Software reuse across robotic platforms: Limiting the effects of diversity

Glenn Smith, Robert Smith and Aster Wardhani
Centre for IT Innovation, Faculty of Information Technology
Queensland University of Technology, Australia
Email gp.smith@qut.edu.au

Abstract

Robots have diverse capabilities and complex interactions with their environment. Software development for robotic platforms is time consuming due to the complex nature of the tasks to be performed. Such an environment demands sound software engineering practices to produce high quality software. However software engineering in the robotics domain fails to facilitate any significant level of software reuse or portability.

This paper identifies the major issues limiting software reuse in the robotics domain. Lack of standardisation, diversity of robotic platforms, and the subtle effects of environmental interaction all contribute to this problem. It is then shown that software components, fuzzy logic, and related techniques can be used together to provide suitable abstractions to address this problem. While complete software reuse is not possible, it is demonstrated that significant levels of software reuse can be obtained.

Without an acceptable level of reuse or portability, software engineering in the robotics domain will not be able to meet the demands of a rapidly developing field. The work presented in this paper demonstrates a method for supporting software reuse across robotic platforms and hence facilitating improved software engineering practices.

1. Introduction

Software engineering in the robotics domain faces unique challenges. Robots have diverse capabilities and engage in complex interactions with their environment. The tasks that are typically performed by robots are complex due to the non-determinism inherent in the environment in which they operate. This complexity makes software development for robotic platforms time consuming. Such an environment demands sound software engineering practices to produce high quality software.

Software engineering in the robotics domain fails to facilitate any significant level of software reuse or portability. Historically robot processing capabilities and memory capacities have been extremely limited. Software engineering in this constrained environment typically "hardwired" code to the hardware to make maximum use of available resources. Consequently the resulting software was very specialised and dependent on a particular robot. These resource constraints have been lifted to a great extent with recent improvements in the area. However, robotics software engineering has failed to improve in line with hardware improvements.

In this paper we will discuss the aspects of the robotics domain that inhibit effective software engineering. There is a diverse range of hardware used to perform typical robot tasks and a lack of standardisation of the hardware interfaces used. This diversity accounts for part of the challenges to software engineering. However, it is the consequences of environmental interaction that poses the most significant challenges. As robots interact with their environment they must be aware of their physical presence and configuration in that environment. Attributes such as robot size and sensor orientation can create dependencies in the controlling software to a particular robot. These dependencies restrict software reuse and portability. A change in orientation of a single sensor could cause the controlling software to fail.

The manifestation of the challenges outlined is the dependency between a robot's controlling software and its hardware configuration. The solution presented here uses a Virtual Robot Framework (VRF) that essentially provides an abstraction layer to limit the effects of diversity and lack of standardisation of robot hardware. Fuzzy logic is used to enable the construction of the VRF. Finally the extraction of the specification of physical attributes from the robot software combine to limit the effects of environmental interaction.

The following sections of this paper present in more detail the challenges for software engineering in the robotics domain, and describe our approach to addressing these challenges. The details of the implementation are provided in Section 4. Related approaches in the area are presented in

Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05)
1530-0803/05 $20.00 © 2005 **IEEE**

Section 5. Finally insight into the experience this work has provided and general discussion of the applicability of the approach is given.

2. Challenges of robotics software engineering

Historical hardware limitations in robotics no longer appear to be the limiting factor in robotics software engineering. This change has not led to the adoption of more modern software engineering techniques utilised in general software engineering. Software reuse is virtually non-existent and significant effort is required to move software to related robot platforms. The use of modular software components in the robotics domain is only starting to develop. It is proposed that significant obstacles to more effective software engineering remain and contribute to the current state of robotics software engineering. There are three major obstacles identified. These are detailed in the following subsections.

2.1. Diversity of robotic platforms

There is a wide variety of robot hardware available giving near infinite number of combinations of this hardware. To simplify our discussion consider the broad dichotomy of this hardware: effectors and sensors. Effectors change the state of the robot or the environment. Sensors collect data from the robot and sample the state of the environment.

Sensors include items such as infrared sensors, sonar, video cameras, thermostats, GPS locators, and pressure sensors. For each of these kinds of sensors there are typically many variations, for example, a video camera could be colour, greyscale, monocular, binocular, fixed or moving. Variations in signal types, and data rates require very particular coupling of software to these devices. There is an equally complex categorisation for effectors including items such as wheels, arms, lights, speakers, and probes.

