A Structured Testing Framework for ADAS Software Development

by

Sachin Fernando

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Mechanical and Mechatronics Engineering

Waterloo, Ontario, Canada, 2023

© Sachin Fernando 2023

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis contains contents that were previously accepted for publication in IAVVC 2023 for which Sachin Fernando was the primary author. Co-authors for this paper include Ansar Khan, Dr. Roydon Fraser and Dr. William Melek. Some low-level design documentation listed in Section 3 was originally written by Ansar Khan, as well as the initial version for **Observer** based monitoring in ROS 1.

Abstract

A major task in the design of automated vehicles is the need to quickly and thoroughly validate a development teams algorithms. There currently exists no explicitly defined common standard for developers working on Advanced Driver Assisted Systems to adopt during their software testing process. Instead different teams customize their testing process specifically to their software systems current needs. Literature indicates that these processes can be comprehensive but convoluted, and not flexible to change as test requirements and the system itself does. This thesis introduces a test framework at the unit, integration, and system test levels with the objective of addressing these challenges through a complete test framework centered around rapid execution and modular test design. At the unit test level a recommendation guide is put forth that is largely aimed at new developers with concrete actionable items that can be integrated into a teams process. For integration and system level testing, a software solution for **ROS** based development referred to as University of Waterloo Structured Testing Framework (**UW-STF**) is described in regards to both the benefits it provides as well as its low level implementation details. This includes how to tie the framework into using data generated from the popular simulator **CARLA** for end-to-end testing of a system. Lastly the test framework is applied to the codebase of **UWAFT** for their development efforts related to connected and automated vehicles. The framework was shown to increase readability/clarity at the unit test level, facilitate robust automated testing at the integration level and provide transparency on the teams current algorithms performance at the system test level (average **F1-score** of 0.77 and average **OSPA** of 2.42). When compared to the standard ROS integration test framework, UW-STF executed the same test suite with 60%+ reduction in lines of code and meaningful differences in **CPU** and memory requirements.

Acknowledgements

First and foremost a thank you to my primary advisors: Dr. Roydon Fraser for his unwavering support of the EcoCAR program at the University of Waterloo and Dr. William Melek for his regular and direct guidance of my thesis via one-on-one sessions throughout the course of my second year. I would have had a much more difficult time staying on course without this. An additional thank you to professors Dr. Derek Rayside and Dr. Chris Neilson for participating in the formal review and acceptance process of my thesis as part of the advisory board.

Thank you as well to Ansar Khan, who had the original idea to apply an Observer based testing framework to ROS. Without his initiation of the project, my efforts at extending it to ROS2 and adding functionality to it would not be possible.

In terms of broad support for the University of Waterloo's EcoCAR team, a thank you to Eugene Li and Dr. Derek Rayside who served as supervisors to our teams trips to yearly competitions, the Student Design Center staff led by Graeme Adair and Dr. Peter Teertsra for their patience, and support and all the staff supporting the teams day to day operations including finances, shipping, the machine shop and facility safety inspections. An additional thank you to Ross McKenzie and WatCAR for their help with track access. Surrounded by the hectic schedule of competition and our own academics we may have forgot to thank those listed above at times, but their work is truly essential to continuing our teams participation in North America's best automotive collegiate competition.

To all the organizers of the competition, especially those from General Motors and Argonne National Laboratory, thanks for overwhelming amount of funding, support and mentorship that is provided to students such as myself. Organizers such as Lucas Shoults, Jesse Alley, Priyash Misra, and Nick Goberville who I've had the opportunity of interacting with directly have all contributed positively to the program. I'm sure I'm missing many more names.

To my UW EcoCAR team, past and present, thank you for supporting the program. It has been the most intense, enjoyable, challenging, and inspiring program I've ever been in all at once. Every moment from the highest highs to the lowest lows have been a learning opportunity for me and I'll be carrying a lot of those lessons with me to my future workplace. A special thank you to the undergraduate leads that stuck around amidst COVID when our team was on the brink, and took on 10x the workload to keep the team going. They are: Aryan Gosalia and Bahar Kholdi-Sabeti (Connected and Automated Vehicle subteam); Geoffrey Qin and Naim Suleman (Connected Software subteam); Michael Done and Haocheng Zhang (Propulsion Controls and Modelling subteam); Will Stairs, Mohammed Abdullah, Kevin Rao and Jason Nguyen (Propulsion Systems Integration subteam) and Daniel Montazeri (System Safety).

Dedication

Dedicated to my parents Shyama and Hasitha, for giving me everything I've ever needed to succeed in life, and then some.

Table of Contents

Author's Declaration ii
Statement of Contributions iii
Abstract iv
Acknowledgements v
Dedication vi
List of Figures ix
List of Tables xi
List of Abbreviations xii
1. Introduction 1
1.1 Background of Industry and Academia Involvement 1
1.2 Problem Statement 2
1.3 Objectives and Proposed Contribution 3
1.4 Organization of this Thesis 4
2. Literature Review 5
2.1 Unit Testing and Automation: Current Practices 5
2.2 Current Methods of Integration testing 9
2.3 Current System Testing Approaches 12
2.4 Chapter Summary 16
3 Test Framework Architecture 17
3.1 Unit Testing Guide 17
3.2 University of Waterloo Structured Testing Framework (UW-STF) 21
3.2.1 Design Intent and High Level Functionality 21
3.2.2 Detailed Design 21
3.2.3 Addressing Design Intents 25
3.3 Generating Simulation Data 27
3.3.1 Choice of Simulator 27
3.3.2 Proposed Simulation Pipeline 28
3.4 Chapter Summary 30
4 Application of Framework to the UWAFT Stack 31
4.1 UWAFT Architecture 31
4.1.1 Software Architecture 31

4.1.2 Hardware Architecture 33
4.2 Unit Testing 34
4.3 Integration Tests 39
4.4 System Tests 46
4.4.1 Metrics 46
4.4.2 Results 48
4.5 Quantitative Performance Analysis 52
4.6 Chapter Summary 53
5 Conclusion 54
5.1 Summary of Work and Current Limitations 54
5.2 Future Work 55
References 57
Appendix A – Template Observers 63
Appendix B – Software Architecture Diagram 66
Appendix C – Network Diagram 67
Glossary 68

List of Figures

Figure 1: Systems engineering "V" diagram [9] ... 5
Figure 2: CI practices summarized by Soares et al. from previous research. [14]. ... 8
Figure 3: Testing framework used in [22] ... 11
Figure 4: Model based design approach to ADAS algorithm design and testing in [29] ... 13
Figure 5: How TestWeaver fits into the system test architecture in [30] ... 14
Figure 6: GRAIC architecture as shown in [31] ... 15
Figure 7: TDD process as depicted in [35] ... 20
Figure 8: UW-STF architecture diagram for Nodes A & B under test ... 22
Figure 9: Contents of .sim file used in UW-STF ... 23
Figure 10: Sample .simresults file associated with the .sim file shown in Figure 9 ... 25
Figure 11: Given an ADAS system with several modules (ROS nodes), the scope of testing may be changed by selecting the appropriate topics. Integration testing (above) and system testing (below) is shown as defined for the following pictured system. ... 26
Figure 12: CARLA provides a flexible suite of environments (top left), vehicles (top right), and sensors (bottom) [49] ... 28
Figure 13: Top five most common two-vehicle light-vehicle crash scenarios as according to [47]. ... 28
Figure 14: Two of the several CARLA Autonomous Driving Challenge scenarios from [51] ... 29
Figure 15: Ground truth sensors initialization function ... 29
Figure 16: Illustration of the full testing pipeline including simulation input ... 30
Figure 17: Definition of UWAFT system used in Section 4 examples ... 33
Figure 18: FOV depiction of stock sensors plus team added LiDAR and camera. Note that FOV and range are not to scale. ... 34
Figure 19: Example of old unit test organization for object tracking module ... 35
Figure 20: New folder structure and naming convention for unit tests as per recommendation #6 ... 35
Figure 21: Highlighted text showing documentation requirement added to team's default merge request template ... 36
Figure 22: Old test code (top) and new test code after application of AAA (bottom) ... 36
Figure 23: The benefit of AAA is more pronounced for more involved tests such as those with multiple assert requirements which are grouped together as opposed to spread throughout ... 37
Figure 24: Test that relies on more external dependencies (a) and test with fewer dependencies (b) ... 38
Figure 25: Example code coverage dashboard from coveralls.io ... 38
Figure 26: Project test runners are assigned using Gitlab’s CI settings screen ... 39
Figure 27: Stationary approach scenario seen in CARLA ... 40
Figure 28: Location of Heartbeat Observer within system ... 40
Figure 29: Setup parameters of .sim file (left) including Observer section (right) ... 41
Figure 30: Results file from running Heartbeat Observer on sensor topics ... 42
Figure 31: Location of In Range Observer within system ... 42

Figure 32: Ground truth plot for two target vehicles in scenario 1 43
Figure 33: Sensor fusion algorithm results overlayed on ground truth for scenario 1 (top) and the corresponding the Observer entry in the .simresults file. 43
Figure 34: Location of frequency observer within system 44
Figure 35: Frequency Observer definition for tracked object data specifying 8Hz 44
Figure 36: Output of frequency Observer test when an 8Hz minimum frequency is specified.... 45
Figure 37: Location of Max Observer within system 45
Figure 38: Max Observer definition for max tolerable number of tracks (left) and corresponding results file output showing that output was equal to max limit (right) 45
Figure 39: Number of tracks detected over course of scenario 1 46
Figure 40:Visualization of precision and recall as seen in [56] 47
Figure 41: Notation description for Equation 4 from [58] 48
Figure 42: Key fields for each system test Observers is highlighted in their corresponding .sim file. F1-score is shown on the left while the OSPA is on the right. 49
Figure 43: Annotation of CARLA and RViz visualization consisting of the ego vehicle, targets (tracks) and ground truth lines (green) 50
Figure 44: Ground truth and system tracking results for scenario 1 50
Figure 45: System test Observer results for scenario 1. F1-score shown on the left and OSPA shown on the right. 51
Figure 46: The moment in simulation where a vehicle turns into the ego vehicles lane (Scenario 2) 51
Figure 47: F score and OSPA results of scenario 2. F1-score is worse than the pass score so the test fails. OSPA however is better than the pass score so it passes. 52

List of Tables

Table 1: Sensor Detail Overview 33
Table 2: Lines of code comparison 53
Table 3: Difference in runtime, CPU and memory requirements of UW-STF compared to SIT . 53

