UNIVERSITY OF VICTORIA

ENGINEERING COMPUTER SCIENCE CO-OP
WORK TERM REPORT
SUMMER 2019

Systems Engineering in Autonomous Underwater Vehicles

Author:
Alec COX
V00846488
ENGR01
Bachelor of Software Engineering
avlec@uvic.ca

Supervisor:
Mr. Aman NIJJAR

Autonomous Underwater Vehicle Interdisciplinary Club
Software Engineering - 4A
Victoria, BC, Canada

August 30, 2019

Supervisor's Approval: To be completed by Co-op Employer

This report will be handled by UVic Co-op staff and will be read by one assigned report marker who may be a co-op staff member within the Engineering and Computer Science Co-operative Education Program, or a UVic faculty member or teaching assistant. The report will be retained and available to the student or, subject to the student's right to appeal a grade, held for one year after which it will be deleted.

I approve the release of this report to the University of Victoria for evaluation purposes only.

Signature: [illegible] Position: President Date: 28/08/19

Name (print): Aman Nijjjar E-Mail: anijjar@uvic.ca

For (Company Name) Autonomous Underwater Vehicle Interdisciplinary Club (AUVIC)

UNIVERSITY OF VICTORIA

Letter of Transmittal

Work Term - 1

Software Engineering - 4A

Bachelor of Software Engineering

Systems Engineering in
Autonomous Underwater Vehicles

Dear Imen Bourguiba,

Please accept the accompanying Work Term Report entitled "Control Systems Engineering in Autonomous Underwater Vehicles."

The report is produced from work completed for AUVIC. AUVIC is located in the University of Victoria's Engineering Lab Wing. AUVIC is a club of undergraduate students from different disciplines that work together to build and compete with a AUV in the annual RoboSub competition in San Diego, California. The work done on the software systems revolves around the control system and other systems closely related to the control systems. This work involves creative problem solving and generating innovative solutions to the complex challenge of autonomous vehicle controls.

This report documents the reengineering process of AUVIC's control system. This starts with providing the reader an understanding of the problem domain. After the understanding has been established the report touches on the software reengineering process. Following the reengineering process the conclusion quickly covers over what was gained and learned from the whole process. Finishing in the fifth section describing roadblocks encountered during the process, and how they were overcome or how they could have been overcome.

I would like to acknowledge the work of other members of AUVIC's software team those being: Rory Smith, Robert Keen, Adam Kwan, and Rob Wassmann.

Sincerely,
Alec Cox

Contents

List of Figures

List of Tables

Summary

This report covers the re-engineering process of an existing software system, one designed to provide autonomous control to an underwater vehicle. Highlighting the success of the dynamic nature when it comes to on the fly customization of the state machine. Covering less positive results of the reengineering process; issues with configuration and integrating the systems that were not touched during the reengineering phase. As well as lack of testing time, both out of and in the pool, led to rushed last minute debugging at competition. And also the good features from the old system that were unnoticed successes, which resulted in problems during the competition.

Some of those good features that were missed during the re-engineering process are as follows.

- Atomic movement instructions, which allowed for pre-programmed movements to be loaded dynamically from a JSON file. Missing this in the final design led to states being significantly more complex because of their need to explicitly control how the AUV moved.
- Simple interface to resources (e.g., detection results and AUV depth). This was possible since the control system provided an interface for them. It is worth noting the way this interface was implemented very poorly.
- Simple interface to navigation commands (e.g., through atomic movement instructions). This allowed states to focus on getting the AUV setup to switch to the next state of execution. This was missed in the new design which lead to all movement related tasks to be done by hand.

The re-engineering process did however fix some of the problems with the existing system. Those problems that were fixed are listed as follows.

- Static state control system was replaced with a control system with states that can be loaded at runtime. This is similar to how movement files were loaded from a JSON file, but these are loaded from a YAML configuration file.
- Collection of too much responsibility within the control system. The solution was to separate the navigation components of the control system into different packages. However, this made access to system resources and navigation commands much more complex.

