

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:  Position: President Date: 28/08/19

Name (print): Aman Nijjar 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

<b>Letter of Transmittal</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Pretext . . . . .	1
1.2 Report Contents . . . . .	1
<b>2 Problem Domain</b>	<b>2</b>
2.1 Operational Environment . . . . .	2
<b>3 Software Re-engineering</b>	<b>4</b>
3.1 System Understanding . . . . .	4
3.2 Perceived Problems . . . . .	5
3.3 Restructuring . . . . .	5
3.4 Forward Engineering . . . . .	7
<b>4 Analysis</b>	<b>9</b>
4.1 Failures . . . . .	9
4.1.1 Resource Access . . . . .	9
4.1.2 Navigation . . . . .	9
4.1.3 Integration Testing . . . . .	10
4.2 Successes . . . . .	10
<b>5 Conclusion</b>	<b>12</b>
5.1 Section 1 . . . . .	12

# List of Figures

3.1	Original Information Flow . . . . .	5
3.2	Direct State Handling . . . . .	6
3.3	New Information Flow . . . . .	8
4.1	New Procedure Overhead . . . . .	10
4.2	Example Configuration File . . . . .	11

# 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. 3

**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



# 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. Development of the system is written in 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.

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.

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.

## 3 Software Re-engineering

This section covers a simplified process of software re-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.

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.

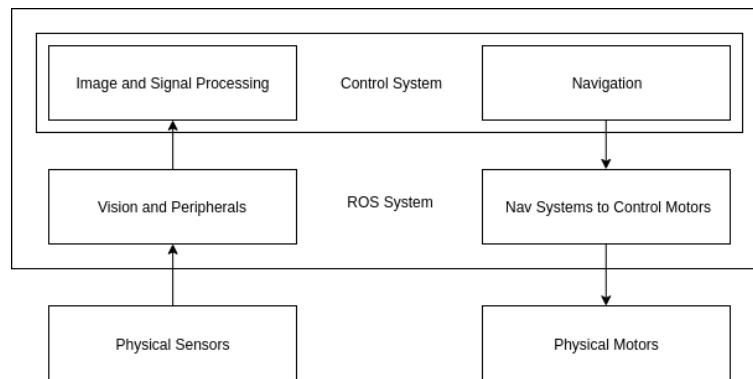


FIGURE 3.1: Diagram showing how information flows in the reference frame of the control system.

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.

## 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

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

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

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.

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

## 3.4 Forward Engineering

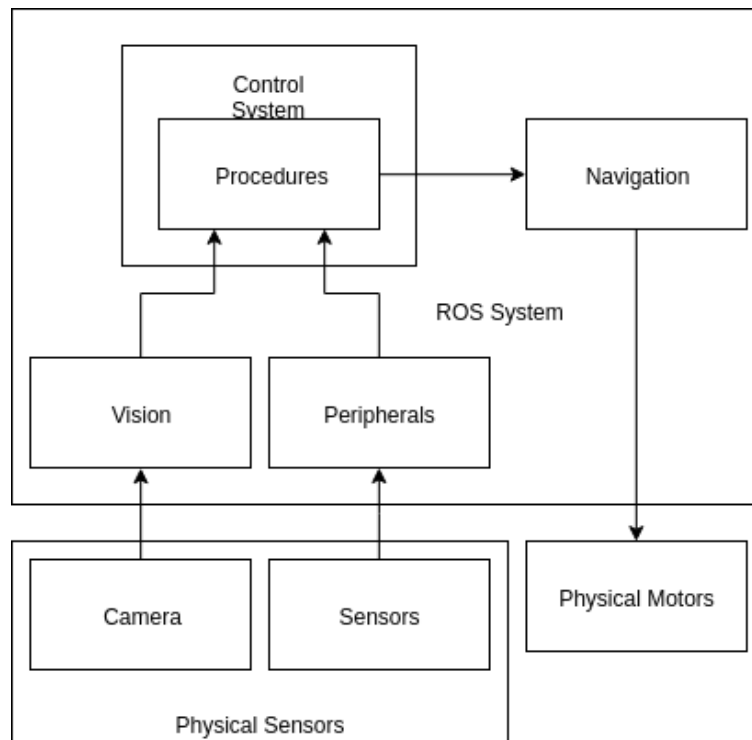


FIGURE 3.3: Diagram showing how information flows in the reference frame of the redesigned control system.

```

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



## 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](#).

```
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.

### 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 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**

### **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.