The software controlling the effectors and sensors needs to correctly interpret input and generate meaningful output across a wide variety of data and signal types. This variation of effectors and sensors may be compared with the variation of input and output devices for desktop computers. However there are two major distinctions. First, effectors and sensors must interact with the real world environment. Second, the physical orientation (location and direction) of effectors and sensors has a profound effect of the interpretation of input and the generation of output. The consequences of these distinctions are discussed below in the context of environmental interaction.

At this point is should be clear that robotics software engineering is at least as complex as general software engineering. This is without considering the effects of environmental interaction or the typical complexity of the tasks performed. Furthermore, there is the added variation introduced by the wide variety of robotic operating systems.

2.2. Environmental interaction

From the software engineering perspective, there are two major causes of environmental interaction difficulties for robots: 1) the effector and sensor location and orientations are significant, and 2) the variation of sensor input is large. The consequences of these will be described and discussed in turn.

Robots can vary significantly in their combination of types of effectors and sensors (hardware configuration). However, the impact on software engineering is felt when considering the possible location and orientation of these effectors and sensors. Even when the hardware configuration is the same, the controlling software may no longer work if the location or orientation of a single sensor is changed. For example, if an infrared sensor located on the front of the robot is turned to face the side, the controlling software must change its interpretation of the input from that sensor. Even such a simple change is likely to require modification of the controlling software. Thus a change in sensor (or effector) location or orientation essentially changes the configuration of the robot. Thus software may not even be reusable between robots with the same hardware configuration. Consider the problems that would occur if software for personal computers needed to be changed because of the direction the monitor was facing.

The input to the robot generated by sensors typically requires more interpretation than input generated by general computer input devices. Sensor readings may vary because of completely independent environmental factors such as the level of lighting. Sensor input received by the controlling software is typically continuous in value. These values may vary with no distinguishable change to the environment. For example it is not uncommon to receive successive readings such as 0.849 and 0.0902, and the difference may or may not be significant. Input to a desktop computer tends to be more constrained and more stable, for example a keyboard produces discrete values within a limited set of possible values. While sensor input alone does not directly cause problems with software reuse, it does add to the complexity of the software created.

The interpretation of sensor values also varies with the physical attributes of the robot with respect to the environment that it is operating within. For example, if a sensor indicates that an object is 2 meters away, the interpretation of this value will depend on the size of the robot. For a large robot, the size of a car, this distance may be considered close. For a small robot, the size of a mouse, this distance may be considered far. There are many other physical attributes

that have similar relationships. The dependencies on physical attributes are likely to create further dependencies between the controlling software and robot hardware.

2.3. Lack of standardisation

Some of the diversity of robot composition could be constrained by standardisation of software interfaces for hardware components. However, there is a distinct lack of interface standardisation, possibly as the robotics domain is still a relatively small market. Currently the number of robots in existence are some orders of magnitude less than the number of desktop computers. This smaller market will not generate those same forces that have led to standardisation of interfaces for desktop computer hardware. The combination of the small market and the diverse range of hardware reduce the impetus for standardisation as it is difficult for a single vendor to dominate and the advantage of doing so is limited.

The problems of platform diversity and environmental interaction are exacerbated by the lack of standardisation. Given these challenges, it is not surprising that the majority of robotics software development produces specialised code that is only capable of working on a single robot configuration. There is little opportunity for software reuse even though it is desperately needed in such a complex software engineering domain.

3. Approach

To limit the effects of diverse robotic platforms and environmental interaction, we propose the use of a component system architecture combined with various levels of data abstraction. The system is composed of several component frameworks, each supporting a typical facet of robot control. These first level component frameworks plug into a single second level component framework (or component system) that mediates their communication. This should not be confused with a traditional layered architecture, it is a tiered component architecture [17]. The general structure is, components plug into the first level component frameworks, and these frameworks plug into the component system.

The individual components are where the majority of software development would occur, and where the specific behaviours of the robot would be encoded. It is in these components that we aim to provide the highest level of reuse across different robots.