Summarizing, the positives from re-engineering the control system improved the parts of the control system that they were engineered to fix. However, the areas where the new control system were the parts left out from the existing system. The proper solution would need to bring the successes of both versions of the control system into one unified system.

Glossary

AUV Acronym for autonomous underwater vehicle.. 1

IMU Acronym for inertial measurement unit. It is a sensor consisting of usually three of the following oriented orthogonal to the other sensor of the same type: accelerometers, gyroscopes, and sometimes magnetometers. It is used to measure pitch, roll, yaw, and magnetic field. 2

JSON An acronym for JavaScript object notation. Despite its original creation for use with JavaScript it has become widely used by other programming languages. This is because of its format being easy to understand by humans, and easy to parse by computers. v

OpenCV A computer vision library, name being shorthand for open source computer vision library. 2

ROS Acronym for robot operating system. It is a C++ and python framework for development of robotics systems. It provides useful tools for device level abstraction, inter-process communication, and much more. See the libraries website for more information here. 2

RPM Acronym for revolutions per minute. 3

1 Introduction

1.1 Pretext

AUVIC, is a team of students whose objective is to design, build, and compete an AUV in the yearly RoboSub competition in San Diego, California. The team provides students with opportunity for hands on experience in submarine design, and work on complex autonomous systems. This kind of experience opens the clubs members up for many opportunities for Co-ops and post graduation careers.

The team has evolved in its membership durastically over the last year, as many long standing members graduate, and take on Co-op jobs, and new less experienced members take the reigns. The club as a whole has a small number of students in relation to previous years which limits what we can accomplish. The club is composed of three different teams: mechanical, electrical, and software. The report will be dealing entirely with the software side of the AUV.

1.2 Report Contents

This report documents the migration from an existing control system. The report starts off with explaining and understanding the problem domain from the perspective of the control system, which will help to inform the reader about what the operational constraints are of the system, and the task which it is trying to accomplish. Software re-engineering occupies a majority of the report and servers for the discussion, and meat of the report, this section covers the different processes that will inform the software redesign. Those processes being: reverse engineering, restructuring, and forward engineering.

After the design and implementation the report concludes with reflection on the software system produced from the redesign. This is done in two parts, a conclusion which talks about the results of the system featured at competition, which outlines the strengths and weaknesses of the new system abstractly. The report then concludes the reflection with a detailed analysis of what failed and succeeded and suggestions of how to fix the failure that was outlined.

2 Problem Domain

2.1 Operational Environment

To start understanding how the system works we first need to understand the environment in which the system operates. The system runs in Ubuntu 16.04 on an NVIDIA Jetson TX2 Development kit. This operating system is required to support the use of the ROS Kinetic software, its use will be explained more below.

The system is written in two different programming languages, C++ and Python. Most of the code written on the AUV is C++ this is done primarily to increase the performance of the concurrent systems, and this languages' type system ensures code correctness. Python, being used to a lesser extent, is used for image processing and the web interface for managing the system. Using Python for image processing seems sub-optimal when the system uses C++ because of its performance benefits. However, the image processing interfaces for Python are more simple than C++, and since the Python code doesn't need to be compiled (C++ needs to be compiled) the Python code can be tested and adjusted significantly faster than the C++ code.

These languages are used as they work with our development environment which consists of two primary libraries, ROS and OpenCV. ROS acts as the backbone for the AUV as it is used to launch all the different required sub-processes, and manage the inter-process communication between them. ROS also aids with hardware abstraction, but that isn't seen much in this report as the report focuses on the higher level components of the system. OpenCV is used to handle retrieving still images from the cameras and processing of those images for image recognition, pattern and feature matching, and cascade filtering to detect various objects for the RoboSub competition.

The existing system consists of several different packages, two of which are impacted by the development of a new control system; those being ai and nav. The ai package handled the state machine functionality of the AUV, also handling the detection systems as well. While the nav package handles direction messages from the ai packages' control system segment.