List of Abbreviations

AAA	Arrange-Act-Assert
ACC	Adaptive Cruise Control
ADAS	Advanced Driving Assisted System
ADS	Advanced Driving System
AV	Autonomous Vehicle
AVTC	Advanced Vehicle Technology Competition
CAGR	Compound Annual Growth Rate
CAN	Controller Area Network
CARLA	CAR Learning to Act
CAV	Connected and Automated Vehicle
CI	Continuous Integration
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
EVC	Electric Vehicle Challenge
FCW	Forward Collision Warning
FCM	Front Camera Module
GUI	Graphical User Interface
LDW	Lane Departure Warning
LRR	Long Range Radar
NHTSA	National Highway Traffic and Safety Administration
OMAT	Optimal Mass Transfer
OSPA	Optimal Sub Pattern Assignment

ROS	Robot Operating System
SIT	Standard Integration Testing
SRR	Short Range Radar
TDD	Test Driven Development
UWAFT	University of Waterloo Alternative Fuels Team
UW-STF	University of Waterloo Structured Testing Framework

1. Introduction

1.1 Background of Industry and Academia Involvement

The market for automated vehicles continues to show accelerated growth and indicates no signs of stopping in the coming decade. In fact, analysis by Market.us forecasts the Autonomous Vehicle (**AV**) market size (which includes both semi-autonomous and fully autonomous vehicles) to exceed $3.4 billion USD by 2032 resulting in a Compound Annual Growth Rate (**CAGR**) of 38.8% between 2023 to 2032 [1]. AV research started gaining traction in North America in the 80s when universities, in conjunction with the U.S. Defense Advanced Research Projects Agency (**DARPA**) began development and testing of these vehicles. DARPA's autonomous land vehicle was a 12 ft tall robot tasked to traverse a path without human intervention. Since then the DARPA Grand Challenge series has challenged research groups to complete fully autonomous navigation of courses. Some say that this Challenge was one of the multiple markers of the transition of implementation of automated features from solely academic research pursuits to industrial development as well [2]. In the 1990s and then 2000s, semi-automated features for consumer vehicles evolved alongside the state-of-the-art research in academia. These features included Adaptive Cruise Control (**ACC**), Forward Collision Warning (**FCW**) shortly after, and then Lane Departure Warning (**LDW**) in the 2000s. Though popularized in mainstream media by Tesla, the first level 2 system actually around in 2013 when Mercedes launched a production vehicle with both integrated lateral and longitudinal control. Tesla's 'Autopilot' released to consumers shortly after in 2015.

All that to say, industry and academia have often gone hand in hand by complementing state of the art research with more practical but limited solutions for vehicle automation. This relationship reveals two key points that are relevant to the research of this thesis.

1. **With the demand for automated driving solutions growing, so too does the demand for tools that support its research/development.** With the vast majority of consumer vehicles possessing at least one Advanced Driver Assisted System (**ADAS**) feature, there is a clear trend towards increasing automation. Billion dollar spending on full autonomy by companies such as Cruise and Waymo further highlight this trend. With such a trend for increased demand of automation, comes the need for correspondingly greater research and development in the fields supporting it. This includes everything from the development of perception algorithms, corresponding sensor hardware, compute resources and even the testing tools to support the development process itself. In the safety critical world of automated driving, the risk of functionality breaking code changes becomes even more severe. Modern vehicles are now in the realm of 100 million lines of code and with increasing autonomy this number does not look to trend downwards [3]. With this comes the need for even more 'continuous integration' as per Mihailovici, the Managing Director of Porsche Engineering Romania [3]. This thesis thus addresses one such implementation of continuous integration relevant to the use case of automated vehicle algorithm development.

2. **The relationship between industry and academia is mutually beneficial and should be considered when developing tools for automated driving.** As highlighted in the section above discussing the progression of automated vehicles throughout time, it is rare not to see both industry and academia playing a part in pushing its development forward. **DARPA**'s Grand Challenge for example brought together industry, academia and the government to push the needle on this frontier. Another competition that has done so in a similar fashion is the Advanced Vehicle Technology Competition (**AVTC**) series. The AVTC series was established in 1988 by the U.S Department of Energy and Argonne National Laboratory in partnership with North America's automotive industry [4]. The multi-year competition tests some of North America's leading engineering institutions to develop the next generation of automotive engineers. The most recent installment in the AVTC series, the EcoCAR Electric Vehicle Challenge (**EVC**), challenges schools across North America to re-design powertrain, consumer-facing and semi-autonomous features of a 2022 Cadillac Lyriq using industry standards and practices. Competition supporters include the U.S. Department of Energy, General Motors, Mathworks and many industry sponsors. Delving into the background of a stakeholder collaboration project such as the **EVC** is relevant for two reasons. Firstly, this thesis research was conducted as part of the University of Waterloo Alternative Fuels Team's (**UWAFT**) development efforts in year 1 of the EVC, and thus the product of this thesis, was designed with the consideration of a student design team work environment in mind as one of the users of the final product. As opposed to a conventional research setting, student team competitions elicit an environment that is fast-paced, highly collaborative, and has high member turnover (e.g., due to changing academic terms, member graduation, etc.). With this in mind, a key consideration for the work presented is that the *interface is simple to grasp, easily repeatable and intuitive in its workflow*. The second reason the EVC is relevant is because it represents a major use case for the proposed work. Of course solutions designed for the limited use case of a *specific* research group or company setting may fare well in that setting, but we sought to find a solution that would be flexible enough to work in almost any environment it was applied to correctly. As explored later in this thesis, design decisions relevant to the proposed work were made with the intent of open-source software that is accessible to all groups.

1.2 Problem Statement

To support the work of industry and academia, we focus on the testing aspect of **ADAS** software development. This section provides a summary of the areas that need to be addressed and why they serve as motivation for the underlying objective of this thesis. Specifics of shortcomings based on current literature are provided in Section 2: Literature Review.

One of the consistent themes that arise after consultation of the literature is the extent of choices that exist in the realm of tools for **ADAS** software testing and development. Though not an evident problem by itself, the multiplicity of proposed options online leave development groups with a lot of choices but little direction in the set of tools that would best fit their use case. This manifests itself in different ways at the different levels of testing. In the realm of **Unit testing**,

findings by Daka and Fraser in [5] indicate that developers have an unclear priority on what makes an individual test high quality. At the integration and system test level, different problems persist. Due to the increased complexity of testing, frameworks tend to be more rigid and specific to an application or a team's software platform. Increased testing time is often a consequence of such complex frameworks, as validated in developer surveys such as [6] which note that a majority of **ADS** (Advanced Driving System) developers cited the need of "speeding up ADS testing" as *very important* or *important*. Additional requirements for test frameworks from the study included that they should support multi-module software **Stack**s, custom metrics and consideration of simulation data.

1.3 Objectives and Proposed Contribution

Based on the discussion in the previous section, the main challenges with respect to **ADAS** software testing that need to be addressed are:

- Rigid testing frameworks that do not lend themselves to usage across different development teams (e.g. due to use of complicated/unclear setup process) as well as different stakeholders or software platforms (e.g. proprietary software usage)
- Multiplicity of software tools and testing techniques to choose from with no standard or direction for ADAS development teams to follow
- A test framework that meets the needs of the developers using them including:
    - Tests that support both custom and standard metrics for evaluation
    - Support for multi-module ADAS stacks as the more common architecture
    - Consideration of real world scenarios and crash reports to drive development
    - Adaptable to use of both simulated and real world data

Note that development will be based around the usage of **ROS** (Robot Operating System) which is the key middleware suite used by **UWAFT**. From [7], ROS "is an open-source framework that helps researchers and developers build and reuse code between robotics applications". It boasts of many benefits including compatibility with C++ and Python languages, a code base applicable to various hardware platforms (robotics, automotive, drones etc.) and most importantly open-source development meaning it is open to contribution and use from any industry and academic institution. Because of its usage in the following key phases of the proposed framework, note that '**Node**' here onwards refers to a ROS node. As per official ROS documentation [8], a **Node** is an executable which uses a ROS client library to communicate with other nodes. They are able to publish or subscribe to a **Topic** which is how data is transferred between nodes.

Thus the central objective of this thesis will be to develop a full scale open-source test framework based on ROS including considerations at the unit test, integration test and system test level that facilitates rapid and modular testing to address the challenges stated above. In addition, it is important to note that the framework is centered around operation in a software-in-the-loop test environment. From here onwards, usage of the general term "test framework" refers to the combination of all three levels of testing as it relates to the proposed contributions below. The contribution at each level of testing is described below:

Unit testing (i.e., testing of individual functions or methods within a **Node**): Because unit tests happen at the lowest level and are highly application/algorithm specific, no novel development contributions are made to standardize this. Alternatively, a guide to unit test creation is provided based on a thorough literature review of current best practices and addressing areas of improvement. As per the study on current developer perceptions of unit testing practices [5], a common pain point is difficulty is how to write a high quality test itself. The provided guide to unit test development will help to supplement this with specific examples of development and common pitfalls that new developers may be prone to.

Integration testing (i.e., testing of a subset of nodes of the entire 'system'): For this and the next phase, a custom developed test framework known as the University of Waterloo - Structured Testing Framework (**UW-STF**) is employed and the benefits of it to streamlining the testing process are shown. UW-STF is a framework that focuses on being simple to customize, integrable with Continuous Integration (**CI**) tools, and modular / easily integrable to different software platforms. It works based on the concept of monitoring nodes called **Observer**s, which compare a target **ROS Topic**s data to an expected value or pass criterion.

System testing (i.e., testing of the entire system (all nodes) as a whole): The system testing phase expands on the functionality of **UW-STF** to include all nodes under test. The difference here is that the Observers produce system level metrics. UW-STF allows for custom metrics to be defined in addition to standard ones which was listed as a key takeaway from the work of Lou, Deng, Zheng et al. [6]. Additionally at the system test level, a pipeline from simulation to testing is included to address timeliness challenges associated with scenario generation and selection, again identified in [6].

1.4 Organization of this Thesis

This thesis is organized as per the following layout:

Chapter 2 provides an overview of the related work in the field with a focus on references to current literature. As well, shortcomings of current solutions are highlighted to give way to the potential benefits of the proposed solution.

Chapter 3 covers the contents of the test framework (testing guide for **Unit testing** and architecture of **UW-STF** for integration and system level testing). This includes how it works and a justification of design decisions based on the design intents it was set to meet.