The first level component frameworks each address a different facet of robot control. In the system we have developed there are five component frameworks for high-level control and these are categorised as Behaviours, Deliberation, Detection, Navigation, and Vision. These are illustrated in Figure 1. These communicate through a Data Exchange (DX) framework. There is also another component framework called the Virtual Robot Framework that provides a "virtual robot" interface to the rest of the system. It is supported by a robot (XML) configuration file and a descriptor file of the fuzzy abstractions it should use. The VRF is a specialist framework and will be described in some detail in later sections. This set of component frameworks is not intended to be final and the architecture allows for new frameworks to be added and implementations of the existing component frameworks to be interchanged. We aim to provide a high degree of reuse of the control frameworks across different robots. The VRF is a special case in that it is tightly bound to the robot configuration and as such is not portable across different robots. The VRF in effect decouples the rest of the system from the robot hardware.

The component system supports the communication of the component frameworks. It is the most stable entity and should rarely change, except when support for a new type of component framework is required.

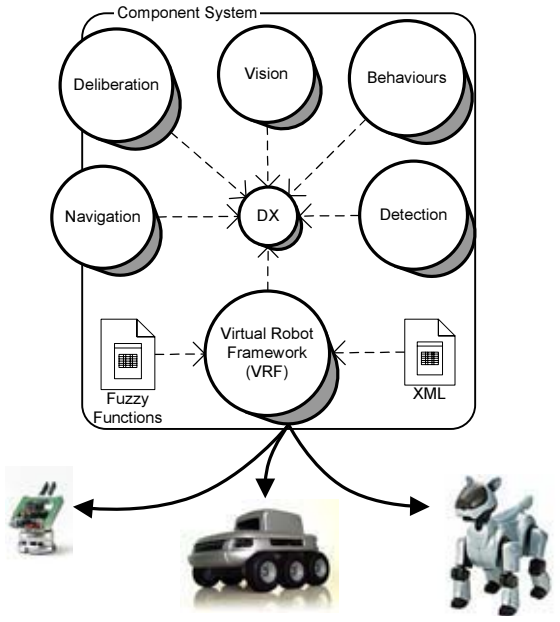

Figure 1. The VRF decouples the high-level frameworks and components from the hardware. The high-level frameworks need to use the standard interfaces and fuzzy abstractions provided by the VRF to be portable.

The VRF is the key to software reuse in this system. To understand the function of the VRF, consider the function of the Java Virtual Machine (JVM) [10]. The JVM provides for platform independent execution of Java Bytecode. That is, Java Bytecode can execute on any platform that has a JVM

available. This is similar to the function of the VRF, any code that uses the VRF interface can be moved between robots that implement the VRF interface. However, this comparison is not entirely accurate as VRFs are necessarily configured based on the capabilities of the robot. So there is a limitation: code can only be used on VRF configurations that support all the required capabilities. Note that the capabilities of the VRF map directly to the capabilities of the robot; so it is the case that if the VRF cannot support the code, then the robot hardware would not support the code either. The benefit for reuse lies in the space where robot capabilities overlap.

This gives an understanding of the general architecture used, we will now consider how the challenges presented in Section 2 have been addressed.

3.1. Addressing diversity of robotic platforms

The effects of diverse robot hardware is limited by the VRF. The VRF provides a set of standard interfaces for interacting with low-level hardware. Implementing these interfaces is hardware dependent, although the interfaces only need to be implemented once for a given type of hardware on a particular robot. An example of an interface to interact with the sensors is in Figure 2. Once implemented, the VRF provides a stable "virtual robot" platform for other component frameworks to use.

The implementation of the interface does not need to be linked to any particular underlying hardware. For example, take the `isBlocked` method which indicates if the path in a given direction is clear of obstacles or not. The implementation of this method may utilise a camera or a set of infrared sensors to determine the result. Thus the method can be implemented on varying hardware. Abstracting the required functionality from the underlying hardware allows the component frameworks using the VRF to be independent of the actual hardware configuration. This adaptation across different kinds of hardware can be applied in many different cases. Further independence is gained by abstracting the values used, this is discussed in the next sub-section.

