PROGRAMMING MOBILE WELDING ROBOTS FOR SHIP BUILDING USING OBJECT-ORIENTED TECHNIQUES

G Lambert, G E Tewkesbury and APF Little

Systems Engineering Research Group, Mechanical and Design Engineering Department,
Faculty of Technology, University of Portsmouth, Portsmouth, PO1 3DJ, UK.

Email: *gareth.lambert@tiscali.co.uk*

Abstract

To achieve the flexibility required by the small amount of repeated work in some industries, automated systems should be capable of being reprogrammed quickly and efficiently. One way of doing this is to create software dedicated to the task of generating the required code. Using Object-Oriented Programming techniques, a software engineer can write efficient code for machines that is faster to implement and extendable. This paper gives a brief overview of object-oriented programming and then goes on to discuss research into the use of that technique to create programs to generate code for a mobile welding robot in the shipbuilding industry.

Keywords: Object Oriented Programming, Control and Automation, Robotics, CAD, Task Machine

Introduction

The programming of automated systems within any industry can be a complex matter. In the naval ship industry it can become even more complex since a low quantity of repeated jobs can require automated equipment to be programmed frequently. Research detailed within this paper has been conducted in conjunction with VT Shipbuilding (VTS) at their Portsmouth shipyard. This software was created for a conceptual mobile arc welding robot but the techniques used are suitable for many industrial automation applications in shipbuilding.

A programming technique that most programmers are familiar with is procedural programming. This is where sets of instructions are sub-divided into procedures which can be used multiple times. A problem with this methodology is that it can become complex to debug and/or alter when dealing with large programs. An answer to this problem is Object Oriented Programming (OOP).

The paper briefly explains the history, concepts and functionality of OOP. It gives a number of examples and briefly introduces the subject. The implementation of OOP within a welding environment is discussed and information regarding how the weld process was modelled and how the software framework was constructed is presented. Robot code generated by the software systems created is then presented along with some of the assumptions made in order to improve the robustness of the code.

Object oriented programming

History: The first time that objects as entities were used in a program was in Simula 67 during the 60's. The two creators were working on ship simulations and noticed how the different attributes of different ships affected one another.

In the 70's the language Smalltalk was created at Xerox Park and the term Object-oriented programming (OOP) was introduced.

OOP continued to rise in popularity due, in part, to its compatibility to graphical user interface creation and computer games development. OOP features and functionality were added to existing languages such as BASIC, Fortran and Pascal. The addition of these features sometimes led to compatibility and reliability issues. Some modern object-oriented languages operate within programming frameworks. Frameworks include Sun's Java and Microsoft's .NET platform [1].

Concept: OOP is based upon fundamental concepts that are akin to how humans see the world [2]. However, these concepts are sometimes not how we may intuitively program a computer. This means that obtaining a firm grasp of the concepts behind OOP is important.

OOP has been increasingly used in various engineering fields. Using OOP can make system design simpler, reduce time taken for software implementation and improve extensibility[3].

Objects within OOP are used to contain not just data but also behaviour. This allows all elements within a program to be represented by objects of some kind. All objects have both data and behavioural characteristics; in this way they are similar to the real-world.

The thought process of the programmer is important to the success of OOP. In the initial stages of software creation the programmer must conceptualise a task into similar elements and then classify those elements into intuitive grouping structures called classes. Take the example of a Class called BALL as seen in Figure 1. All balls (tennis balls, footballs etc.) are members of BALL Class. BALL must contain the data and behavioural elements that are common to all balls. These classes form the building blocks for OOP and are used as templates when objects are instantiated from classes during runtime.

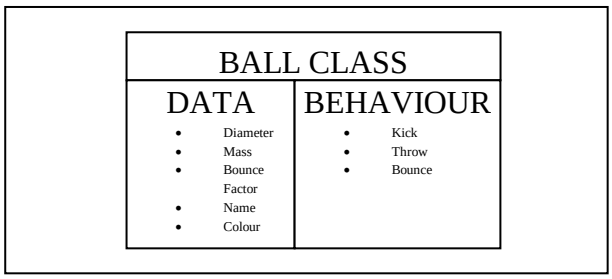

Figure 1: A Ball Class with Data and Behaviour

This object oriented approach means a programmer should not think in terms of program paths as in procedural programming. Programs are thought of as collections of objects which co-operate and interact. These interactions are initiated by events or messages which are sent between objects.