Chapter 4 then moves on to a walkthrough of the test framework when integrated with **UWAFT**'s existing codebase as part of their **ADAS** software development efforts for year 1 of **EVC**. This includes specific examples of how the framework improves the quality of the codebase at the uniting testing level, and how it improves modularity, ease of use, and automation at the integration and system test levels.

Finally Chapter 6 covers conclusions made from the test cases including limitations of the work and future improvements required to address them.

2. Literature Review

Common to projects of large scale and of complex development requirements, the industry standard "V" diagram (shown in Figure 1) is used to illustrate the relationship between development and validation cycles of an automated vehicle system. As evident in the forthcoming literature review, many current solutions to testing are specific to a testing phase and thus lack modularity as the product/feature moves through different levels of the V diagram.

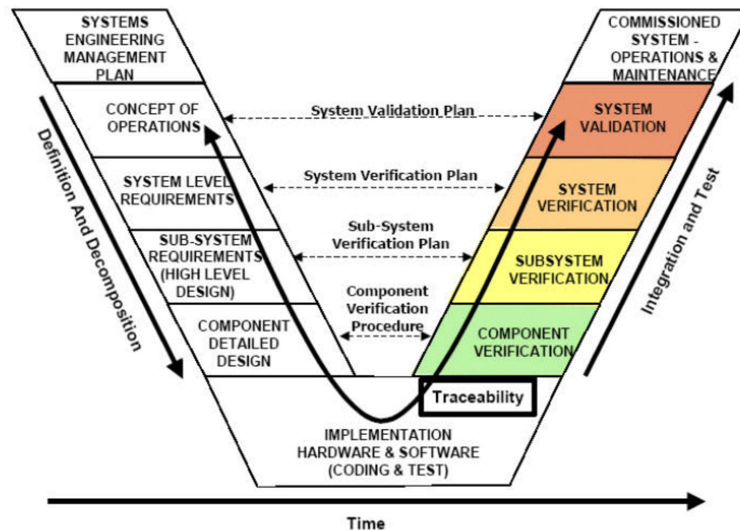


Figure 1: Systems engineering "V" diagram [9]

The remainder of Section 2 two delves deeper into each of the three levels of the testing framework covered in this thesis – unit, integration, and **System testing**. Specifically, many sources proposes their own pipelines for each level, while others note weaknesses of that level thereby helping to understand what the proposed framework may help address.

2.1 Unit Testing and Automation: Current Practices

Section 1.2 introduced the work of Daka and Fraser, in which a mass survey was conducted with developers to gauge to what extent current **Unit testing** methodologies served a purpose [5]. In addition, it sought to highlight any areas where further research was necessary including what part of current processes could use improvement. The survey included participants from a wide range of personal and programming backgrounds. The first key response was that unit testing itself was largely driven by management requirement and developers' own conviction. From this it can be surmised that the adoption rate of new techniques must have benefits quantifiable enough to convince management as well as immediately evident enough to convince developers. Thus any proposed solution would benefit from improvements to user interface and performance so that users can both see and measure their newly improved development pipeline. The second question in [5] looked at what activities take the most time in the development process. When including things like debugging and fixing tests, developers perceived to spend close to half their

time on “testing”, indicating just how important streamlining this process would be. In conjunction with this, fixes to the test itself were seen as a much more common approach than fixes to the code itself. This was especially prevalent with less experienced developers who admitted to deleting tests significantly more than their experienced counterparts. Tools / guidelines in writing tests at the early stages of a developer’s career might be a way to address this uncertainty in test writing. Complimenting this finding was the response to the follow-up question of *how* developers approach test writing. Though most respondents claimed to use a systematic approach to testing, many also admitted to not having a clear prioritization of what the acceptance criteria for a ‘good’ test even was. Lastly in the realm of how unit testing could be improved, developers again had a lot to say. In writing unit tests, identifying the code to test was ranked as highest, followed then by isolating the test and then determining what to check as the second and third greatest challenges. Though support such as code coverage tools exist, developers may not know how to use them or have a good grasp of which ones are applicable to the development they are conducting. In the matter of failing tests, ‘flaky’ tests were cited as the highest concern. This could be attributed to things like non-deterministic code or environmental factors affecting the outcome. Other issues such as difficulty in code test interpretation or a lack of test applicability were also cited. Again the issue of a lack of direction when writing tests seems relevant. The last question asked from participants was general to their overall perceptions of unit testing of which a clear majority stated an interest in more tool support as the test writing itself was not an enjoyable aspect of their development experience. Summarizing the findings of Daka and Fraser as relevant to this thesis, there seems to exist a general confusion on best practices to unit testing and a lack of direction on what characterizes a high quality, robust test suite. Furthermore, tool support would be of great benefit considering the many hurdles and often quickly outdated status of previously written tests.

Next a review of some of the empirical research related to **Unit testing** is provided:

The first case study is by Bhat et al. in which the effect of using Test Driven Development (**TDD**) on code quality as well as time spent on writing unit tests is analyzed [10]. In this study, projects from two different divisions in Microsoft are compared to similar ones within their division. The metrics used are defect density and effort estimates which were provided by team managers. Both cases showed a reduction in defect density when TDD was used by a factor of 2.

In a similar study on the use of **TDD** by Nagappan et al., software quality measure by defect density was shown to be reduced between 40-90% given its deployment, at the expense of initial development time which increased 15-35% [11]. A key recommendation of this paper was the automation of the testing process in response to this increased initial development time. Some other key recommendations to note are to: start TDD from the beginning of the project and introduce automation shortly after as well; add tests incrementally as problems arise whenever they arise; track the project using quantifiable metrics such as code coverage; and share unit tests within the team to help identify integration issues as early as possible.

A study conducted by Williams et al. involves the automation of **Unit testing** [12]. In this case study, a company team transitioned from ad hoc unit testing practices to the more

standardized NUnit automated testing framework for version 2 of the product. Comparisons in terms of test defects and development time were made to the previous version. For version 1, the development process was kept quite linear, following the general flow of:

feature list creation → design documents → design and test plan reviews → coding and debugging → build verification tests → executing test plans

Additionally automated test cases were rarely every written with the majority of test cases being private to the developer and usually one-time use. Execution of the test plans then occurred by running manual ad-hoc testing. Version 2 followed a similar initial process in terms of feature lists being used to draft up design documents by developers. However, the key difference was that automated unit tests was mandated from the beginning in addition to the build verification tests that needed to be passed before checking in code. By the end, almost four times the number of automated unit tests had been written for version 2 including a shared bed of unit tests for development collaboration and transparency. In the end, the version 2 of the produce achieved a 20.9% decrease in test defects. Similar to previously mentioned case studies however, this came at a cost of increased development time. Of greatest importance was the effect this had on the end user. Data indicated that there was a relative decrease in customer-reported failures. Consistent with the results from other case studies, the conclusion drawn was that code quality improved while total defects decreased. Lessons from this case study include: management support for unit testing is necessary to keep it active in the pipeline; single tool mentality is important to minimize complication (i.e. tool standardization); unit testing must be open to contributions across the team; and execution of the tests should be as simple as possible.

Generalizing the effect of test automation in general (that is, not just limited to **Unit testing**), Kumar and Mishra show its positive effects on cost, quality and time to market for a software application [13]. Specifically, they use leverage **CI** to calculate the return on investment from test automation. The experiment is set up by monitoring metrics relating to cost, quality, and time to market on three separate software projects. For example software quality is measured by Functionality, Reliability and Maintainability measurements which are defined in Equations 1-3 respectively:

$$Functionality\ F = \frac{number\ of\ features\ fulfilled}{total\ number\ of\ features\ required} \quad (1)$$

$$Reliability\ R = \frac{mean\ time\ to\ failure}{mean\ time\ to\ failure + mean\ time\ to\ repair} \quad (2)$$

$$Maintainability\ M = \frac{time\ spent\ on\ testing}{total\ development\ time} \quad (3)$$

Results indicated consistently higher metrics across all fields for the project with test automation. The drawbacks of test automation were found to be high implementation and maintenance costs. However, the authors of the paper still favour the use of automated testing due to highly beneficial returns in the long run. To address the drawbacks of high implementation and maintenance costs, the proposed solution would benefit from using open source software and being highly intuitive to reduce developer time commitments when using it.

Soares et al. take a broader look at the effects of continuous integration in a systematic review of the literature in [14]. They ask: What are existing project criteria to justify the use of CI? What are the claims of the effects of CI on software development? And which methods are used in the studies that investigate these effects? Looking into the current practices of CI, the study summarizes those proposed by Duvall et al. [15] and Fowler [16] as seen in Figure 2:

	Duvall et al. practices [13]	Fowler practices [24]
Integration Practices	Commit code frequently	Everyone commits to the mainline every day
	-	Maintain a single source repository
Test Practices	Write automated developer tests	Make your build self-testing
	All tests and inspections must pass	Test in a clone of the production environment
	-	Make it easy for anyone to get the latest executable
	-	Automate deployment
Build Practices	Don't commit broken code	Automate the build
	Run private builds	Every commit should build the mainline on an integration machine
	Fix broken builds immediately	Fix broken builds immediately
	-	Keep the build fast
Feedback Practices	Avoid getting broken code	Everyone can see what's happening

Figure 2: CI practices summarized by Soares et al. from previous research. [14].

Of highest relevance in their findings were the results related to question 2 of the study, i.e. what are the effects of CI? These effects were categorized into several themes:

1. Development activities: The most significant mentions across literature were that CI is associated with increases in productivity and efficiency. As well several more studies claimed to associate it with an increase in confidence, presumably associated with the quality of the resulting product. Negative associations were also documented with several other studies referencing added complexity when CI is involved.
2. Software processes: In regards to effects on software processes, the highest number of claims across the literature were that CI positively affected release cycles, were associated with an increase in cooperation, and that it improved process reliability. On the negative side was an association with more technical challenges when CI is integrated into a development pipeline.
3. Quality assurance: The majority of studies claimed that CI had a positive impact on test practice and was related to an increase in quality assessment. There was also reliable evidence on the association between CI and increasing code coverage.
4. Integration patterns: The standout claim was that CI had a positive impact on pull request life cycle and integration practice in general. A smaller number of studies claimed it to be

associated with a commit pattern change, which aligns with theme #2's findings of "increased technical challenges".
5. Issues and defects: Defect reduction was among the most common claims. For defects that existed however, studies claimed an overall decrease in time to address them based on the metric of issue resolution rate.
6. Build patterns: Most studies related to build pattern mentioned a strong association with build health, thereby contributing to more successful builds.

