certainties in how their hardware is configured. The problem with this diversity is two-fold. Firstly, software support is required to interact with each piece of hardware. Usually a device specific driver is provided for this. Even though the device driver offers some level of abstraction from the hardware, there is no interface standardisation between drivers even when the driver is for the same type of device, such as a servo. Thus the use of drivers does not address the issue of diversity. Secondly, higher-level algorithms that combine different hardware elements to achieve an overall result still need to handle various and changing configurations. How can the algorithms be written and packaged to operate unchanged on different robots?

2.2. Software portability

Software development for robotic systems faces many difficulties. The diversity of hardware and performance constraints has made the production of satisfactory solutions difficult. Prior to the relatively recent improvements in processing capacity, software development could only afford to address the essential requirements. This has led to the use of software engineering practice that does not consider the need for software reuse. Even if software reuse were considered, the diversity of hardware and its configurations would have probably restricted reuse to a single robot. This situation does not provide the motivation to commit additional resources required to develop reusable software.

Software development for robots is relatively immature in both time and scale compared with software development for more established environments such as the desktop computer. The standardisation of desktop platforms and the sheer scale of development have expedited the improvement of the software development process. This has left the software development process used for robotics platforms lagging behind. More modern software processes use component-based techniques. It is proposed that the lack of use of such techniques hinders the portability of software for robots.

3. Proposed Solution

To address these problems the concept of a Virtual Robot Layer is introduced in this paper. The VRL has arisen to address these issues:

- the abstraction of hardware devices; and
- to manage the diverse variety of possible robot hardware and its configurations; and
- to act as a translator of a common instruction set to robots of different configurations.

The concept aims to provide a standard means of accessing a robots functionality, as well as providing a protocol for communicating between high-level software and low-level executions, allowing the high-level instructions to remain unchanged across different robots with different hardware or configurations. Note that the VRL concept is limited in that it can only extend its services to the group of robots that have been targeted by a particular VRL framework, such as mobile-wheeled robots for example. Other groups of robots (such as humanoid) can be subsequently targeted with VRLs designed for those groups.

4. Benefits

The VRL framework can provide the following benefits.

4.1. Portability

Methods for robot abstraction and creating independent robot software, will improve the system's portability. In operational terms, the components and abstraction framework will work on various robot run-time environments. In functional terms, the components will be able to control various robots using the same infrastructure and high level instructions.

4.2. Code Reuse

By making a set of software components that are completely portable, new robots could be quickly configured using existing software with minimal effort. These components can be shared between robots of similar types. They can even remain unchanged when the robot hardware is reconfigured (for instance a sensor is moved to a different position) as they will adapt with support from the architecture using the metadata available for the current configuration. More code reuse will then be possible and the configuration and deployment time of robots will be reduced.

5. Methods and Approaches

Here we describe the techniques used in the design.

5.1. Software Components

Technologies are emerging today that allow applications to be built from reusable components more than ever before. Component-Based Software Engineering (CBSE) has become recognised as a new sub-discipline of software engineering and should equally apply to robotics software.

The major goals of CBSE are the provision of support for the development of systems as assemblies of components, the development of components as reusable entities, and the maintenance and upgrading of systems by customising and replacing their components [13].

Components, quite broadly speaking, are units for composition. In terms of software, a precise definition by Szyperski is frequently used today:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. [18]

We are using software components as, by definition, they bring modularity and well-defined interfaces and explicit context dependencies to the design and implementation of the abstraction framework and the sub-systems. The use of component-based software improves software development by enabling a design-by-composition environment and promoting software reuse [3].

5.2. Abstraction Principles

The essence of hardware abstraction is to decouple the users of the hardware from the non-essential details of its use. The user need only know how to manipulate the single abstraction, which in turn can be applied to a variety of hardware.

In software terms, hardware abstraction broadly means a separation of software from device dependencies or the complexities of underlying hardware. For instance, it enables programs to focus on a task, such as communications, instead of on individual differences between communications devices.

A hardware abstraction layer (HAL), in computing systems, is a layer of programming that allows a computer operating system to interact with a hardware device at a general or abstract level rather than at a detailed hardware level. Windows is one of several operating systems that include a hardware abstraction layer. The hardware abstraction layer can be called from either the operating system's kernel or from a device driver. In either case, the calling program can interact with the device in a more general way than it would otherwise.

In robot systems, the same approach is sometimes used. HALs exists for a few current robotic architectures ([10], [12], and [7]). The controlling software makes calls on hardware in only abstract terms, and the HAL then converts these calls into concrete signals to the hardware. To illustrate, a servo command such as `setspeed(byte speed)`, could be converted by the HAL to a series of ASCII characters to achieve that speed on the servo. When hardware is replaced or changed the HAL will also change the required signals, but the `setspeed(byte speed)` function call remains the same.