Collections of these objects are inherently data stores meaning that the program becomes data-driven as opposed to process-driven.

Functionality: Although OOP is a programming concept or technique, it is widely accepted that a true OOP language has certain functionality. The following functionalities are considered to be requisite for a true OOP language. [4,5]

Class: A class is the abstracted definition of an object. It contains both characteristic data and behavioural methods. These data and methods are traits that exist within all possible objects of that class. Classes provide the framework for object oriented programs with modularity and structure.

Object: An object is a particular sort of Class. The BALL Class in Figure 1 has data fields entitled diameter, mass etc. but these fields have no values as a Class is an abstracted definition of an object. An object of Class BALL, for example, a tennis ball, will have the same data fields and methods as the Class BALL. These fields will now have values as a tennis ball is a real object and not an abstraction.

Inheritance: Inheritance is a process by which Classes can pass their data and methods to sub-Classes. This means that sub-Classes can retain the description and functionality of their parents but can also have further functionality or description added. For example, consider a Class called HUMAN. Some of the members of the Class HUMAN may be:

- Number of Legs
- Hair
- Walk

All objects of Class HUMAN will have these attributes, to some extent. Now we may want to create a Class called ENGINEER and rather than defining every abstracted detail of ENGINEER, a programmer can use inheritance. An ENGINEER is a HUMAN and therefore inherits all the members of the HUMAN Class. The ENGINEER Class can then have additional members added to better define ENGINEER and give added functionality.

Polymorphism: Polymorphism allows a programmer to use child class members in the same way as their parent's class members.

There are two types; Overriding Polymorphism and Overloading Polymorphism.

Consider two classes that both inherit from a single parent class. The parent class is called ANIMAL and the two child classes are DOG and HUMAN. The ANIMAL class has a member called SPEAK() and both the child classes therefore inherit this member. A dog and a human do not speak in the same way; Overriding Polymorphism allows the programmer to individually code the child class HUMAN to talk and the class DOG to bark. However, both these members are called with the same command, SPEAK().

Overloading Polymorphism is when a single method signature is used to allow multiple functions depending upon the situation.

A member such as Add could need to add a pair of integers or concatenate a pair of strings. By defining one method as, perhaps, Add(int,int) and one as Add(string,string) the programmer can specify the two different methods by which the addition will take place.

This improves code readability since the same command is used in both instances and the actual routine is determined at either compile or run time.

Weld implementation

Any software written using OOP techniques must be carefully planned to provide clear abstracted models to design any required classes. The conceptualisation of welding as a task allowed the creation of a weld model.

The software hierarchy that was created is described and this software hierarchy integrated with the weld model.

After the process had been modelled and the software hierarchy had been determined, the next stage in the system creation was to produce a method by which the various elements worked together to produce a compatible program.

Weld modelling: A model was developed to describe a weld in object related terms. This was to allow any programming solution to integrate with the real world weld required. Figure 2 shows the objectified model of a weld beginning with a whole panel and working down to individual points.

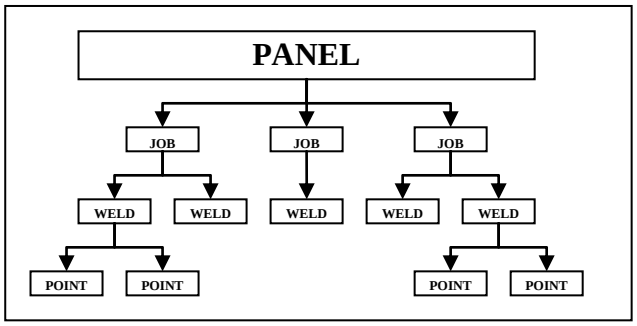

Figure 2: Hierarchy of a Ship Panel

In the same way that the construction of the superstructure of a ship is broken down into smaller elements such as sections, units and panels; the weld requirements were sub-divided. Figure 2 shows that a PANEL was considered the largest practical part. This was intuitive as a factory system can be such that PANELS have specific documentation. It was then proposed that each PANEL could be made up of collections of one or more JOBS. The inclusion of this layer allowed collections of WELDS (the next layer) to be logical grouped together in order to improve production efficiencies. The final layer was that WELDS are collections of POINTS. This was where the anatomy concept fell back into line with the Real-world. Robot programs that perform most welding were made from collections of POINTS. These POINTS described where the robot was to go.