The themes across literature show quite confidently a handful of measurable positive effects when **CI** is integrated into a software development pipeline. Specifically, doing so translates to increases in productivity, team collaboration and final product quality itself. It is however important to keep in mind the additional workload that CI entails for a team. If CI is chosen to be integrated into a test framework, it must be done in a way that limits additional development workload but has a huge potential for long term benefits. Donca et al. quantified some of these improvements by implementing a custom CI pipeline system and comparing it to manual testing and Gitlab CI systems [17]. They found pipeline runtime differences that were improved by a significant order of magnitude and pipeline infrastructure costs that were also much lower than their manual counterparts.

2.2 Current Methods of Integration testing

Given that the current research on the practices and effects of **Unit testing** have been presented it is logical to do the same for the next level of testing as well: **Integration testing**. Ernits et al. cover one approach to integration testing which is model-based using **ROS** [18]. **ROS** infrastructure is known to have plenty of well-documented support for unit testing including leveraging testing tools such as Google's *gtest* and Python's *unittest*. The same cannot necessarily be said for higher level integration testing. This paper therefore extends on the Robot Unit Testing methodology introduced by Bihlmaier and Worn in [19] (which uses a simulation environment and control software to test robot performance) but have introduced functionality to measure code coverage as well as drive the robot through scenarios. Uppaal Tron is used as the text execution engine in which the output of the robot when provided a certain input action, is checked for equivalence against a target value. In addition code coverage is checked at the end of each scenario. The study concluded that their test engine setup was able to achieve a high degree of code coverage and incorporate multiple scenarios to check robot functionality. Additionally, automating generation of models allows valuable validation of difference scenarios. However, the topological map approach becomes quite convoluted because of the focus on low level robot signals when expanded to more complex systems. Moreover, code also needs to be modified for higher fidelity levels of testing as specific signals/interfaces change. As highlighted in [14], the burden of increased technical complexity in a software pipeline is a relevant concern, so care must be taken that too involved integration practices don't deter from adopting integration testing itself.

Bures proposes an integration test framework for IoT solutions which is a hybrid of conventional testing protocols [20]. In it he describes three common concepts in testing / quality assurance.

The first is simulation which benefits from “a reduction of potential expensive testbed composing of physical devices”. However, it is still known to be limited in accuracy and not meant to be a 100% replacement for hardware testing. The next two concepts are sub-styles of **Integration testing**: unit integration testing and end-to-end (E2E) integration testing. In unit integration testing, specific interfaces are put under test by corresponding API calls and compared to an expected results. E2E testing on the other hand verifies system functions or sub-systems as part of the system as a whole. The higher-level interaction of the units under test lend itself to catching problems of higher technical complexity. The downside is that it may be less clear which part of the sub-system is the issue. Due to complementary benefits/drawbacks of the different methods, Bures proposes a solution that allows developers to reap the benefits of all three testing concepts, i.e. a framework that uses simulation as an input to the system, leverages Junit’s E2E testing capabilities and a configurable library that allows integration testing to start early in development and be integrated quickly because tests just need minor modifications. Though the specific tools and frameworks from IoT testing cannot be apply to the domain of **ADAS**, the principles of the framework can be. Namely, designing a system that incorporates simulation and either unit or E2E integration testing. In this way the system is exposed to multiple potential fault scenarios while minimizing full system scale setup complexities.

In their project based on a testing infrastructure in **ROS**, Lasaca et al. approach **Integration testing** from a completely different angle [21]. The second of the two tools developed in their project known as TopicFuzzer is based on the concepts of compiler sanitizers and fuzzing. Specifically, they sought to test a single ROS nodes external API. The TopicFuzzer employs a grey-box testing technique in which a parameterized drive program sends variations of input to explore a **Node** under test in a systematic way. The benefit of this testing solution is that it is scalable and fully automated. After testing the solution with a case study, the benefits of an easy to use solution that achieve high code coverage in a timely manner are seen. The additional layer of automation when parameterizing test inputs is an often overlooked but time-consuming aspect of integration. Automation using a tool such as TopicFuzzer may be an effective tool to remedy this.

Brito, S. Souza and P. Souza propose their own framework for **Integration testing** which is easier to connect to the domain of **ADS** [22]. As it relates to **ROS** the researchers write that the integration testing in question involves evaluating the communication of publishers/subscribers between different components in the robotic system. They note that ROStest, a built-in integration testing framework suffers from a lot of additional code development required and limited types of input data. Instead, the researchers apply a model-based test (MBT) technique to evaluate the functionality and performance of a real-time system. An overview of their testing approach is used in Figure 3, which inspires some of the design elements in the proposed testing framework. First test scenarios are generated using a simulator. Then the test criterion is set to identify target functionalities. A **ROS** communication and abstracted robot system graph is then generated to assure adequate communication channels are designated. Following this the system is executed with the evaluation of test coverage done in parallel. Once a test report is generated, it is used to inform the creation of new tests. Ultimately applying it to the case of separate development paths provided evidence of the applicability of the approach in other scenarios.

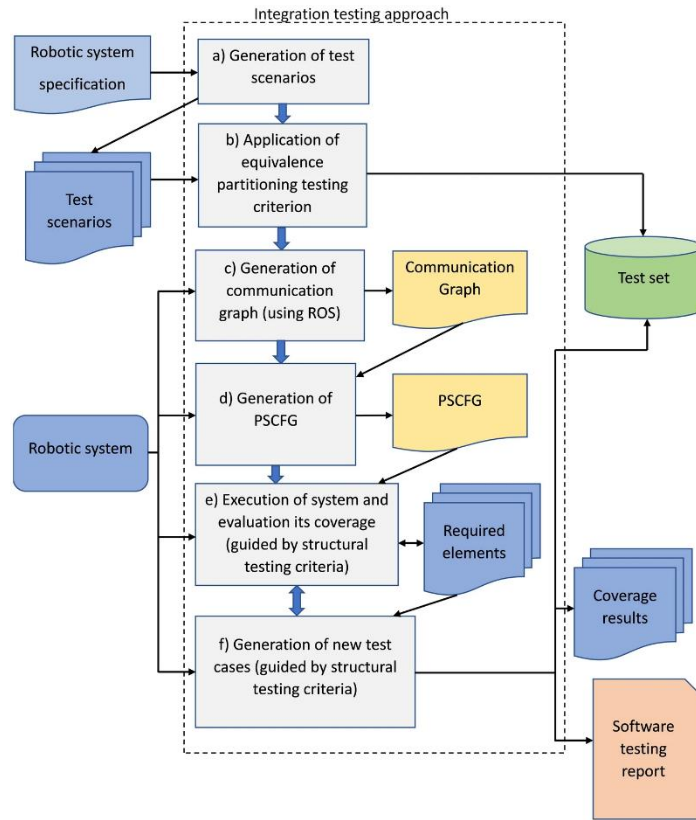


Figure 3: Testing framework used in [22]

In a further effort to optimize the **Integration testing** approach, Leung and White identify the common faults in combining modules as well as recommendations on test selection to minimize redundant test running [23]. Firstly, in regard to the most common errors, they are described as: interpretation errors (difference in actual specification of a module vs. what the user interprets), miscoded call error (call instruction to a submodule is listed at the wrong point in a program), and interface error (mismatch exists between interface standards). Secondly, in regard to principles for test selection strategy, they suggest several metrics and test selection strategies. One such strategy is the concept of a firewall which is defined as a limit on the modules which must be considered when its surrounding modules are modified. Doing so prioritizes testing of key parts of the module that are actually affected at runtime. Applying a similar strategy to test case development may lend itself to considerable time savings in the long run.

ROS itself also has an officially supported framework for **Integration testing** that leverages the pre-built libraries such as roslaunch and pytest. Though the official ROS documentation falls short in adequately describing its usage, the Autoware foundation has a well-documented support site on its own [24]. The specificity requirements of test case developments can be seen as both a pro and a con. For every test in the suite, changes must first be made to the *package.xml* and *CMakeLists.txt* files for listing dependencies. The main test file then requires the following be specified: relevant package imports, **Node** launching process, specific test cases. Because the

framework relies on existing test libraries such as pytest for Python, the issues of low level, and often tedious test case specification still remain. The advantage of this is a high degree of customization in terms of test setup.

In [25], Bihlmaier and Worn develop a system to gain introspection capabilities of **ROS** based systems. Though not explicitly intended for traditional integration tests, it is easy to see the use case for this. The system works based on the concept of a monitoring **Node** that compares output data against reference values for correctness. In addition, a Graphical User Interface (**GUI**) is developed to report on system status. In their paper, the introspection system is applied to the case of a robot with attached sensors to demonstrate its efficacy in a real world scenario where fail-safe measures can be triggered based on the monitoring node. A note left for future outlook is the ability to write checks that detect anomalies with a high degree of accuracy but do no impede on flexibility in future use (i.e. robust tests). As seen in the literature review section on **Unit testing**, this is a common area of concern for test frameworks.

2.3 Current System Testing Approaches

At the system test level, developers seek to gain a true understanding of their system as a whole. It is at this stage that even the tiniest of errors can manifest into disastrous results in the case of a real time scenario such as a vehicle with an activated automated feature. Thus, a development team's efforts at the previous unit and integration test level can have a major impact on their experience with system level testing.

Lou, Deng, Zheng et al. conducted a comprehensive study and literature review to analyze the gap between current **ADS** development practices and the actual needs of software engineers working on them based on experience [6]. Initially the researchers got a gauge of what the current common practices are in ADS testing, starting with the type of ADS that participants worked on. The majority responded to have worked with multi-module systems as opposed to end-to-end driving models (e.g. PilotNet [26]) and thus a conclusion was that multi-module ADS's "deserve more attention in future research". Another finding was that the use of both standard performance metrics as well as custom-tailored metrics is a desired requirement of testing. A key takeaway from this is that test tooling must be able to support both types of metrics for test evaluation. With regards to **System testing**, the study found valuable perspectives on it as well. From these insights it was clear that a proposed testing framework must be adaptable to both simulation and real-world scenarios, have the ability to evaluate system level performance, and that it would benefit greatly from ease of custom drive scenario integration for simulation. As part of [6], emerging needs of ADS testing were also identified. The need to speed up testing was mentioned in Section 1.2 as one of the major findings of the survey. The justification for this is evident upon the evaluation of the high reliability requirements for testing. As per the insights of ADS practitioners, though simulators just take one-tenth of the time compared to real-world testing, the high reliability requirements means that lots of testing mileage is needed. In fact, existing work suggests a catastrophic failure rate should be minimized to 10^{-7} to 10^{-9} for 1-10 hours of driving [27] [28]. Thus, even in simulation, assuring high reliability is a significant time investment. The study suggests areas such as test

selection and prioritization as future areas of improvement. Applying these lessons to the context of this thesis, it is evident that methods in reducing the time requirements of the ADS testing pipeline would have worthwhile benefits in assuring overall system reliability.