5.3. Degrees of Abstraction

Robotic software requires an even greater level of robot abstraction than provided by HALs. This is because robots interact with the real world. They are given commands that embody notions of position and direction, for example: `moveForward()`, and `turnRight()`. However, modern HALs have no information on relative placement of servos or sensors, making any abstraction involving location or direction impossible. This also precludes even higher levels of hardware abstraction to provide answers to questions such as:

- Is there an obstacle in that direction?
- How far away is the obstacle?
- Can I get through that gap in the wall?
- Where are my sensors pointing?

Most robot HALs only achieve the simpler level of hardware abstraction: the basic interface to the actual hardware. Higher-level algorithms still require built-in knowledge in to co-ordinate the hardware. It is far more useful (but complicated) to also abstract the configuration of hardware.

Most high-level algorithms use explicit knowledge of the hardware, such as where sensors are positioned and what the return values mean. This knowledge is usually encoded at a level higher than the HAL, which prohibits portability. For the higher-level algorithm to be portable, any such knowledge must be provided as a service from the HAL.

A HAL is useful because it allows code to be more portable. The more portable the code, the more diverse are the platforms on which it can be deployed. A HAL that supports even higher-level abstractions can remove platform dependence even further. This means that controlling software can run on different types of robots. How different the robots can be will depend on the sophistication of the HAL.

Even more sophisticated HALs in current robot architectures still only support robots from the same vendor, such as the ERP1 from Evolution Robotics [10]. Evolution Robotics has a proprietary robot architecture called ERSP [10]. This incorporates a HAL that uses XML specifications to support changes in the physical structure of their ER1 robot [9]. Hardware such as extra cameras and sensors can also be added. However, it is unable to support the abstraction of a completely different robot, such as the Khepera [15] for example.

The OROCOS project [7] also uses a hardware abstraction layer. They describe a layer between hardware and the framework, which translates calls from the framework to the present operation system and hardware drivers. OROCOS uses a suite of device drivers, which provide the hardware functions. There is no configuration specification or means for understanding higher level instructions or semantics such as left, right or forward.

The Player/Stage [12] abstracts hardware using interfaces that use TCP/IP socket communication from controlling software to the robot client. As for OROCOS, the level of abstraction is at a low level with basic interface on device driver support for the hardware.

In computing terms, the extension of the HAL towards higher-level abstractions can be served by a ‘virtual machine’. The equivalent concept in robotics would be a ‘virtual robot’. The notion of a virtual robot as a translator for abstraction is new. Following is a broad description of this with respect to the traditional concept of a virtual machine.

5.4. The Virtual Robot

A virtual machine is a hypothetical computer, whose characteristics are defined by its machine language, or instruction set. In general, a real machine with the same characteristics could be constructed with hardware. In its popular use today, the virtual machine is a software emulation of a physical computing environment used to execute instructions on the real machine. The virtual machine is a level of abstraction even greater than a typical hardware abstraction layer. It defines a set of rules for what it can execute, and can provide feedback and results.

A natural extension of a virtual machine is a virtual robot. The virtual robot would be an abstraction of a notional robot. It would be configurable so it can change its virtual shape, size and accessories. It would define its virtual capabilities and return information from virtual sensors. It could reply to queries about its specification. Most importantly, it would also translate and execute instructions that make sense on a real robot. See section 6.1 for the elaboration on the idea of a virtual robot.

5.5. Fuzzy Mechanisms

The forms of the abstractions used by the VRL are fuzzy. In that, crisp (non-fuzzy) values acquired by the VRL from sensors, are fuzzified according to the specifications and membership profiles, and these are provided to the higher level components that make use of the values according to fuzzy rules that guide its navigation and other functions. Instructions to the VRL are usually in a fuzzy format and these are defuzzified for the hardware to use. The abstractions are described more fully in section 6.3.

6. System Design

The hardware abstraction problem has been partially addressed in some architectures ([7], [10], [14] and [19]), where hardware abstraction layers (HAL) have been designed to allow basic control functions to be ported to different robots. These control functions allow low-level abstractions such as setting a servo speed, or requesting a sensor value.

However, a high-level command to move forward, turn left, or scan an area, can have radically different implementations on different robots. A mechanism to provide consistent interpretation of these high-level commands on various robots would be useful. These high-level commands need to be defined in a fuzzy format to allow algorithms to manipulate them at their highest abstract level. Then the high-level functional components can be completely portable because they only deal with the fuzzy handlers. The lower level implications of the fuzzy manipulations are left to the abstraction mechanism (the VRL).