Software hierarchy: After the hierarchical object model of a weld had been created, the software object hierarchy model was created. This was to provide a framework within which the software was created. Each layer of the model represented a different level of abstraction from the Real-world. Figure 3 shows the hierarchy of the created system when compared to Rock's Level Categorisation model [6]; it can be seen that a WELD required a robot PROGRAM. That PROGRAM was then constructed from a number of ACTIONS. These ACTIONS are determined by sub-dividing a PROGRAM into multiple tasks. A PROGRAM generated to perform a linear WELD could contain the following stages:

- Cut electrode wire to length.
- Orientate robot to weld posture.
- Move to touch sense position.
- Touch sense part to be welded.
- Recalculate start of weld.
- Weld line with positional feedback on.
- Move to safe exit position.

Each of these tasks were performed by a combination of COMMANDS. These combinations of COMMANDS were termed collections. These COMMANDS included Weld (turned the weld on) or LinearMove (moved the end effector in a linear movement). Each COMMAND was modelled using OOP techniques; this meant that to create a new COMMAND was simplified by using inheritance. When used, COMMANDS were linked to one or more INSTRUCTIONS.

An INSTRUCTION was defined as being in the Primitive Motion Layer; this was because basic code to operate the robot was emitted when called. All the documented robot instructions were modelled within the created system. This meant that, theoretically, there was no limitation to operation due to software.

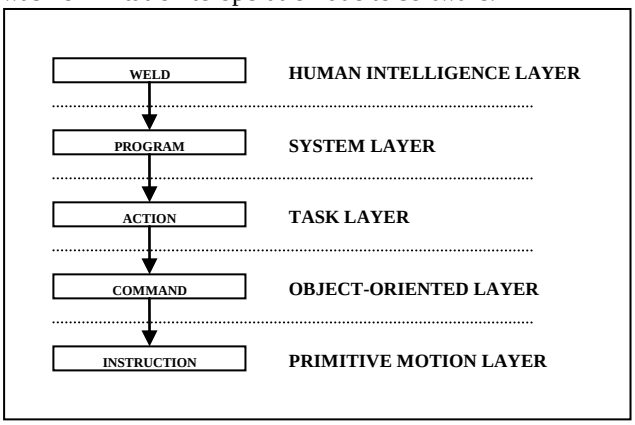

Figure 3: Software Hierarchy for Software System

Weld objects

The object-oriented elements of the code can be separated into two levels; the COMMAND objects which inhabit the Object-Oriented Layer of Figure 3 and the INSTRUCTION objects which are positioned in the Primitive Motion Layer of the Software

Hierarchy (Figure 3).

Command Objects: COMMAND object functionality was inherited into three different child classes. These classes were WeldCommand, MoveCommand and ProgramCommand as displayed in Figure 4. The primary role of these classes was to separate any sub-classes into logical groupings.

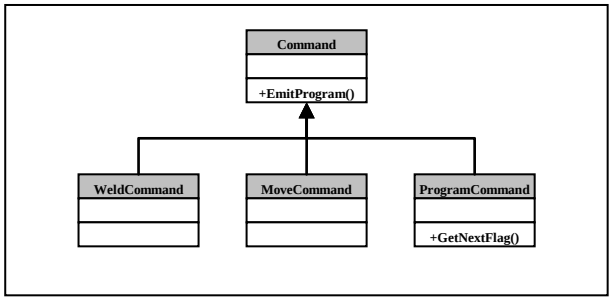

Figure 4: Objects inheriting from Command

The WeldCommand class was then inherited by two more classes called ComArcWeld and Weld, (Figure 5). The purpose of these classes was to contain all the required information needed to enable the welding process. Neither class contained any movement instructions and would always need to be used in conjunction with one of the MoveCommands in Figure 6 to perform a weld.