Lattarulo, Perez and Dendaluce present a validation methodology centered around the path planning and controls algorithms for automated vehicles in [29]. The proposed framework consists of all the submodules of an automated vehicle including modules for Human Machine Interface (HMI), communication, a vehicle dynamics model as well as Dynacar for visualization as well as sensor data generation. Dynacar was chosen as the simulation software because of the focus on high-fidelity vehicle physics. Additionally, Matlab-Simulink was chosen to implement the system model because of its wide reach in academia and its easy to use interface. When put together, it can be seen how a model based approach to algorithm design and testing looks like in Figure 4. The proposed testing architecture also benefits from integration with Hardware In the Loop (HIL) testbeds. Thus created scenarios can be run in simulation and then controller responses can be measured either through a physical or soft Electronic Control Unit (ECU). In addition, the focus of modularity in model based design allows for different blocks to be tested and replaced/modified as necessary for testing.

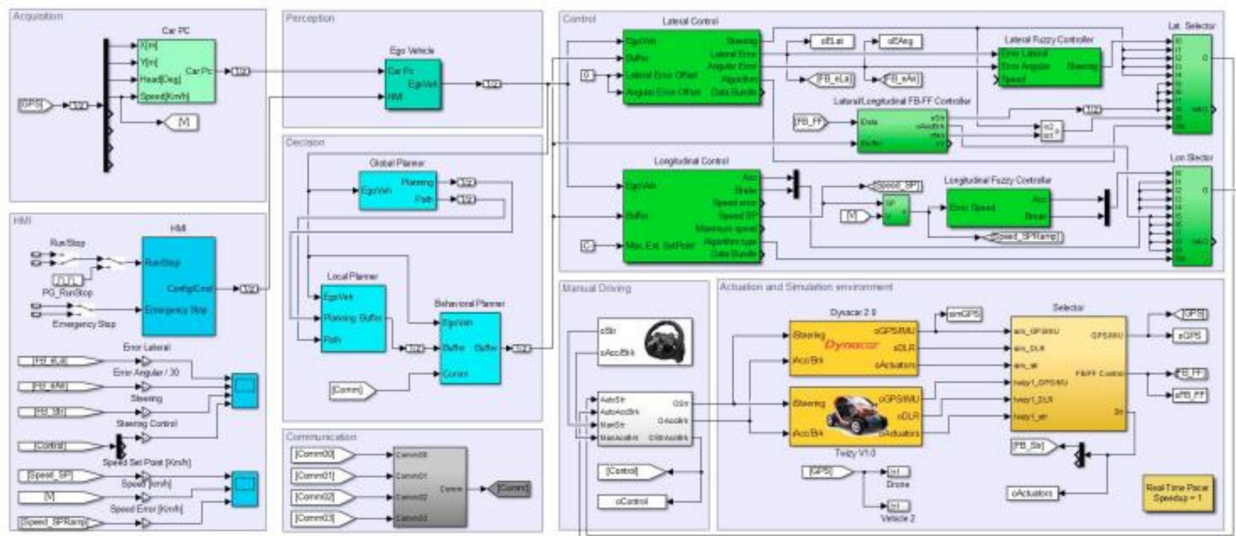


Figure 4: Model based design approach to ADAS algorithm design and testing in [29]

Though a comprehensive approach to testing, the lack of intermediate unit/integration test levels means a lot can go wrong in between. In addition, the test process is not trivial and/or quick to set up. A scenario must be run in Dynacar and then the inputs to the algorithm must be monitored as they make their way through the different blocks in the system. This approach thus seems inflexible to quick system or testing changes.

Tatar maximizes test coverage while minimizing test design time requirements using a software application known as TestWeaver [30]. As illustrated in Figure 5, Tatar incorporates TestWeaver in the center of a close loop between model outputs and inputs so that as many states in the state space can be reached as possible. The applications controls the specified parameters and based on predefined 'correct' values, automatically generates inputs to drive the system to as many

different states as possible. The intuition is that the more states that can be reached, the higher the likelihood of finding defects that were not previously caught. Additionally, because the 'test case' is automatically generated after initial system setup, time is saved in the long run. Though compatible with Simulink, it does not directly interface with **ROS** which is an issue for teams that use it in their algorithms. Furthermore, though the test inputs are automatically generated, this framework does require that the system as a whole is well defined in advance including the plant model, controllers and quality **Observer**s as noted in Figure 5. Thus, intermediate testing is again not trivial with this proposed test architecture.

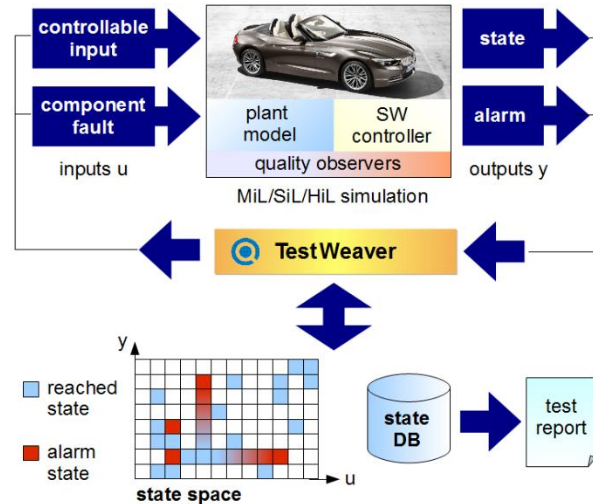


Figure 5: How TestWeaver fits into the system test architecture in [30]

In [31], Jiang proposes the Generalized RAcing Intelligence Competition (GRAIC) framework in which the **CARLA** (CAR Learning to Act) simulator and **ROS** is used in conjunction for **System testing**. Specifically, the focus of GRAIC is to provide ground truth detection results so the focus of development can be on purely algorithm development with the rest of the testing pipeline fully automated. This is done so by incorporating **CI** tools. The architecture of the GRAIC framework is illustrated in Figure 6. CARLA's Scenario Runner application is used to generated varying scenarios including different actors, vehicle dynamics and environments. This is then integrated with GRAIC such that ground truth results can be input to planning and controls nodes where simulator results are checked against correct values. Based on the results of previous scenarios, new ones can be created dynamically to detect as many failure states as possible.

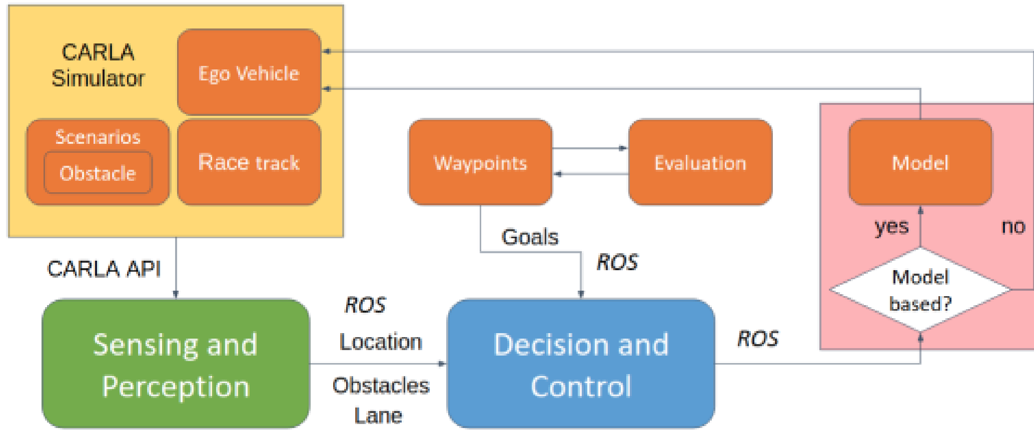


Figure 6: GRAIC architecture as shown in [31]

The shortcoming of the GRAIC framework is that it does not expand to perception module testing as these results are directly produced by the simulator. Instead, as mentioned, planning and controls is the focus.

Son et al. present yet another approach to **ADAS** development and validation using tools not mentioned in previous papers but ones that are still seen in industry [32]. The scope of test functionality is set to ADAS features including demonstrations of **ACC**, Vehicle to Infrastructure (V2I) communications and autonomous parking. LMS Amesim is a physics-based simulation platform that for the purposes of ADAS /**AV** testing allows for the simulation of vehicle dynamics in a model based design fashion. PreScan is another software that some in the automotive industry are familiar with because of its advanced visualization capabilities and intuitive **GUI**. Controls and planning development for the different features then come from a combination of Amesim as well as Matlab and Python. Putting the pipeline together allows for a closed loop testing environment in which drive scenarios seen in real world conditions are able to be simulated with a relatively high degree of accuracy. The conclusions from this methodology reveals benefits as well as areas for improvement. In terms of benefits, the development team noted that the high fidelity of the model, customizable scenarios and great visualization possibilities stood out. On the other hand the time consuming nature of the simulation was seen as the biggest drawback considering everything was done manually and in a linear fashion. As well, the different tools, environments and even languages seemed to add a layer of complexity and thus time intensiveness to the project.

In the aforementioned works, several tools are used for simulation and validation of ADAS /AV algorithms. With the many options for testing and the various use cases within the domain of automated vehicle development, it can be hard to choose the best toolset for the project at hand. Kaur et al. provide a description of the minimum criterion for a good simulator for **System testing** as well as a comparison of the commonly used simulators today [33]. The functional requirements listed for a good automotive simulator are that it is able to effectively demonstrate/showcase: perception capabilities, multi-view geometry (e.g. for SLAM applications), path planning and controls support (e.g. controller tuning), a 3D virtual environment, traffic infrastructure / scenario simulation, and ground truth. In additional non-

functional requirements, as sought after in most software applications, include good stability/maintenance, modularity (such as through a flexible API), portability, scalability and open-source (preferred). The paper compares MATLAB/Simulink, CarSim, PreScan, **CARLA**, Gazebo, and LGSVL. Ultimately several key observations are made. For instance LGSVL and CARLA are described as most suited to E2E testing because of the comprehensive support for automated features and modules such as perception, mapping etc. Software such as MATLAB/Simulink is described as a top choice for *upper-level* algorithms based on model based design.