The ai package manages the information need to inform the control system about the external state. Currently this is done with camera input and depth input from the power board. This was intended to be expanded to support other devices, such as hydrophones and a IMU. The other devices used currently, camera and power board, have proven to give enough control from use in previous competitions. However, the addition of the IMU would allow for more accurate measurement of the orientation of the submarine; which would allow for a more accurate heading. Furthermore, the addition of hydrophone functionality would allow for the AUV to speed up execution of some tasks at competition.

This is because the competition course provides two underwater pingers (sound emitting sources) that can be used to navigate quickly to sections of the course. As currently the AUV must search for those sections of the course with the cameras.

The navigation package is responsible for taking in navigation messages and then processing them within the navigation package itself. There is a fair amount of redundant separation of duty occuring within this package. As well as several different processes that are being spawned to handle very small tasks. An example of this is a class that takes in a message containing the desired velocties and orientation and converts that to RPM values for each of the motors.

3 Software Re-engineering

This section covers a simplified process of software re-engineering. Rhat is a process for taking an existing system developing an understanding of how it works, then refactoring that system and engineering changes, then implementing the system as to the conclusions made during refactoring.

Reverse engineering is the process of reviewing the existing software systems and documenting how they function. This can be done through diagramming, documenting, and refactoring. This process allows the engineer to understand the problem domain of the system more, and also provides information on parts of the software that need to be restructured and changed.

Restructuring is the process of taking an existing design and making changes to improve the design. This is the main transitional phase that will happen after the original understanding of how the current system works through reverse engineering. Restructuring will involve reorganizing code as granular or abstract as desired. Most restructuring will focus on making small changes to the system to make it more maintainable or run more efficiently. However, the restructuring of the current system will appear to be more of an overhaul.

Forward engineering is the process of taking the results of the system specification created from the two previous steps of the reengineering process integrating the system into the code-base.

3.1 System Understanding

The system is arranged for the most part into different functional groups, referred to as packages. There are some exceptions to this as seen in the "ai" package, where the responsibility of controls, vision, and movement is all amalgamated together.

The flow of information is handled quite elegantly within the system. With lower level systems passing information up to and receiving back from higher level systems. See Figure 3.1 for understanding the general idea of how this flow happens.

The individual devices are handled via their own ROS nodes, which then make the information from the devices available in a 'friendly' way. This allows for specific driver-like programs to abstract the possibly complex interface to the physical devices.

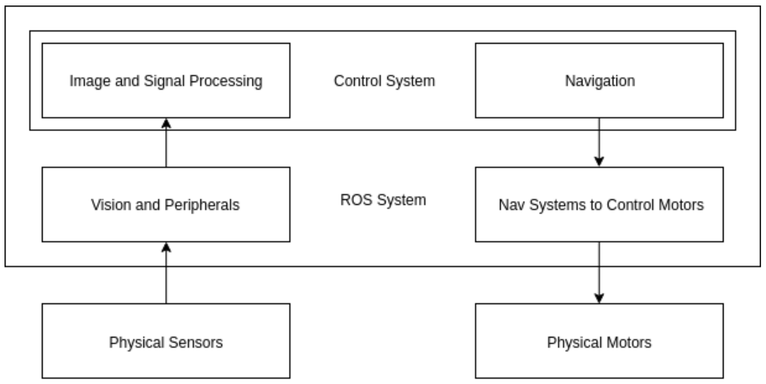

FIGURE 3.1: Diagram showing how information flows in the reference frame of the control system.

3.2 Perceived Problems

The control system featured no error catching or any kind of protocol to recover from an error state. For most states the control system loops on a single state then after a condition is met will advance to the next state. This type of control system is autonomous controlling but it isn't designed for the kind of failures expected in the real world.

The control system works directly with the state information and directly handles both the condition to switch states and the assignment of the next state, see Figure 3.2.

The control system also directly handled the vision systems. The control system did offer a simple interface for the states in the state machine to utilize.