The mobile robot was considered to be able to perform four different types of movement. These types of movement formed the child classes of the parent class MoveCommand.

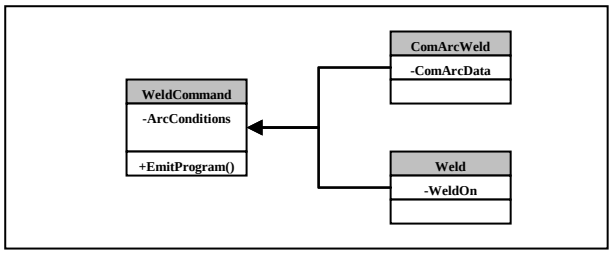

Figure 5: Objects inheriting from WeldCommand

These four child commands had many similarities which could be inherited from the parent MoveCommand. It was theoretically possible to weld with all the child commands, however, JointCommand was likely to prove difficult to control accurately. JointCommand was used only for weld posture movements which will be discussed later in the paper.

Some of the child classes of ProgramCommand are shown in Figure 7. These classes were required to provide any functionality within the robot program that was not either welding or moving. Examples of these functions were ConditionalJump, Shift and Search.

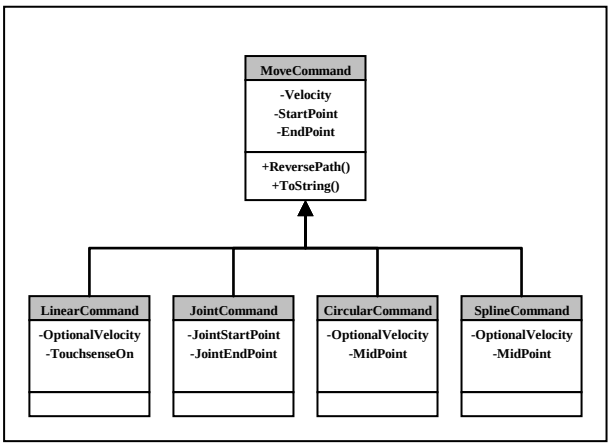

Figure 6: Objects inheriting from MoveCommand

ConditionalJump allowed a condition to be evaluated and a depending on the outcome a set of instructions would be run. This necessitated a list of commands (containing the instructions to be run) to be contained within the object. These commands were then nested in the correct place within the finalised robot code. The Search class provided an element of functionality required to be used in conjunction with a LinearMove class to achieve the touch sensing positional check.

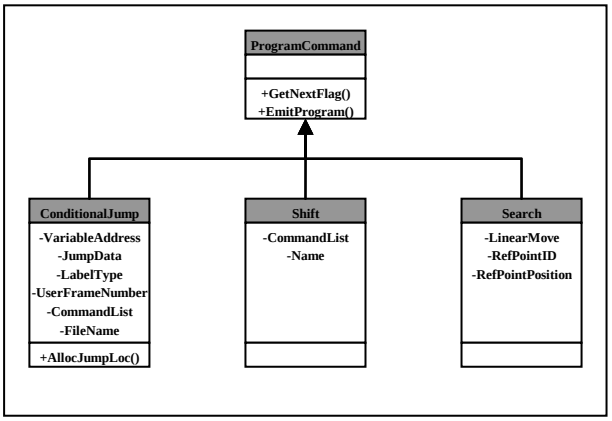

Figure 7: Objects inheriting from Program Command

Instruction Objects: This was the lowest level of the programming and generated script that the robot controller understood. When the EmitProgram() method of any descendant of Command class was run then the program emitted was a predetermined list of instructions that had been tried and tested.

Figure 10 shows some of the different positions that the end effector needed to move through to successfully weld.

The touch sense points allowed the mobile robot to determine the precise location of the part to be welded in relation to the end effector. This was important as the end effector must be positioned within 2mm of the correct weld start point to achieve a satisfactory weld quality.