2.4 Chapter Summary

For each of the testing levels discussed, a summary of the key findings are listed:

Unit Testing

- There is a general lack of clarity among developers on what defines a high quality test [5]
- Tool support for unit test development is highly sought after [5].
- A **TDD** approach has long term benefits in product and development process quality at the expense of increased short term effort and time expenditure [10] [11] [12].
- Similar to TDD, test automation has a significant potential upside for long term time savings at the cost of higher short term pipeline development effort [17].

Integration Testing

- Model based testing approaches such as in [18] provide great scalability and comprehensive testing at the expense of high technical complexity.
- Testing frameworks centered around **ROS** such as in [22] benefit from evaluation of publishers/subscribers as the most common failure point.
- The standard ROS **Integration testing** framework, though highly customizable is also extremely tedious to develop around.

System Testing

- A survey of **ADS** software engineers revealed major shortcomings in system test runtime, scenario selection/prioritization, and the importance of custom metric definition [6].
- Many options to test systems in software exist such as in [29] and [30] but closed source software tools are a barrier to adoption.
- Other system test setups often suffer from high technical complexity and time investment for implementation and/or test running [32].

3 Test Framework Architecture

This section proposes the test framework that was developed to address the challenges described in Section 2. At each level of testing (unit, integration, and system), the proposed contribution is explained in detail, including a justification of the design decisions that led to its development.

3.1 Unit Testing Guide

Being the lowest level in the test framework, **Unit testing** is extremely application and implementation specific. For example, consider the module for state estimation in two separate **ADAS** software algorithms, where algorithm A chooses to implement a simple Kalman filter to realize this module and algorithm B uses a particle filter. Unit tests would then test sub functions of these algorithms themselves. In the case of A, perhaps tests would cover a specific case of the prediction step and verify that the covariance matrices output results within a reasonable tolerance. In the case of the particle filter on the other hand, we may instead want to test that samples are generated correctly in the initial sampling step. Due to the innate differences in algorithm definition, it is thus clear just how wide the variance of unit tests may be if they were incorporated by two different development groups. For this reason, this section will provide a more generalized list of guiding principles for unit test development as opposed to concrete examples of test cases themselves in the **ADAS** domain. The lessons learned from the literature review in Section 2.1: *Unit Testing and Automation: Current Practices*, as well as an additional online literature review of ten additional sources were used to compile the six guiding principles for unit test development described below. They are listed below in order from least to most common frequency of mention among the sources consulted.

Recommendation #1: Consistent naming convention

Mention of proper naming convention was referred to in four of the sources [34] [35] [36] [37]. Though a seemingly trivial consideration in the development of a suite of tests, a unified convention serves to minimize barriers in communication within a team. Tests are often written and maintained by multiple developers, and consistent naming conventions make it easier for them to grasp the purpose and behavior of each test without having to dig into the implementation details. A well-defined naming convention also aids in organizing tests into logical groups. By following a consistent naming pattern, developers can categorize tests based on functionality, modules, or classes. This organization simplifies the process of locating specific tests and helps identify any gaps in test coverage. [34] recommends the convention of *Method_Scenario_ExpectedBehavior*. In addition to improving clarity for the originators of the test suite, future readers of the code will benefit from increased readability. [36] even suggests that the behavior of the code should be able to be interpreted without ever having to look at the code itself, i.e., functionality should be evident from the test name.

Recommendation #2: Code clarity and documentation

In a slightly different but related vein of overall test development clarity, five of the sources referred to the importance of enhanced organization and/or readability of the test code itself as well as made the case for clear documentation of its purpose [34] [35] [37] [38] [38]. Clear organization promotes scalability and helps to more easily facilitate modifications to the test

suite in the future. [36] leverages the Arrange-Act-Assert framework to maximize standardization and thus readability between tests. That is, plan the tests initialization, call an action, and then verify the outcome. The resulting high degree of readability allows development teams to quickly understand the purpose of tests without having to dive too deeply into the specifics of its implementation. It may also serve to prevent redundant copies of tests that are known to come up when communication within the development team is limited. Documentation is of importance for teams that are new and/or that may have to deal with a high degree of knowledge transfer and onboarding. The learning curve for new developers is reduced and thus simplifies the work of future developers who may have to modify tests that they didn't initially set up themselves.

Recommendation #3: Avoid test interdependence

Another prevalent theme in the source material was the proper isolation of tests, which can be increasingly challenging as the test suite, and the algorithm that it supports increases in size and complexity. Isolation in this case refers to the practice of isolating units under test in a way that prevents interference from external dependencies. As mentioned in six different sources, the ability to isolate tests properly is a significant indicator of good test quality [34] [35] [39] [38] [36]. This means that tests should not interfere with the functionality of others and each should be responsible for its own setup/cleanup processes. As stated in [35], no real-world or external dependencies should affect the outcome of a given test. The very practice of unit tests is said to help with code decoupling itself. By designing and planning test suites for specific sections of your codebase, it encourages development teams to modularize their algorithms functions into manageable subsections. Additionally, when it comes time to automate test processes, **Test runner**s tend to run multiple test simultaneously without attention to a specific order. Thus reliance on the outcomes of other tests or sections of the codebase undergoing test may lead to failures in the suite. Testing in parallel enables faster cycle tests of test execution and may at the worst case, point out potential concurrency or thread-safety issues.

Recommendation #4: Minimize tests, maximize coverage

The fine balance between minimizing test while maximizing coverage is a central challenge in any approach to test case development, but especially at the unit test level when specificity is high but so are the potential costs of maintenance/test running. Seven sources brought up this challenge in one form or another [12] [35] [36] [37] [40] [41]. A helpful insight when determining the scope of a test is to focus on a single use case for each test. However, this should not detract from the goal of maximal coverage given current resources. Of course, 100% coverage is ideal target, but development groups have to decide for themselves what the tradeoff will be given the increased time and resources required for additional development. A common shortcoming of test development is doing so without a firm plan in place. [40] described the benefit of proper planning and even a teamwide known test strategy before undergoing the testing phase. The paper brings up the following two questions as points of consideration whenever a test is written: 1 - "What does it mean when the test passes?" and 2 – "what does it mean when it fails?" Failure to answer both thus implies improper understanding/planning of the test at hand. Khorikov's book *Unit Testing: Principles, Practices and Patterns* succinctly covers this theme with the characterization of a good unit test suite as amoung other things, one that

provides maximum value with minimum maintenance costs [41]. When there are less tests to update/modify, developers are less burdened with having to scour the codebase for a specific test. The importance of this is further justified when analyzing the learnings of developer experience such as those found in [5]. Data shows that failing tests are often treated with fixes or deletion of the tests rather than fixes to the code itself so the ability to keep a well maintained and organized test suite is of high importance. On the note of code coverage, learnings from a Microsoft case study showed the benefit of having an automated but single tool for code coverage across the team [12]. As well, incorporating the coverage as a formal metric that is visible teamwide was seen as critical takeaway to team buy-in of unit test practices.

Recommendation #5: Test driven development

TDD [42] is an iterative approach to development which focuses on writing tests *before* writing the actual production code. In other words, programmers using the approach would not write a new function until a text exists that would fail because of the lack of this said function. The general cycle is visualized in Figure 7. The goal is seen by some as focusing on feature specification as opposed to validation, implying it is more of an agile requirements technique [43]. This is because writing a test means that the developer / development team has already defined the pass criterion that the added functionality is supposed to meet. An alternative viewpoint is that TDD is in itself a programming technique with the goal being clean working code [44]. Irrespective of viewpoint however, the majority of sources consulted in this literature review recommended it as a key consideration to incorporating unit test practices [12] [34] [35] [36] [41] [45]. [34] claims the benefits of TDD include more readable code (due to regular refactoring), easier maintainable codebase (consequence of smoother and more iterative design process) and better organized dependencies. As evaluated in the Microsoft case study, these improvements were quantified by defect comparison in development groups using a TDD approach to ones that were not [12]. The TDD group's product had 60-90% fewer defects as well as an increase in test coverage. By nature of its design process, TDD also improves regression prevention as existing functionality must pass before the failure status of any new tests can be evaluated. Thus, unintended side effects or regressions in previously working code are tackled earlier. A criticism of the method is that longer initial development timeline than traditional approaches. However, TDD enables faster development iterations because smaller units are focused on at a time.

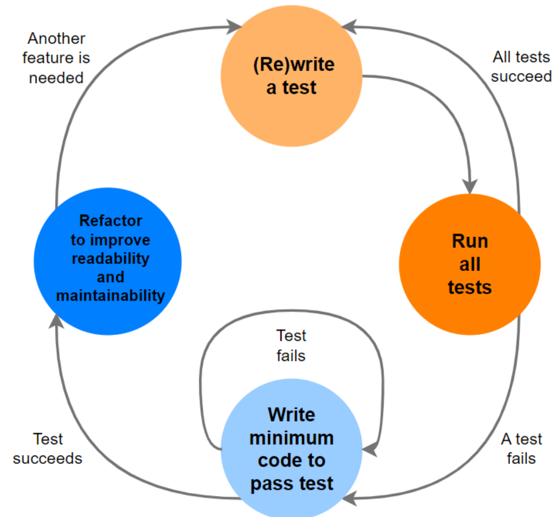


Figure 7: TDD process as depicted in [35]

Recommendation #6: Automated/timely

The consideration of timeliness of a unit test pipeline or its outright automation was the most prevalent recommendation across the consulted sources [12] [35] [36] [37] [38] [39] [40]. Manual testing is both extremely time intensive and potentially less reliable in the scheme of an entire suite of tests covering a development project. Thus, an automated version makes clear sense in many scenarios. [40] defines completion of automation as test suite integration that requires no human input, allows for tests that are consistently executable, does not have dependencies, and that can run repeatably in all supported configurations. Timeliness in the sense of writing test code is recommended to be kept in mind as well. As recommended in [39], the length of time taken to write the test should not be disproportionately higher than the time taken to write the code being tested. A situation where this does arrive is a reason to reconsider the test design. This very lesson was explicitly stated as a learning in Microsoft's unit test case study [12]. In addition, an automated **Unit testing** pipeline allows for rapid feedback with little effort. This in turn reduces the cost and effort associated with fixing bugs in later stages of development. Preventing regressions is also much easier if the codebase is being verified with a higher frequency. Ultimately timeliness / automation is largely seen as a required characteristic of any major development project. For the developers it reduces time and effort, and for the customers it produces higher quality software products.

