UNIVERSITY OF VICTORIA

ENGINEERING COMPUTER SCIENCE CO-OP
WORK TERM REPORT
SUMMER 2019

Control 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 28, 2019

UNIVERSITY OF VICTORIA

Letter of Transmittal

Work Term - 1

Software Engineering - 4A

Bachelor of Software Engineering

Control 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

Glossary

apple a fruit. 1

1 Introduction

1.1 Pretext

AUVIC, is a team of students whose objective is to design, build, and compete an AUV. The team evolves as the students at the university join, enter Co-ops, and graduate. The team provides students with opportunity for hands on experience in submarine design, and work on complex autonomous systems. Talk about team. Teams history. Role in team. Different sub-teams. Communication and meeting between sub-teams and members.

1.2 Report Contents

This report documents the migration from an old statically programmed control system with more responsibility than needed to a dynamic control system and two other systems taking over the excess responsibility. Starting off with explaining and understanding the problem domain from the perspective of the control system. The report then ventures into the domain of software reengineering where three different processes are described that occur. Those processes being: reverse engineering, restructuring, and forward engineering. The report then concludes with an analysis of the failures and successes of the produced system. All previous followed by the conclusion which expands on what was learned from the entire process.

1.3 Glossary

AUVIC - autonomous underwater vehicle interdisciplinary club
apple

2 Problem Domain

2.1 Operational Environment

The development for the AUV's high-level software system is developed in C++ and python. These languages are used as they work with our development environment. Our development consists of two primary libraries, ROS and OpenCV, running on Ubuntu 16.04. ROS, short for - Robot Operating System - is a amalgamation of different libraries useful for the development of robotic systems. We primarily use ROS for inter-process communication management, and hardware abstraction. OpenCV is a software package for the processing of images and video, used to give the AUV understanding from visual inputs.

The existing system consists of several different packages, two of which are impacted by the development of a new control system; those being "ai" (which would be renamed to controlsystem) and "nav" (which would be renamed to navigation). The ai package handled the state machine functionality of the AUV, also handling the detection systems as well.

Existing systems. Inter-connectivity through ROS communication structures (that diagram). Programming languages, and technologies.

Hardware. Hydrophones, hydrophone boards, power board, IMU, Jetson TX2. Diagram of hardware connections.

3 Software Re-engineering

This section covers a simplified process of reverse engineering, that 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.

Information in the AUV flows from the physical input devices: IMU, Hydrophones, and Cameras

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

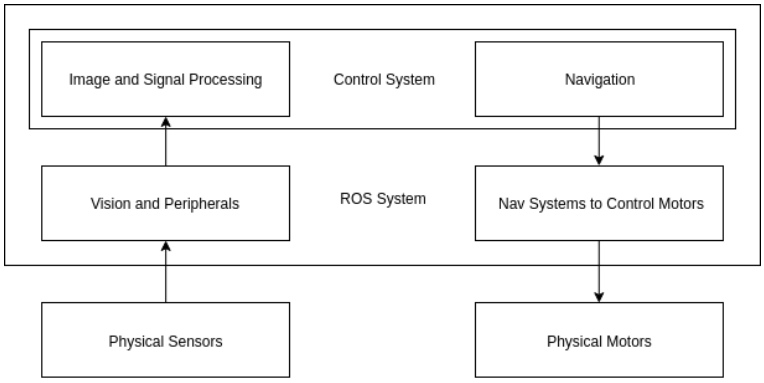

FIGURE 3.1: Diagram showing how information flows in the reference frame of the control system.