3.2. Addressing effects of environmental interaction

There are two main aspects to environmental interaction, the first discussed here is the dependency on effector and sensor location. When considering this aspect the physical attributes of the robot relative its environment must also be taken into consideration. If the controlling software required implicit knowledge of the size, shape, and sensor locations for a robot it would be bound directly to that robot and reusing the code on another robot would almost certainly require change.

```
/*********************************************
 * @return        errNone if successful
 * Pre: ID != null
 * Inv: ID != null
 * Pst: rawValue[n] == *reading* &&
 *        timeStamp == *time of reading*
 *                  for all n
 * Dsc: Polls all the sensors and stores the
 *        reading in rawValue[]
 */
       public int pollAll();
       public int poll(int ID);

/*********************************************
 * @return        timeStamp
 * Pre: True
 * Inv: True
 * Pst: True
 * Dsc: Returns the timing stamp of the latest
 *         polled data
 */
       public double getTimeStamp(double ID);

/*********************************************
 * @return        rawValue
 * Pre: True
 * Inv: True
 * Pst: True
 * Dsc: Returns the rawValue property
 */
       public double getRawValue(double ID);

/*********************************************
 * @return        Distance from rawValue
 * Pre: timeStamp != 0
 * Inv: timeStamp != 0
 * Pst: timeStamp != 0
 * Dsc: Calculates the distance equivalent to the
 *        rawValue reading
 */
       public double calcDistance(double ID)
              throws Exception;
```

Figure 2. An extract of the `SensorGroup` interface.

The VRF utilises specifications of the robot configuration including sensor and effector positioning as well as the physical attributes of the robot. This information is loaded into the VRF from a configuration file, which is written once for each robot type and altered according to hardware changes. A configuration file using XML is very flexible, as used by [8] and [4]. The XML configuration file used to specify the robots (see Figure 3) is based on work done by [4]. This way the VRF can be easily modified for new robots, or along with changes in an existing robot hardware configuration.

The XML configuration file also indicates the placement of hardware items such as sensors and cameras (see Fig-

```
<Robot RobotType="Khepera">
      <Dimension Height="70"/>
      <Polygon
                NPoints="6"
                XPoints="-35,-35,-35,35,35,35"
                YPoints="-40,0,40,-40,0,40"/>
      <CenterOfRotation
                     XCenterOfRotation="0"
                     YCenterOfRotation="-10"/>
      <Drive
            DistanceLeftRightWheel="40"
            WheelDiameter="20"
            MaxSpeed="200"/>
...
</Robot>
```

Figure 3. An extract of the XML configuration file defining the physical structure of a robot.

```
      <Sensor>
            <Label>I1</Label>
            <Infrared Typ="INFRARED">
                <SensorGroup>1</SensorGroup>
                <ScanRange>55</ScanRange>
                <ZeroDistance>0</ZeroDistance>
            </Infrared>
          <Position
                     XPosition="17"
                     YPosition="15"
                     ZPosition="8"/>
          <Rotation XRotation="45"/>
      </Sensor>
```

Figure 4. An extract from the XML configuration file showing the definition of an infrared sensor.

ure 4). Sensors can be grouped together to form zones. The concept of sensor zones is used in abstraction work done in the Pyro programming framework [2]. These are used by Pyro to help manage varying robot morphologies. These zones, for example, could be Front, Back, Left, and Right. This allows measurements to be consolidated for these zones and returned in a consistent manner independent of the actual hardware configuration supporting that zone. The group of sensors included in a zone can overlap, for example, if a sensor is on the front-right aspect of a robot it could be included in the Front and Right zone. Equally if the front-right sensor was then rotated to face the front only, then it would be removed from the sensor group corresponding to the Right zone.

These sensor groups provide additional abstraction from the configuration of individual hardware items, allowing the VRF to present a standard representation of varying collections of sensors. This further reduces the direct dependencies between controlling software and hardware configuration. If the controlling software is written to use zones, then provided the zones can be constructed from the sensor groups on a given robot, the software can be reused on that robot.

The configuration information can also be used to assist in the initialisation of software components, and allow the components to ensure all the services that they require are in fact available. This is important to support a culture of plug-and-play robotic software components as it will provide some assurances that if a new component initialises, it can in fact operate correctly rather than leading to run-time failure. There are still many cases where run-time failure can occur, but these are yet to be solved for general component based software engineering.

The second aspect of environmental interaction is the interaction with the environment through effectors and sensors. The values used are typically continuous and have some degree of variation resulting from the number of environmental factors that can influence the sensor readings, such as lighting, and vary the effect of actions taken by effectors, such as moving on a slippery surface. This effects of these variations needs to be limited to help reduce the complexity of the controlling software and to support a simplified virtual robot interface. We have applied fuzzy logic to this problem.