Program generation

Once the hierarchy of the software and the required objects had been created, it was then necessary to create a framework that could combine the elements to generate a compatible program. The program needed to be syntactically correct in order for the robot controller to understand it.

This was achieved by the creation of a program object that modelled the requirements of a compatible robot program. This meant that all the instructional rules were extracted from knowledge of the existing system and then modelled. Some of the syntax was modelled within the instruction layer and some could only be modelled within the program object.

The program object became a collection of actions entered in order of processing. As stated earlier, actions were collections of commands, made up of instructions. The program object contained all the instructions that were required to perform the objective. The program object then generated other areas of the code that were required to maintain compatibility, such as adding positional points.

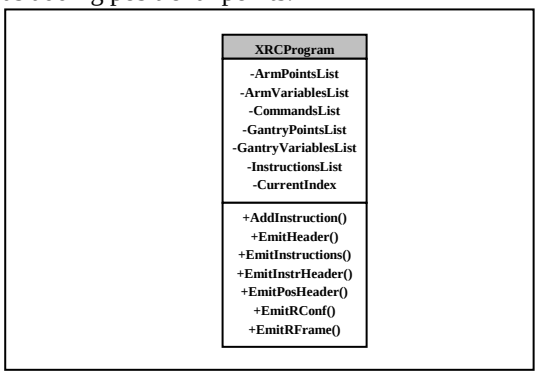

Figure 8: Program Object 'XRCProgram'

Generated robot code

Previous Sections dealt with the concepts of OOP and the implementation of those concepts into the welding environment. This Section details the actual robot code methodology used by the created system to perform a weld. The robot was considered as two separate sub-systems, a welding arm and a mobile robot. The arm was considered to be a standard robotic arm and was considered to be mounted on the mobile robot. The purpose of this was to allow the welding arm to reach as much of the ship as possible. The mobile robot had an operational area of approximately 15m by 10m.

Robot Code Methodology

Figure 9 shows the operational flowchart of the robot programs generated by the created systems. The code was kept as simple as possible to make the system more robust. The arm was used to obtain the correct posture for welding and the mobile robot was used to navigate into, along and out of the weld.
The positional offset calculation was required to allow for any inaccuracies in the position of the work piece and also in the robot system itself.

The trajectory of the end effector is shown in Figure 10. This path was determined by the requirements of the robot system and shows the necessary positional points for the corner tracking sub-system (ComArc) within the robot controller.

Robot Code Discussion

The discussion presented in this sub-Section relates to some of the assumptions made and also to some of the real-world findings of the research.

Constraining Arm Movements: The posture for welding is critical to the standard of weld quality. This posture is the same relative to any weld within the same plane. The arm system was considered to be a 6 DOF articulated model using three pivot joints and three hinge joints mounted on a mobile platform.

The calculation of the relative joint positions to achieve the correct weld posture for any weld in the horizontal plane was a complex task. To simplify this, a joint configuration was found which placed the end effector on the centre line of the main pivot joint (joint S) when in the correct weld posture. This meant that for a horizontal weld the end effector could be correctly aligned to the weld line by rotating the arm about the main pivot joint. A disadvantage was that the S joint could not revolve through 360^{o}, so an additional joint configuration was found. With these two configurations the end effector could be positioned correctly for any horizontal weld and only one joint (joint S) position needed to be calculated.

End Effector Path

The existing RinasWeld system used a method that produced a complex path to the start of the weld. The need for this was not understood and in this research that complex path has been replaced by a path which obtains the correct weld posture (as discussed earlier in the paper), then moves the end effector almost vertically above the start point of the weld line and then drops the end effector down to the touch sense point as seen in Figure 10. This is based upon the assumption that the robot could move freely even when the arm was in the weld posture position. This was not unreasonable as the end effector (the lowest point) is still over 500mm from the weld deck. Another assumption was that the end effector had a clear vertical path. In the case of large T-bar this may not always be the case.