6.1. Virtual Robot Layer

The VRL is the layer that provides the interfaces of a ‘virtual robot’ for the high-level functional components to use - concealing how these commands are translated into hardware calls. The VRL will also reply to high-level queries about the robot’s configuration. This is useful when initialising a component.

For example, a high-level `Navigation` component may plan a path based on sensory input. Then a low-level `Motion` component may manage the servos of the robot to achieve certain types of locomotion - for instance ‘turn left’, and ‘set speed’. The `Navigation` component should not need to change between robots, yet the `Motion` component almost certainly would. The robotic framework’s VRL will support this portability by:

- Translating high-level commands into a standard low-level protocol;
- Providing a specification of the configuration of the robot for any component to use; and
- Converting robot specific output or results into predefined standards used throughout the architecture.

To illustrate further, the VRL can convert infrared readings from the raw voltage value to a distance in centimetres. It specifies where the infrared sensors are and pointing. It also defines the size and extremities of the robot. It has information on the servo inputs required to achieve a particular speed profile. The VRL mediates communications between high-level and low-level components and systems as seen in Figure 1.

6.2. Operational and Functional Portability

To achieve cross-platform portability, a component must be portable in two (2) ways:

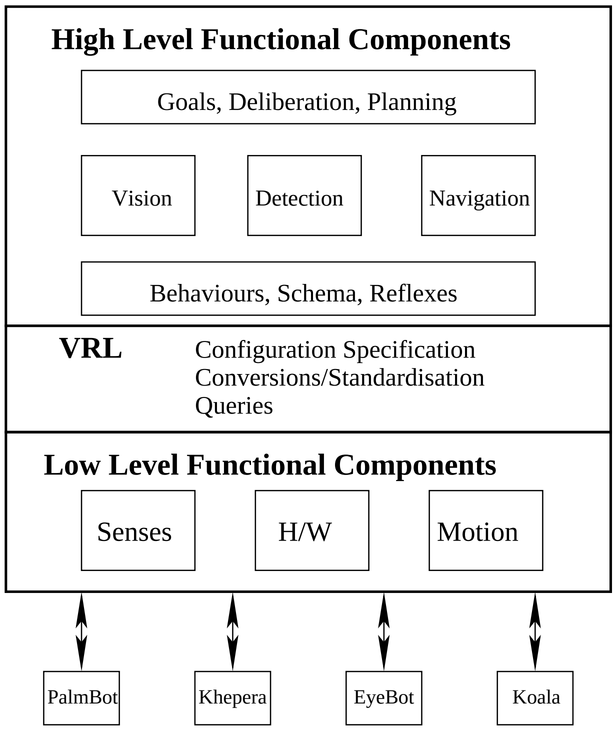
High Level Functional Components
Goals, Deliberation, Planning
Vision
Detection
Navigation
Behaviours, Schema, Reflexes
VRL
Configuration Specification
Conversions/Standardisation
Queries
Low Level Functional Components
Senses
H/W
Motion
PalmBot
Khepera
EyeBot
Koala

Figure 1. The VRL connects the high-level and low-level systems and components.

- Operationally - in that the component can be initialised and run using the new operating system or run-time environment; and
- Functionally - so that the control structures and algorithms that work in one robot (or configuration) continue to work on another.

In our prototype implementation, the robotic framework and the components are written in Java. This allows them to operate on any platform for which a suitable JVM is available. This provides the *operational* portability of the robot component. JVMs are run on any Windows or Linux OS based robot and can also be run on the Palm OS and there are even ports for the Motorola 68k series.

Furthermore, the *functional* portability of a component is supported by the VRL and operating infrastructure. Thus components can operate on any robot that has a VRL with a configuration that covers all required functionality.

It is important to note however, that not all components will work on all robots (even amongst those in the target group). For instance, before a particular component can initialised and run, sensors may be required in a certain direction. This preconditioning of operation precludes the component running on some configurations. Thus portability of code depends on some overlap in robot capability.

6.3. Fuzzy Abstractions

A simple illustration of the operation of the VRL will help identify the types of abstractions used. Take the pseudo-code for a Braitenberg styled obstacle avoidance algorithm as is shown in Figure 2.

```
while (true) {
        if (IRSensors(forward, blocked))
                Rotate(speed);
        else MoveForward(speed).
}
```

Figure 2. Simple pseudo-code for obstacle avoidance.

Even this simple algorithm requires the robotic framework to know the answers to questions such as:

- Which way is ‘forward’?
- What does ‘speed’ mean in the contexts of moving forward and rotation?
- How do I ‘rotate’?
- Where are the infrared sensors pointing?
- What infrared sensor reading means ‘blocked’?