In general, a fuzzy system [7] has three parts. The first is a fuzzification section. This section is responsible for taking real input data (also referred to as crisp data), and converting it to data that has meaning to the fuzzy system. Next, the fuzzified data is applied to the fuzzy-rule base section. The rule base is applied and result calculated within the high-level components. The result of the rule-base section is fuzzified output data. This fuzzified output data is converted back to real or crisp data using a defuzzification process. The VRF completes all of these tasks.

The use of data fuzzification allows the use of imprecise data, and enables grouping of values that have similar meaning but distinct concrete representations. Consider the simple example, introduced in Section 3.1, of determining if the path in a particular direction is clear or blocked implemented by an `isBlocked` method presented by the VRF. One robot may have a video camera facing forward and use optical flow to determine free space, another robot may have three infrared sensors pointing forward. Very different methods of interpretation are required of these sensors. However, these values can be translated to a fuzzy value with the direction and the range of the obstacle. The appropriate range can then be returned by the VRF satisfying components that use that VRF method. Obviously the implementation of the `isBlocked` method will be very different for the VRF for each robot.

However, most importantly, the fuzzy membership func-

tions can be easily decoupled from their crisp meanings. Decoupling here is the key so that the meaning of the linguistic terms (i.e. natural language names for the range of values) can be modified easily to meet the description relative to different robots. Examples of these are shown in Figure 5.

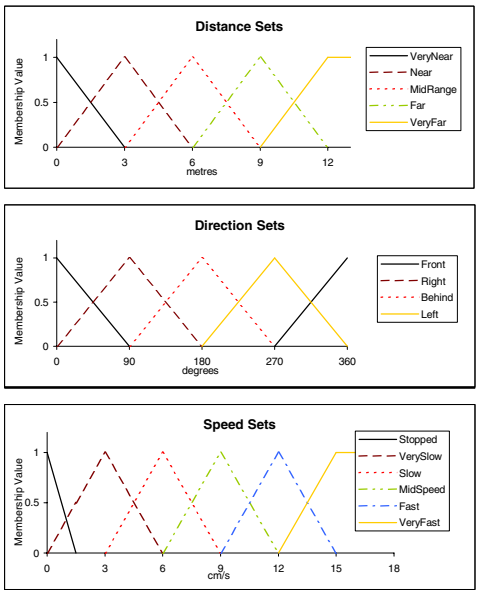

Figure 5. The input membership sets for some linguistic variables. These sets use example scales for illustration only. They will be different according to the robot they are trying to describe.

These fuzzy values 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 VRF is configured with the fuzzy set membership profiles accordingly to make the correct interpretations. Presently, the fuzzy membership functions have a predetermine number of sets and profile shape (e.g. triangular). A *negotiated* number of sets and profile shape is being considered so as to provide more flexibility for the components. This would be limited by the robot hardware and possible options it can provide.

The combination of these approaches are used to decouple the controlling software from the robot hardware and simplify the virtual robot interface. While they do not ensure standardisation, the benefits of software reuse from using such a standard robot platform will hopefully be sufficient motivation for both robot manufacturers and users alike. We will now discuss the implementation of these approaches to demonstrate their feasibility.

4. Implementation

Prototypes of the VRF have been implemented on multiple robot systems to test its support of portability. The following robots are quite varied in features and have been selected as they provide a good sample of mobile robots:

- Palmbot [1] - a triangle shaped, three wheeled, omnidirectional robot. Sensing is done using three infrared sensors and a camera.
- Khepera [11] - a small circular shaped robot approximately 4cm in diameter. It uses two-wheeled locomotion with eight infrared sensors, light sensors and a linear camera.
- Koala [11] - a six wheeled robot with multiple infrared sensors and a colour camera.
- AIBO [16] - from Sony, a four legged walking dog-inspired robot.
- Webots [12] - a 3D robot simulator from Cyberbotics simulating multiple robot platforms.

A VRF has been constructed for each of these robots (for the actual and simulated). The component frameworks in the system (shown in Figure 1) have been implemented, however these only required one robot independent implementation that depends on the specific VRF implementation. For example, an obstacle avoidance component has been developed that operates unchanged on each of the robots by interacting with the robot's VRF. The configuration of the sensors on each robot is quite varied so the VRF must interpret requests to read the sensors and return meaningful results to the component. The obstacle avoidance is guided by the VRF to realise meanings for each robot's size, sensors position & signal meanings, servo positions, and speed setting.