Though not listed as specific recommendations in the testing guide, other less common but still useful recommendations for proper unit test design include:

Deterministic tests – Determinism in this context refers to the ability of the code to provide consistent outputs across multiple different runs. A lack of determinism makes it challenging to separate genuine test failures from false positives or intermittent issues. This is another benefit of having a continuous integration pipeline. If certain runs fail non deterministically, the test suite has not been designed in an optimal fashion.

Avoiding logic – Avoiding logic in **Unit testing** allows developers to focus on the unit's behavior and not the test, thereby enhancing readability and maintainability of the test suite. The inclusion of logic into the code increases the likelihood of rework for developers who would then have to make sure the logic in test cases keeps up with a dynamic codebase.

3.2 University of Waterloo Structured Testing Framework (UW-STF)

The unit test guide suffices for lower level testing which is too specific to benefit from a structured framework in many applications. However, moving to integration and **System testing** in **ROS** presents the opportunity to streamline the process with a flexible framework that can be used in many applications and software platforms. This is the central theme of design for **UW-STF**.

3.2.1 Design Intent and High Level Functionality

UW-STF focuses on addressing some of the shortcomings of current testing approaches that were discussed in detail in Section 2.2 and 2.3. Specifically UW-STF was developed with the following design intents:

1. An intuitive and simple to use interface for test specification.
2. Integration with existing **CI** tools for automated test execution of the entire test suite.
3. Adaptable to integration and system level testing.
4. Adaptable to different software platforms and accessible for any development team through a focus on open source software.

At a high level, UW-STF is responsible for spinning up **ROS** nodes, monitoring the desired **Topic**s from these nodes, and then evaluating the incoming data for correctness based on predefined validity criteria. It also handles setup/cleanup operations of the nodes without any further specification. Once the nodes have been spun up and are ready to receive data, prerecorded **ROS** data is played back using the specified **Bag file** which serves as the input to the system under test. When the test is complete, a results file is generated that lists the system performance for each test as well as pass/fail status. When connected to a **CI** tool, the overall system status is determined by these individual pass/fail status results.

3.2.2 Detailed Design

An architecture diagram for **UW-STF** is presented in Figure 8. In this section we cover specific implementation details for each of the major components underlying UW-STF including perspectives on how they address some of the major challenges of current integration test practices raised in the literature review section. The descriptions are based on internal team documentation originally written by **UWAFT** team member Ansar Khan and then updated as necessary by the author of this thesis.

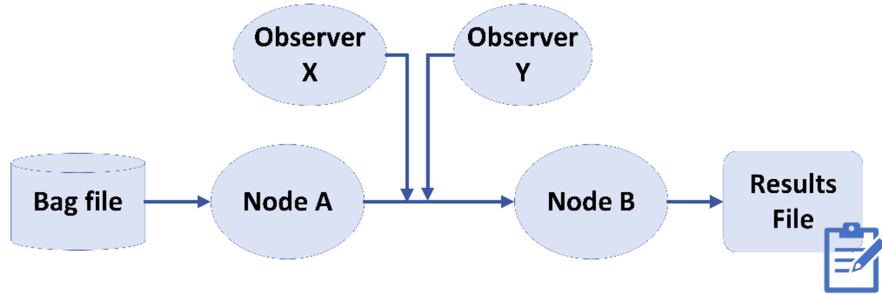


Figure 8: UW-STF architecture diagram for Nodes A & B under test

1. *.sim file*

The **.sim file** is central to the operation of **UW-STF** as it defines all important high level parameters for test execution. A picture of a sample *.sim* file is shown in Figure 9. This file includes the following fields to be input by the user:

a. *Launch files*: location of launch files specified in conventional ROS2 format using the preexisting *roslaunch* tool to spin up nodes [46]. The user is responsible for correctly calling out the nodes to be run during test execution. However, failure to correctly populate the **Node** graph with the necessary dependencies will be raised as an error by UW-STF such that tests will not continue through till execution.

b. *Published* **Topic***s*: name of topics that are published by the nodes under test and that will be monitored by the **Observer** nodes or *Observers*

c. *Required topics*: topics to publish from the bag file that serve as inputs to the nodes under test. They can be played directly using the rosbag tool.

d. ***Bag file***: path to bag file containing prerecorded **ROS** topic data which serves as the input to the software stack. In the context of **ADAS** testing, this is often sensor data. Bag files are ideal because they can reflect either simulated or real-world data.

e. *Test duration*: test timeout value in seconds. After this time, the test will be stopped and final pass/fail criterion is evaluated for each of the Observer nodes

f. ***Observer**:* custom implemented node classes that monitor data published from nodes under test against validity criterion. Observers are described in greater detail below.

my_test_example.sim 1.19 KiB

```
launch_files:
  - package: structured_testing
    file: my_test_example.launch

published_topics:
  - /filtered_obj

required_topics:
  - /associated_object

bag_files:
  - /int_test_bag_1/int_test_bag_1.db3

test_duration:
  - 30

observers:
  - name: LeadVehicleInWindow
    observerClass: InRangeObserver
    topic: "/filtered_obj"
    msgType: "common.msg.FilteredObjectMsg"
    field: "obj_dx"
    observerType: ALWAYS_TRUE
    minVal: 0
    maxVal: 85

  - name: LeadVehicleFurtherThan
    observerClass: MinMaxObserver
    observerType: ALWAYS_TRUE
    topic: "/filtered_obj"
    msgType: "common.msg.FilteredObjectMsg"
    field: "obj_dx"
    value: 90 #in m
    isMin: False
```

Figure 9: Contents of .sim file used in UW-STF

2. Observers

Observers are custom implemented Python classes that extend the Node() class and define how a specific Observer processes and evaluates incoming data. Though custom implemented Observers require similar time commitments as conventional **Integration testing** practices, the aim with created Observer classes is that they can be added to an open repository or database of Observers that are reusable. As seen at the bottom of Figure 9, Observers require several common fields. Current Observer templates are listed in Appendix A. Other fields not shared between them are specific to the purpose of that Observer and should be documented in the associated Observers usage documentation. The common fields are as follows:

a. *name:* name of the Observer that is used to associate the specified Observer with its corresponding results file entry.
b. *observerClass:* Python class defining the functionality of the Observer.
c. *oberverType:* Type of Observer must be one of {ALWAYS_TRUE, TRUE_ONCE, TRUE_AT_END, TRUE_AT_START}. Note that this field is not necessarily required for all Observers.
d. ***Topic****:* Name of topic that Observer subscribers to. Depending on the functionality of the subscriber multi topic inputs may be required (e.g. comparing ground truth topic to output of perception module)

e. *msgType:* Data type of the message in the same format that it would be imported in a Python script (e.g. std_msgs.msg.Float32).
f. *field:* Subfield of the message that is being 'consumed' by the Observer. It is indexed through a dot operator.

The **Observer** framework is designed to support rapid development of new custom Observers targeted to specific use cases. After creating the class in the Observers directory of the structured testing package, the class must inherit the parent BaseObserver class which does not require any user modification itself. The BaseObserver provides functionality common to all Observers including methods to start up the Observer nodes and subscribers, create single or multi **Topic** callback functions, consume and log topic data, and print the results to a results file when test execution is complete. Any of these methods may be overridden as necessary in the specific Observer class. New Observer development also requires the inclusion of a few main methods to properly integrate it with **UW-STF**.

An **initializer** should take following form, ensuring that the super initializer is called.

```
def __init__(self, *, name, topic, msgType, observerType, customArg1, customArg2,
...        ,  **kwargs):
        super().__init__(name, [topic], [msgType], observerType)
```

A **consumeMsg** function is called every time a msg is published with the purpose of parsing the data and then evaluating it against a correctness criterion if necessary within the function. Note that **Observer**s can be classified as memoryless (i.e. its truth value is able to be determined with messages at current time step without memory of previous values) or not. If memoryless, the getResult function will return an overall truth value at a time that is determined by the specified Observer type: ALWAYS_TRUE, TRUE_ONCE, TRUE_AT_END, TRUE_AT_STAR.

A **getResult** function is called once at the end of the simulation and is expected to return a boolean that corresponds to a pass/fail of the test.

Lastly, the **metaDict** function is called once at the end of the simulation and is expected to return a dictionary which contains any contents that the user wants to output to the results file beyond just the pass/fail status. This is most commonly parameters used specific to a certain test. For example when using the In Range **Observer** which checks if the **Topic** data is within a certain range, the metaDict should contain the minimum and maximum values used to define a valid range.

3. *.simresults* file and CI integration

The **.simresults file** displays the results of all **Observer**s specified in its corresponding **.sim file**, and is automatically produced as an output of **UW-STF** following completion of the test(s). Figure 10 shows the results of the tests executed from the .sim file in Figure 9. It is evident that fields like name, status and observerClass are common to every Observer but other fields are specific to that Observer itself.

my_test_example.simresults 824 bytes

```yaml
- name: LeadVehicleInWindow
  status: PASS
  observerClass: InRangeObserver
  topic(s):
  - /filtered_obj
  field: obj_dx
  observerType: ALWAYS_TRUE
  minValue: 0.0
  maxValue: 85.0
- name: LeadVehicleFurtherThan
  status: PASS
  observerClass: MinMaxObserver
  topic(s):
  - /filtered_obj
  field: obj_dx
  observerType: ALWAYS_TRUE
  isMin: false
  value: 90.0
- name: NodeIsAlive
  status: PASS
  observerClass: HeartbeatObserver
  topic(s):
  - /filtered_obj
  observerType: TRUE_ONCE
- name: LeadVehicleCloserThan
  status: PASS
  observerClass: MinMaxObserver
  topic(s):
  - /filtered_obj
  field: obj_dx
  observerType: ALWAYS_TRUE
  isMin: false
  value: 120.0
```

Figure 10: Sample .simresults file associated with the .sim file shown in Figure 9

Following the creation of the **.simresults file**, a seperate *check_results.py* script parses the results for its pass or fail status and returns the appropriate exit code. Thus integrating UW-STF into a **CI** tool such as Gitlab simply requires the addition of the following lines to the .gitlab-ci.yml file.