These questions are answered by the VRL in terms of fuzzy abstractions and membership values of fuzzy sets. The current sets are:

- Direction - Front, Left, Right or Back
- Movement - Forward, RotateLeft, RotateRight or Reverse
- Speed - VeryFast, Fast, Medium, Slow, VerySlow or Stopped
- Distance - VeryFar, Far, Near or VeryNear
- Size - VerySmall, Small, Large or VeryLarge

These fuzzy constructs can then be used to describe hardware placement and orientation, the direction of obstacles or targets, speeds of travel and distances to the robot. All fuzzy terms are relative. By this we mean robot-centric. So an obstacle that is `Near` to a `Large` robot may only be `Far` to a `Small` robot. The VRL of course can be configured accordingly to make the correct interpretations.

The VRL is also configured with the appropriate hardware placements of sensors and cameras etc. Sensors can be grouped together to form zones of measuring (each group having its own membership function as to its direction from the robot). These details can be made available to the components on initialisation to check prerequisites as well as during run-time.

The VRL uses predefined standards within the system and these are built into the interface contracts. An example of this would be the `Speed` measurement, which is always available in cm/s before any fuzzification. Both the crisp and fuzzy value is available on demand. Other questions are translated by the VRL and queried or executed in the lower level components themselves. For instance, the motion component manages the rotation and movement of the robot. A VRL and associated low-level functional components are implemented uniquely for each robot. Again note, these particular components are not transferable between platforms as they are robot specific.

6.4. Configurability

The VRL provides data on the robot configuration including servo and sensor positioning and their signal input and output meanings. This function is loaded into the VRL from a configuration file, which is written once for each robot type and altered according to hardware changes. An configuration file using XML is very flexible, as used by [10] and [5]. We use XML configurations to completely specify our robots. This way the VRL can be easily modified with changes in the robot hardware or for entirely new robots.

7. Outcomes

The VRL concept has so far been implemented on three indoor mobile robots in our laboratory. These robots varied a great deal in size, performance and shape. From the small sized Khepera [15], to the larger Koala [15] with six wheels and many more sensors, to the Palmbot [1] with far fewer sensors and uses holonomic motion.

An obstacle avoidance component using infrared proximity measurements can operate completely unchanged on each of the three robots. The VRL would take the high level commands from the component and translate them into the appropriate robot level instructions. In future we hope to apply the VRL concept to humanoid and flying robots as well.

References

[1] Acroname. Palm Pilot Robot Kit. *www.acroname.com*, May 2002.

[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17(4):315–337, April 1998.

[3] J. E. Beck, J. M. Reagin, T. E. Sweeny, R. L. Anderson, and T. D. Garner. Applying a component-based software architecture to robotic workcell applications. *IEEE Transactions on Robotics and Automation*, 16(3), June 2000.

[4] R. P. Bonasso. Experiences with an architecture for intelligent reactive agents. *Journal of Experimental and Theoretical AI*, 9(2), 1997.

[5] A. Bredenfeld. Behavior engineering with “dual dynamics” models and design tools. In *Sixteenth International Joint Conference on Artificial Intelligence IJCAI-99 Workshop ABS-4*, pages 57–62, Veloso, Manuela, 1999.

[6] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotiecs and Automation*, RA-2(1):14–23, March 1986.

[7] H. Bruyninckx. Open Robot Control Software: the OROCOS project. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, Seoul, Korea, May 21-26 2001. IEEE.

[8] C. Elsaesser and M. G. Slack. Integrating deliberative planning in a robot architecture. In *Proceedings of the AIAA/-NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS ’94)*, pages 782–787, Houston, TX, 1994.

[9] Evolution Robotics. ER1 Robot. *www.evolution.com*, October 2002.

[10] Evolution Robotics. Robotic Architecture. *Technical White Paper*, www.evolution.com/product/oem/, January 2003.

[11] R. J. Firby. Programming CHIP for the IJCAI-95 robot competition. *AI Magazine*, Spring 1996.

[12] B. P. Gerkey, R. T. Vaughan, K. Sty, A. Howard, G. S. Sukhatme, , and M. J. Mataric. Most valuable player: A robot device server for distributed control. In IROS, editor, *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1226–1231, Wailea, Hawaii, November 2001.

[13] G. Heineman and W. Councill. *Component-based Software Engineering, Putting the Pieces Together*. Addison Wesley, 2001.

[14] iRobot. Mobility Package. *www.irobot.com*, March 2003.

[15] K-Team. Mobile Robotics. *www.k-team.com*, July 2002.

[16] M. Lindstrom, A. Oreback, and H. Christensen. BERRA: A research architecture for service robots. In *International Conference on Robotics and Automation*. IEEE, 2000.

[17] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Co., Palo Alto, CA, 1980.

[18] C. Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley Press, New York, 1998.

[19] R. Volpe, I. A. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *Proceedings of the 2001 IEEE Aerospace Conference*, Big Sky, Montana, March 2001.