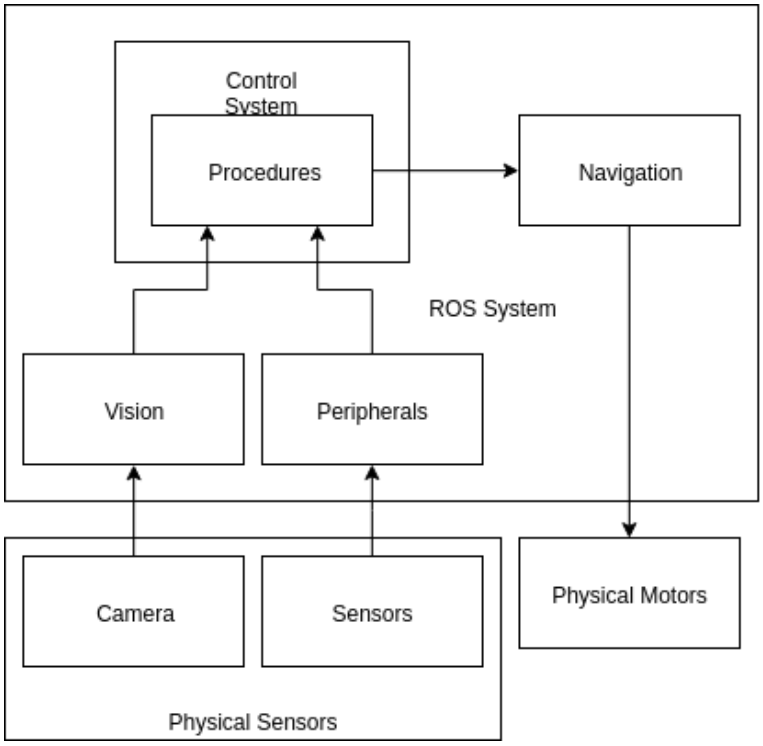

FIGURE 3.2: Diagram showing how information flows in the reference frame of the redesigned control system.

```
507     while(ros::ok())
508     {
509         switch(am.fsm_state)
510         {
511         case(dive):
512             am.run_submerge();
513             am.fsm_state = stop;
514             break;
515         case(dead_reckon_gate):
516             am.run_forward();
517             if(--state_count <= 0)
518             {
519                 am.fsm_state = gate_detect;
520                 state_count = am.gate_detect_count;
521                 am.scanner_en = true;
522             }
523             break;
524         case(gate_detect):
525             am.scanner_en = true;
526             am.run_forward();
527             if(am.gate_passed)
528             {
529                 am.fsm_state = dice_detect;
530                 state_count = am.dice_detect_count;
```

FIGURE 3.3: Code segment showing the control systems way of handling states, controlling submarine functions, and getting information from other submarine systems.

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 3.3.

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

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 inter-process 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 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.

The primary more generalized features to handle the support for different kinds of detection being: access to result of detection and the ability to change the current active detection algorithm which could also be set to none.

3.4 Forward Engineering

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. This lead to procedures being more complex than required, which caused an increase in procedure development time. This was because the complexity of the code lead to longer implementation and debugging time. This 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 3.3. 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.

4.2 Successes

The systems modularity and on-the-fly customization worked exactly as intended. This would allow for varying the ordering tasks.

```
167 class DiveProcedure : public Procedure {
168         ros::NodeHandle n;
169         ros::ServiceClient set_heading;
170         ros::Subscriber depth;
171 
172         double desired_depth;
173         double current_depth;
174         public:
175 
176                 void depthUpdateCallback(navigation::depth_info message)
177                 {
178                                 desired_depth = message.desired_depth;
179                                 current_depth = message.current_depth;
180                 }
181 
182                 DiveProcedure()
183                 : n{},
184                 set_heading(n.serviceClient<navigation::nav_request>("/navigation/set_heading")),
185                 depth(n.subscribe("/navigation/depth", 1, &DiveProcedure::depthUpdateCallback, this))
186                 {}
187 
```

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

```
1  states:
2    dive:
3      procedure: DiveProcedure
4      error: DiveProcedure
5      next: idle
6    idle:
7      procedure: IdleProcedure
8      error: idle
9      next: idle
10   surface:
11     procedure: SurfaceProcedure
12     error: surface
13     next: surface
```

FIGURE 4.2: Shows a simple example idle configuration file. Demonstrating use of simple state organization and definition of state transitions.

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.