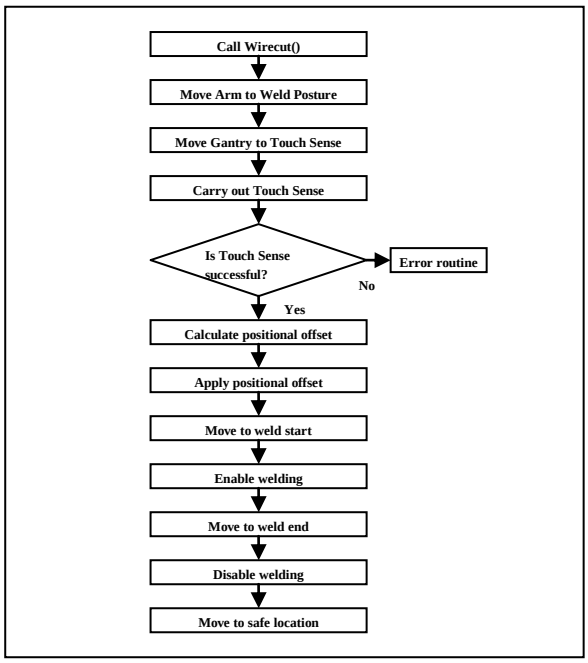

Figure 9: Flowchart showing Robot Code Operation

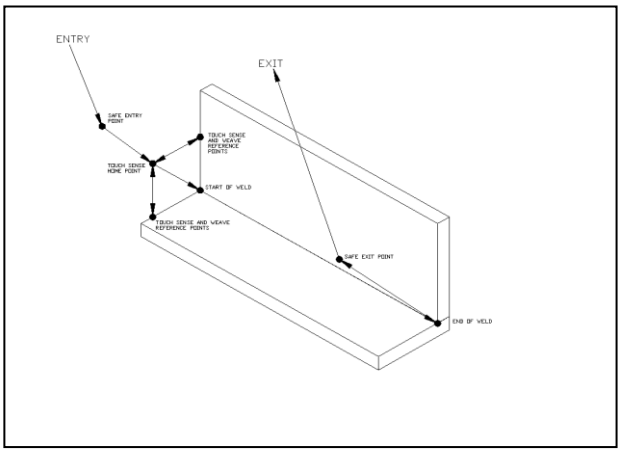

Figure 10: Welding path

This method has reduced the number of positional points to move to the start of the weld from around thirty to eight. The main benefit is not in processing time but in reliability as the calculation of those eight points is simple and highly repeatable.

Results

The system was tested by performing a straight line horizontal weld just using the welding arm. A test piece was placed in the robot welder's workspace. The start and end coordinates of the required weld were measured and the data entered into the program generation system. The generated program was then sent to the robot controller and run.

The robotic welder performed the weld in the required position on the test piece. The quality of the weld was of a satisfactory standard.

Conclusions

This paper began by giving a brief overview of the history, concepts and functionality of OOP. It stated that any software written using OOP techniques must be carefully planned to provide clear abstracted models to design any required classes.

Discussion then switched to the practical application of OOP techniques to write software capable of programming a mobile welding robot within the shipbuilding industry and the hierarchy of welding and how requirements may be achieved within a software framework.

The specific weld application robot code was introduced and that included a description of the robot code methodology and a discussion of some of the assumptions made to simplify the process.

The program was used to generate a system to perform a straight line horizontal weld. Further development of the system could include adding vertical weld or curved weld functionality.

References

1. MEYER, B., 'Object-Oriented Software Construction', *Prentice Hall*, 1997.

2. BOOCH, G. *et al*, 'Object oriented design with applications' 3rd Ed., *Addison-Wesley*, 2007

3. FANG, H. *et al*, 'An object-oriented framework for finite element pavement analysis', *Advances in Engineering Software* 38, 2007.

4. ARMSTRONG, D.J., 'The Quarks of Object-Oriented Development', *Communications of the ACM* 49(2):123-128, 2006.

5. DENNIS, A., WIXOM, B.H. & TEGARDEN, D., 'Systems Analysis and Design with UML Version 2.0' 2nd Ed., *John Wiley & Sons*, 2005.

6. TEWKESBURY, G., 'Design using Distributed Intelligence within Advanced Production Machinery', PhD Thesis, *Portsmouth: University of Portsmouth*, 1994.

.