Using a component design allows software components implementing different avoidance algorithms to be easily deployed. These software components plug into the appropriate component framework implementation, where the component framework that interacts with the robot specific VRF. A simple illustration of the operation of the VRF will help show the usage of the abstractions. Take the pseudocode for a Braitenberg [3] styled obstacle avoidance algorithm as shown in Figure 6.

Implementation of this simple algorithm requires knowledge or interpretation of facts such as:

- Which direction is 'front'?
- What is 'midSpeed' value for this robot?

```
if (isBlocked(front, veryNear)) then
     if (isClear(right, veryFar))
           then rotate(slow, right);
     else rotate(slow, left);
else moveForward(midSpeed);
```

Figure 6. Pseudo-code for simple obstacle avoidance.

- How to 'move forward'?
- How to rotate?
- Where is the location of the sensors?
- What do the sensor readings mean?

To implement even this simple algorithm would typically lead to dependencies on the robot's hardware configuration. Using the VRF concepts such as 'front' and 'midSpeed' are specified when configuring the VRF and its fuzzy membership sets. Concepts such as 'move forward' and 'rotate' are available as standard method calls presented by interface provided by the VRF and implemented for each specific robot. The configuration of the sensors is provided by way of the XML configuration file and interpreted by the VRF to enable calculations sensitive to the current robot configuration. Interpretation of the sensor readings is handled within the VRF through a process of fuzzification.

An example of the robot specific implementation of the `isFrontClear` method provided in the VRF interface is given in Figures 7 and 8. This code shows how the same method is implemented for two very different robots. The input parameter to the `getRawValue` method is the identifier of the sensor, and the return value is compared to a value appropriate for that sensor type. A more sophisticated implementation may use a heuristic to merge the returned values, however the code shown in the diagram provides a simple example for illustration purposes.

```
public boolean isFrontClear() {
        pollAll();
        return (getRawValue(0)<300 &&
            getRawValue(1)<300 &&
            getRawValue(2)<300 &&
            getRawValue(8)<300 &&
            getRawValue(9)<300 &&
            getRawValue(10)<300);
}
```

Figure 7. An extract of the `isFrontClear()` method from the Koala robot VRL.

```
public boolean isFrontClear() {
        pollAll();
        return (getRawValue(1)<10 &&
                getRawValue(2)<10 &&
                getRawValue(3)<10 &&
                getRawValue(4)<10);
}
```

Figure 8. An extract of the `isFrontClear()` method from the Khepera robot VRL.

To implement methods in the VRF that make use of continuous values, fuzzy logic is used. The value sets are configured as shown in Figure 9. This makes use of predefined set types provided by the the NRC FuzzyJ Toolkit [15] which is a set of Java classes that provide the capability for handling fuzzy concepts and reasoning. This extract from some configuration code corresponds to the fuzzy sets shown in Figure 5. The decision of how to structure the fuzzy sets is quite complex, but the implementation with the support of the fuzzy library is relatively easy.

```
public FuzzyVRL() {
...
  speed = new FuzzyVariable("Speed",0.0,15.0,"cm/s");
  speed.addTerm("stopped", new ZFuzzySet(0.0, 1.5));
  speed.addTerm("verySlow",
        new TriangleFuzzySet(0.0, 3.0, 6.0));
  speed.addTerm("slow",
        new TriangleFuzzySet(3.0, 6.0, 9.0));
  speed.addTerm("midSpeed",
        new TriangleFuzzySet(6.0, 9.0, 12.0));
  speed.addTerm("fast",
        new TriangleFuzzySet(9.0, 12.0, 15.0));
  speed.addTerm("veryFast", new SFuzzySet(12.0, 15.0));
...
}
```

Figure 9. An extract of the fuzzy set formation code from a FuzzyVRL implementation.

The use of these fuzzy sets enables implementation of methods using values that are decoupled from the robot hardware. An example of the `moveForward` method is shown in Figure 10. This example demonstrates how a value is generated to move the robot forward. The speed of the robot is dependent on the construction of the fuzzy sets.

In some cases fuzzy values cannot be used as the sensor input and effector output must meet some precise standards. For these cases there are methods provided to use crisp values. However, this will still create dependencies on the robot hardware. These interfaces are only used when accuracy is more important than portability and should be used only

```
public int moveForward(String speedS) {
        double crispSpeed=0.0;
        try{
          FuzzyValue fuzzySpeed = new FuzzyValue(speed,
              speedS);
          // calculate the defuzzified value
          crispSpeed =
                Math.max(fuzzySpeed.maximumDefuzzify()
              , 1);
        } catch (FuzzyException fe) {
        }
        System.out.println("Moving at speed "
              + crispSpeed);
        return setSpeed((int)(crispSpeed));
}
```

Figure 10. An extract of the defuzzification of the `Speed` variable for use by the robot's `moveForward` method.

when absolutely necessary.