3.3 Restructuring

3.3.1 Overview

The control system has been assigned excessive responsibility; it performs the tasks of a control system, detection system, and navigation system. The detection systems within the control system shall be separated into a vision package and offer an interface through ROS inter process communication structures. Namely offer the same features the state machine was using, and expand the features on top of that. And the navigation system uses should also be pulled out, but an interface still be made available through ROS interprocess communication structures. To use the two new systems identified above they would need to be compatible with the existing system structure.

For the navigation system this required either using interface that existed and offered functionalities by replaced system, or a new system that acts as an interface to manage communication between old and new software systems. The latter involved much more consideration for best creating the system for usage in the future. While the former got the

```
    while(ros::ok())
    {
        switch(am.fsm_state)
        {
        case(dive):
            am.run_submerge();
            am.fsm_state = stop;
            break;
        case(dead_reckon_gate):
            am.run_forward();
            if(--state_count <= 0)
            {
                am.fsm_state = gate_detect;
                state_count = am.gate_detect_count;
                am.scanner_en = true;
            }
            break;
        case(gate_detect):
            am.scanner_en = true;
            am.run_forward();
            if(am.gate_passed)
            {
                am.fsm_state = dice_detect;
                state_count = am.dice_detect_count;
```

FIGURE 3.2: Code segment showing the control systems way of handling states, controlling submarine functions, and getting information from other submarine systems.

job done, and allowed to expand and slowly update functionality later on. Thus, the former of the two options was chosen, for the reasons given and because of time constraints for the project.

The detection systems' functionality and usage was restricted to within the control system. This allowed creation of a new system much simpler and allowed much freedom with the design of this system. Which is why the system used a simple interface to retrieve detection results, and change detection type. And this system was incorporated directly into the vision package, as having close proximity to the retrieved image files would mean for faster image processing. However, this document's purpose is the redesign of the control system and this won't cover the new design of the detection system.

The updated version of the ROS communication diagram was too large to embed into this document, but it has been provided here.

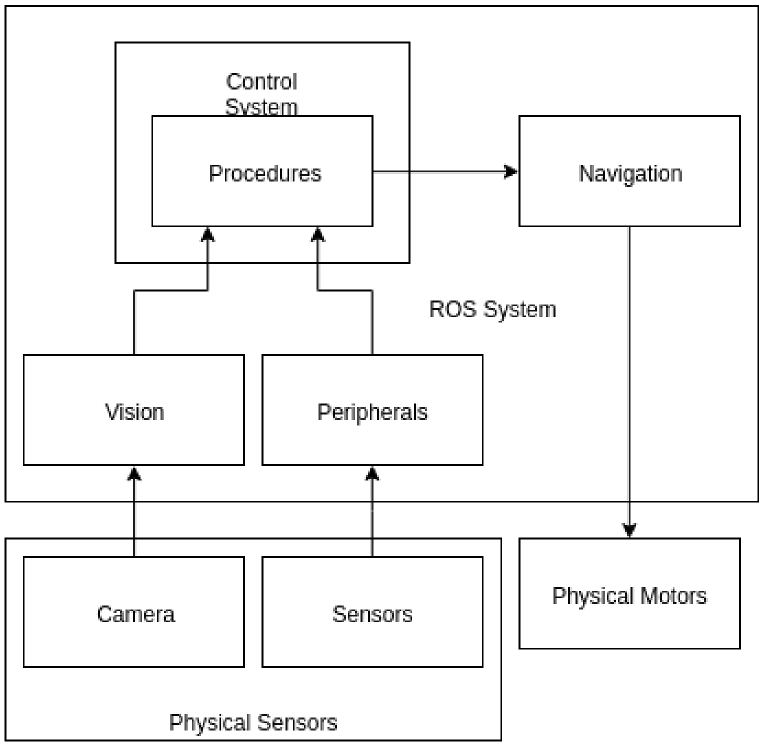

FIGURE 3.3: Diagram showing how information flows in the reference frame of the redesigned control system.

```yaml
states:
  dive:
    procedure: DiveProcedure
    error: DiveProcedure
    next: idle
  idle:
    procedure: IdleProcedure
    error: idle
    next: idle
  surface:
    procedure: SurfaceProcedure
    error: surface
    next: surface
```

FIGURE 3.4: Shows a simple example idle configuration file. Demonstrating use of simple state organization and definition of state transitions.