Software components (as described by [17]) are used because they provide a flexible solution of varying configurations while allowing a modular and independent implementation. Components are still tailor made to fit a solution but can be reused across multiple robot configurations. In most cases this can be done without modification, where this is not possible only the essential modifications are required.

In our prototype implementation, the robotic framework and the components are written in Java. The component model used is the Sun JavaBean model [10]. This allows them to operate on any platform for which a suitable JVM is available. 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.

Benchmarking has been performed on our implementation to ensure the overhead introduced by the component system architecture is within realistic bounds. Table 1 shows one example of these results. It shows the timing results for code execution within the architecture and subframeworks running on the Sony AIBO using a Vision component whilst moving throughout the laboratory. The framework is running on a PIII 733 MHz desktop with wireless link to the remote AIBO. The overheads added by the architecture and associated frameworks and components are highlighted with bold text.

While the time in the system architecture accounts for 24% of the total time, this includes the start-up configuration time and processing time that would be required in any architecture used. So at a conservative estimate, the results show that the component system architecture at most introduces a 30% overhead, but in reality it is much less than this for individual actions once the robot is running. The VRL overhead is the major contributor in this latter case, thus giving less than 5% overhead during typical processing.

Execution Area	*Time (%)*
TCP Communications to the AIBO (java.io.InputStream, OutputStream)	32%
Camera display and image processing (robots.aibo.camera.*)	20%
Java processing (java.awt, java.util, java.lang, javax.swing)	18%
Component Architecture overhead (framework.Framework.run)	**11%**
Component Framework overheads (framework.*, behaviours.Schema.*)	**10%**
VRL overheads (vrl.AiboVRL.*)	**3%**
Component actuator commands (robots.aibo.MechaController)	3%
GUI display	3%
Total	100%

Table 1. A summary of the execution time of the architecture.

These results are encouraging. Given the trade-off between this overhead and increased software reuse, the additional overhead is acceptable. Consider that software reuse should lead to improved quality and performance, the additional overhead may be insignificant. There are cases where performance is required over portability, in these cases there have been interfaces provided that circumvent much of the additional processing required, but the architecture still enforces the same structure. Using these parts of the interface will increase the dependencies between the controlling code and the robot configuration.

5. Related work

There are a few robot control systems in the reviewed literature that use component-based paradigms for robotic control architectures. Some of these incorporate mechanisms for abstraction and reuse. The hardware abstraction problem has been partially addressed in some architectures ([5], [8], [9] and [18]), where hardware abstraction layers (HALs) have been designed to allow basic control functions to be ported to different robots. These control functions allow low-level abstractions such as requesting a sensor value. More abstract concepts such as *turn right*, or *is obstacle near*, require different implementations on different robots even with the abstraction provided by HALs. A mechanism is needed to provide consistent interpretation of such methods on various robots. The implementation of these meth-

ods needs to take into account the physical size and shape of the robot and the configuration of sensors and effectors. It is proposed that the use of fuzzy values to provide an abstraction from actual values returned from sensors and passed to effectors allows a more general definition of these methods. For example the actual distance that defines whether an obstacle is near will depend on the physical size of the robot and its task in progress.

The OROCOS project [5] aims at becoming a general-purpose and open robot control software package. Its goal is to develop robot control software under an Open Source licence [14] with extreme modularity and flexibility featuring software components using CORBA [13] technology. The mechanisms for abstraction are achieved through diligent decoupling of implementation from interfaces and the use of the 'object-port-connector' [6] software pattern.

Notable proprietary software also exists. The high profile MOBILITY architecture from iRobot [9] and the Evolution Robotics Software Platform (ERSP) from Evolution Robotics [8]. The MOBILITY package is a object-oriented, CORBA-based robotics control architecture. This product uses extensible "building blocks" and tools for construction of any style of robot control system. The ERSP architecture uses components to achieve its modularity and a hardware abstraction layer to facilitate reuse.