3.3.2 Control System Specification

This section will focus on the specific design of the new control system that is intended to replace the existing control system. Specifics will be provided for the format of the dynamic state configuration file, as well as the procedures that will be defined to use with the new state machine.

State Machine - Configuration Specification

State configurations are define in a YAML file and are loaded as an argument through the launch file for the control system package. The configuration file must have a root tag 'states' which is a dictionary of either all state or state lists. To differentiate between a state and a state list restrictions must be placed on what must be in a state, and what must not be in a state list. It is also to note that the root dictionary, labeled states, would be classified as a state list.

State lists are dictionaries containing keys being the state names, and values being the state parameters. State lists must not use reserved words, listed below in Table 3.1, or required state names, listed below in Table 3.2 their names.

States are dictionaries containing the fields listed in Table 3.1. State names cannot be the same as the name of the fields that the states themselves have as part of their dictionaries, as this would severely complicate the parser.

State Machine - Procedures

Each procedure will be derived from the base procedure class. This allows the use of virtual functions to ensure that each derived procedure class is implementing the appropriate

Field	Description
procedure	This defines the c++ procedure that this state relies on.
error	This specifies the path name of the state this state transitions to for error behaviour.
next	This specifies the path name of the state this state transitions to for normal behaviour.

TABLE 3.1: List of reserved words for states and their meaning.

State Name	Description
dive	This state represents the start state for the state machine.
surface	This state represents the end state for the state machine.

TABLE 3.2: List of required states and their expected function.

methods and also to simplify creation of procedure objects when loading states dynamically.

```
class Procedure {
  public:
  // Enum class used to literately inform control
  // system of meaning of return code.
  // (e.g., accessed via Procedure::ReturnCode::NEXT)
  enum class ReturnCode : int {
    FATAL,
    ERROR,
    CONTINUE,
    NEXT
  };

  // Left default, but derived procedures are free to have
  // various members and initialize them however they please.
  Procedure() = default;


  // This is the function for carrying out the
  // procedure functionality will return a result
  // which informs the state machine when it's
  // ready to advance to the next state, continue with
  // the current state, or if it's encountered an error.
  virtual ReturnCode operator()() = 0;
}
```

3.4 Forward Engineering

3.4.1 Preparing Procedures

The state machine needs to have a list of all the available procedures that it has to work with before it can begin to parse the states from the configuration file. This is done by first creating a map with the key being a string with the same name as the procedure object, and a value being an instance of the function object. This allows for the parser to look up the procedure from the provided name, and throw an error if a procedure cannot be found. When the procedures are loaded from the parsing new instances of the function object are created, which allows for state of the function object to be unique to the state of control system it is being loaded with. It is also worth noting that if behaviour was desired where each state in the control system had state information from a procedure of the same type being used elsewhere in the system that could be accomplished by using static object members.

3.4.2 Dynamic State Loading

Implementing the configuration parser that would be used to parse the configuration provided was made a lot easier through ROS' parameter server library. Using it the configuration files are automatically broken up into a logical representation of what was in the configuration file. The parser starts with the root tag of the configuration file which it parses as a state list, then identifies all the items within as either a state or a list of more states. If it's a state it will generate a logical representation of that state, otherwise it will add it to the list of things to parse like it did with the root state list. The only required work was to structure code to look for identifiers for when the parser had encountered a state or if it needed to add more suspect states to the

4 Analysis

This section covers both the failures and success stories of the system. It is worth noting that these were uncovered after the systems were integrated, and testing was being done preparing for the competition and at the competition.