From the information available, it was found that none of these technologies provide a basis for true portability of software components. They achieve some modularity and flexibility through basic hardware abstraction, but not in the more sophisticated "virtual robot" sense.

6. Experience

In our experience, adapting a component to be portable is largely about decoupling the controlling algorithms from the implied knowledge of the robotic platform. Implied knowledge includes such things as the position and orientation of the sensors, the style of drive mechanism and how to access and manipulate the hardware of the robot. By carefully using the interfaces to define clear boundaries between components that are responsible for each of these areas, and by providing several layers of abstraction for accessing the services these components provide, we have achieved a high degree of portability of high-level components across a number of robots.

In some cases a mismatch between interfaces needed to be overcome. An example is a vision exploration component, which was originally developed for the Koala robot and is able to navigate by identifying regions of free space on the floor. It uses a simple pattern-matching algorithm looking for floor coloured segments. This was easily adapted for use by our component architecture by providing an adaptor class that implemented the `Vision` interface of the target robot's VRF. This was successfully done and tested on the Sony AIBO robot. The main problem here was mismatched method calls that required adaptation. This kind of mismatch will be reduced when new components are created with knowledge of the interface standards. However, this approach will still be required for legacy components. In this case once the adaptation was done, the component then worked across all implementations of the VRF supporting video input.

7. Discussion

The use of component frameworks to support different facets of robot control provides the opportunity to support varying styles of robot control. Component frameworks can be swapped in and out of the component system. Provided the component framework interact correctly with the component system they are able to implement specific styles of robot controls when required. Component frameworks will vary less than the software components they support, and this flexibility does not add any further dependencies on robot hardware.

The combination of a component system architecture and the VRF component framework limit the effects of diverse robotic hardware. The use of fuzzy logic and configuration specifications for robots limit the effects of environmental interaction. Using these approaches together provides support for software reuse in a complex software engineering domain. This is essential to allow robotic software engineering to mature and meet the needs of a rapidly expanding domain. Without such an approach, robotic software will continue to be specialised to particular robots and software reuse will continue to be largely non-existent. This will have adverse effects on the quality of robotic software and the effort required for software engineering process.

While this work provides for significant increases in software reuse and portability, it will not work in all cases. It is still limited by the capabilities of the underlying robot. In some cases adaptations can be made to support required functionality indirectly. It is suggested that component frameworks for specialised robot types will be created to cater for that domain, for example fixed manufacturing robots, or household cleaning robots. With or without the use of the approach suggested here, software would still be unlikely to be portable across these domains.

In conclusion this paper has highlighted the current obstructions to software engineering in the robotics domain. Through understanding these obstructions it can be seen how they create dependencies on robot configuration and subsequently limit software reuse and portability. Our approach to these problems combines the use of software component technology and abstraction techniques, to reduce the impact of these obstructions, thereby allowing a more effec-

tive application of modern software engineering practice in the robotics domain.

References

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

[2] D. Blank, H. Yanco, and et al. Avoiding the Karel-the-Robot Paradox: A framework for making sophisticated robotics accessible. In *Accessible Hands-on Artificial Intelligence and Robotics Education*, Stanford, CA, 2004. AAAI Spring Symposium.

[3] V. Braitenberg. *Vehicles: Experiments in Synthetic Psychology*. The MIT Press, Cambridge, Massachusetts, 1984.

[4] 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.

[5] 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.

[6] H. Bruyninckx. Decoupling in complex software systems. *www.orocos.org/documentation/decoupling.html*, July 2004.

[7] D. Driankov, H. Hellendoorn, and M. Reinfrank. *Introduction to Fuzzy Control*. Springer, Berlin, 1998.

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

[9] iRobot. Mobility software package. *www.irobot.com/rwi/p10.asp*, March 2003.

[10] JavaSoft. Products and APIs. *www.java.sun.com/products*, May 2002.

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

[12] O. Michel. Webots. *http://cyberboticspc1.epfl.ch/*, March 2003.

[13] O. M. G. OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, Framingham, MA, 1997.

[14] OpenSource.org. The Open Source Page. *www.opensource.org*, June 2002.

[15] N. Research Council of Canada's Institute for Information Technology. FuzzyJ Toolkit. *http://ai.iit.nrc.ca/IRpublic/fuzzy/fuzzyJToolkit.html*, October 2004.

[16] Sony. Sony AIBO. *www.sony.net/Products/aibo/*, January 2004.

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

[18] 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.