4.1 Failures

4.1.1 Resource Access

The system design did not provide a simple mechanism for acquiring resources from submarine. Complex access lead to procedures being more complex than required, which caused an increase in procedure development time. This was because the complexity of the code caused implementation to take longer because the procedures are compiled C++ code. The issue can be mitigated by implementing an interface that provides access to resources through a globally available singleton that abstracts access to all available resources. Alternatively, some abstraction offered by the control system to specific resources which the procedure would need to request.

4.1.2 Navigation

The system design missed out on the easy interface for movement that was seen in Figure 3.2. With the predefined movement commands it was easier to add new states with different movement characteristics. This was completely overlooked on the new system design, the idea of writing procedures to complete individual tasks didn't inherently support this idea of using predefined movement commands. But seeing the results from a similar approach to navigation controls as was seen in resource access.

This again caused more complex code which took longer to develop. Although it offered the highest degree of control, it was unnecessary as most of the procedures should have the movement abstracted away much like the detection system. This also led to procedures requiring to have access to ROS level inter-process structures as seen in lines 168 to 170 of Figure 4.1.

4.1.3 Testing

The system was designed and implemented correctly. The control, navigation, and detection systems worked individually and ran without trouble. However, during competition

```
class DiveProcedure : public Procedure {
        ros::NodeHandle n;
        ros::ServiceClient set_heading;
        ros::Subscriber depth;

        double desired_depth;
        double current_depth;
        public:

                void depthUpdateCallback(navigation::depth_info message)
                {
                                desired_depth = message.desired_depth;
                                current_depth = message.current_depth;
                }

                DiveProcedure()
                : n{},
                set_heading(n.serviceClient<navigation::nav_request>("/navigation/set_heading")),
                depth(n.subscribe("/navigation/depth", 1, &DiveProcedure::depthUpdateCallback, this))
                {}

```

FIGURE 4.1: Code segment shows new per-procedure overhead.

there were several problems getting the submarine launched with the three new systems with the other existing systems.

Issues arose with the launch files that are used to start all systems required for the AUV to function. Which was missed during testing of the systems individually, and after implementing the changes mentioned above would need to spend a significantly longer amount of time on.

Looking into getting testing frameworks for simulations like Gazebo would allow us to ensure our systems are working with minimal amounts of in water testing. Gazebo would allow us to run a model of the AUV in a simulated environment with emulated sensors. Which would allow testing to be done from each developers computer. This is the highest priority item to ensure that issues are caught before competition, to avoid from last minute rushed troubleshooting.

4.2 Successes

4.2.1 Dynamic States

The dynamic loading of the state machine accomplished its goal of making the state machine operate in a less complicated way. The loading of the states dynamically based off a configuration file allowed functionality to be added and removed easily. The only downside of this system is that despite the states being loaded in dynamically the procedures needed to be statically typed and compiled to work properly this way. Changes that could make this better would be allowing the procedures to also be loaded in dynamically through python or some other language.

4.2.2 Procedures

Procedures were a success in that the idea of what they should do was correct. That is to say that despite the interfaces with accessing resources for and navigation for moving the AUV the design was correct.

4.2.3 Vision and Navigation Systems

The separation of duty from the original control system into the vision and navigation systems allowed for a more maintainable control system. Though this did lead to issues as mentioned earlier about the complexity of the systems interfaces. If the procedures were to be provided with a simple interface to the navigation system and vision system the procedures would be easier to write and more effective.

5 Conclusion

5.1 Section 1

The implementation of a new control system to handle the control of an AUV is no simple task. The design focused on fixing the failures of the existing system, while keeping some of the successful features in the design. However, the new design missed out on an abstracted interaction with the rest of the submarine systems. This lead to more complicated procedures, which hindered the on-the-fly development of different procedures.

The dynamic loading of states also sparked the idea of creating a system for writing dynamic procedures in some language that would be interpreted and loaded into the control system on startup. This would remove the requirement of changes of procedures resulting in recompiling the control system package. Would allow for procedures to be defined alongside the configuration files, and allow rapid-fire prototyping with the ability to tune the procedures and immediately test against a running system.