For a certain stage just add the command to run the start_sim.py file with the **.sim file** as the command line argument:

```
python3 path/to/start_sim.py path/to/my_test_example.sim
```

And then with the newly generated *.simresults* file as the command line argument, the *check_results.py* script can be run:

```
python3 path/to/check_results.py path/to/my_system_test.simresults
```

3.2.3 Addressing Design Intents

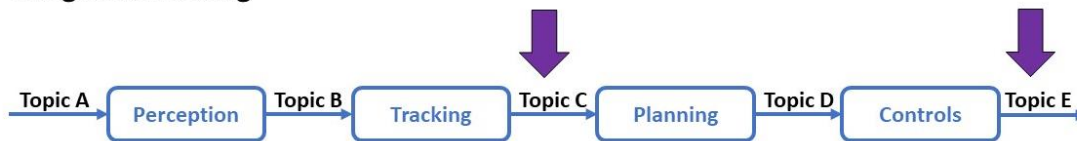
Now that low level design aspects have been described in detail, a preliminary analysis of how the framework meets the design intents listed in Section 3.2.1 can be addressed. Firstly in regard to point #1 – an intuitive and simple to use interface, YAML was selected as the language for **.sim file** specification because of its human readable input. Parameters are modified much easier and more intuitively than in conventional programming language test development. In addition, it has strict syntax (which is better for robustness) and matches data structures native to Python

for parsing. On the development side, it is also extremely version control friendly since it is purely test based as well as being open to comments. Though the rest of **UW-STF** is developed in Python, users that are not test developers can still easily interpret the tests themselves as YAML is used without requiring a specific operating system or programming language.

Point #2 (integration with **CI** tools for automated test execution) is addressed in the last part of Section 3.2.2. Specifically, the *check_results.py* script is able to parse the results file and return an error code that is compatible with the standard convention for CI pipelines.

Point #3 (UW-STF's adaptability to both integration and system levels of testing) is possible by nature of its evaluation of the interfaces of a **Node** (**Topic**s) in the system. A demonstration of this concept is shown in Figure 11 in which dependent on the topics that are evaluated the scope of testing changes. **Integration testing** involves testing of a subset of the modules in isolation (e.g. planning and controls nodes). **System testing** on the other hand involves the whole system and so must encapsulate all its modules. Thus the input and output topics of the entire system are monitored.

Integration Testing



System Testing

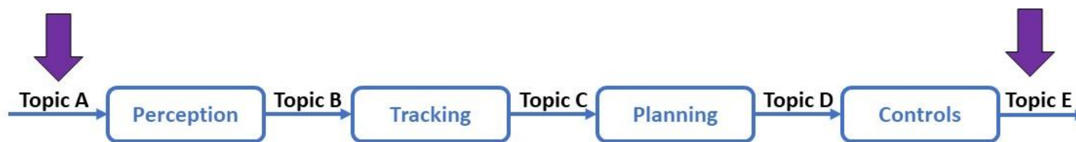


Figure 11: Given an ADAS system with several modules (ROS nodes), the scope of testing may be changed by selecting the appropriate topics. Integration testing (above) and system testing (below) is shown as defined for the following pictured system.

Lastly, point #4 (adaptable to different software platforms and open source accessibility) is evident given the tools and software used for its implementation. The framework itself is built around testing of **ROS** nodes which itself is an open-source middleware. Python and YAML are also freely available languages used in the execution and development of tests. UW-STF requires no other external dependencies or paid software subscriptions to carry out tests. This means there are no barriers to its adoption in either industry or academia.

3.3 Generating Simulation Data

Beyond tool support for testing, the review of literature in earlier sections revealed a major requirement for **ADAS** testing teams was to speed up the simulation process [6] [32] . Additionally, Williams et al. determined a key takeaway for adoption of a formal testing framework was to keep execution of the tests as simple as possible [12]. To address these challenges, a pipeline for sensor data generation that serves as input to **UW-STF** is proposed including a selection suggestion for the source of **ADAS** driving scenarios.

Two areas of potential pipeline speed improvements identified in [6] are test selection and prioritization. Fortunately however, the National Highway Traffic Safety Administration already provides a comprehensive report on the most common pre-crash scenario typology [47]. As stated in their report, the portrayal of frequency and type of crash scenarios is done so with the intent to among others, enable researchers to prioritize crash typologies, devise appropriate countermeasures and specify automated vehicle control capabilities that can assist drivers in preventing these types of crashes in the future.

3.3.1 Choice of Simulator

The subteam within **UWAFT** responsible for software development related to vehicle automation is known as the Connected and Automated Vehicle (**CAV**) subteam and so the selected simulator had to be suitable for their needs. **CARLA** is selected for the criterion described below, but UW-STF is not limited to just this choice. The only requirement for a simulator is that its data can be converted to **Bag file** format. From UWAFT's internal training documentation, "UWAFT has chosen to use the open-source, feature-rich, high-detail offering of CARLA for its perception simulation environment. Coupled with extensive open assets created for the purpose of **CAV** software development, full control of custom environments, traffic, and weather, and a deep, flexible sensor suite, this program allows for a high degree of customization with high fidelity graphics. In turn, this will benefit the team by allowing for increasingly complex test scenarios as the perception algorithm evolves over the course of the competition. CARLA also provides a seamless connection to **ROS** via its custom ROS bridge" [48]. Research from Kaur et al. further justify this decision with their comparison to other simulators by stating CARLA was one of the most suitable for end to end testing because of its comprehensive support for automated features [33]. Of particular usefulness when compared to other simulator options is CARLA's customizable suite of actors, environments and especially sensors as seen in Figure 12. Beyond the provided list of configurable sensors such as lidars, cameras, and radars, new sensors can be created from scratch. As will be explained in Section 4, this feature was a necessity in the **EVC** because of the competition provided vehicle's stock sensors and their unique output of preprocessed detections to the Controller Area Network (**CAN**) bus.

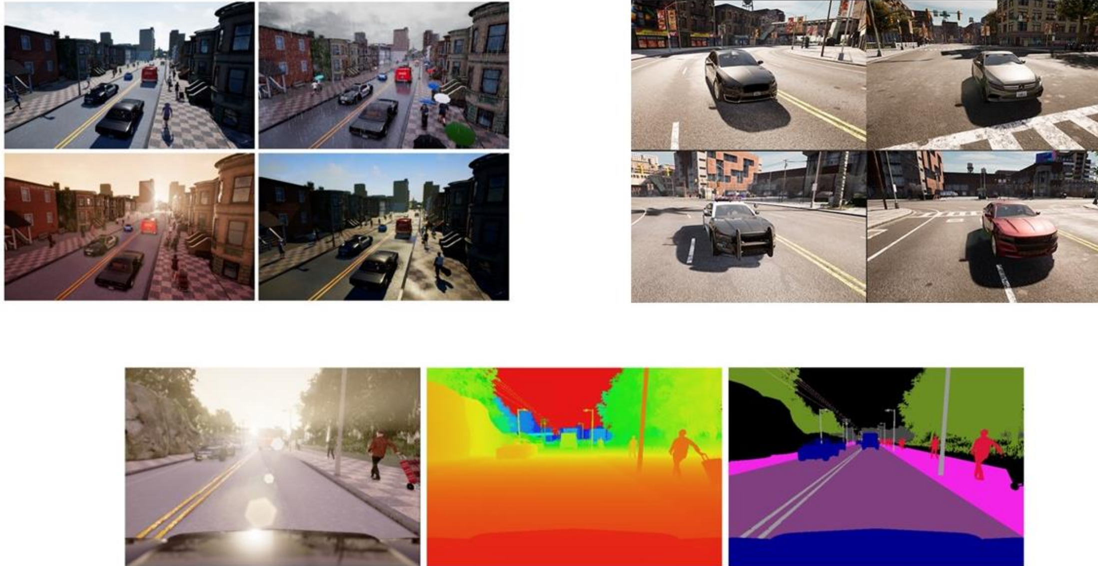


Figure 12: CARLA provides a flexible suite of environments (top left), vehicles (top right), and sensors (bottom) [49]

3.3.2 Proposed Simulation Pipeline

To comply with the requirements of **UW-STF** as well as address the issue of simulation timeliness as a testing pipeline bottleneck, the approach below is proposed:

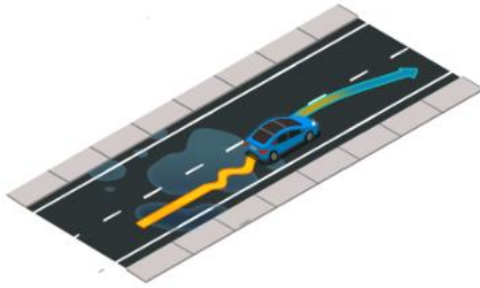
1. **Pick suite of highest priority scenarios from NHTSA pre-crash report and find/implement them in CARLA.**

For example if the **ADAS** development team was still in the preliminary stages of algorithm development and testing it'd make sense to choose scenarios in which the operational design domain was less complex than others such as in the case of two-vehicle light-vehicle crashes. In this case the team may consult some of the scenarios from the report, ordered based on highest relative frequency (Figure 13).

No.	Scenario	Frequency	Rel. Freq.
1	Lead Vehicle Stopped	792,000	20.46%
2	Vehicle(s) Turning at Non-Signalized Junctions	419,000	10.83%
3	Lead Vehicle Decelerating	347,000	8.96%
4	Vehicle(s) Changing Lanes – Same Direction	295,000	7.62%
5	Straight Crossing Paths at Non-Signalized Junctions	252,000	6.52%

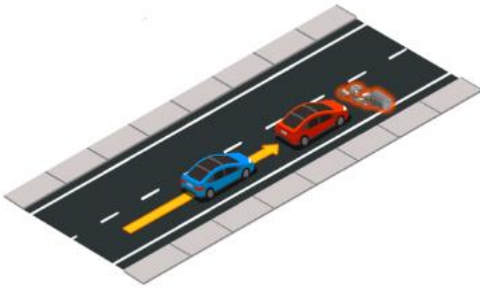
Figure 13: Top five most common two-vehicle light-vehicle crash scenarios as according to [47].

The scenarios can be implemented by the development team with the help of CARLA's dedicated traffic scenario definition module ScenarioRunner [50]. Alternatively, if the requirements for scenario selection are even less rigid, development teams can leverage the pre-built scenarios used for the CARLA Autonomous Driving Challenge which were also based off the **NHTSA** pre-crash report [51]. Some of these scenarios are visualized in Figure 14.



Traffic Scenario 01: Control loss without previous action

- **Definition**: Ego-vehicle loses control due to bad conditions on the road and it must recover, coming back to its original lane.



Traffic Scenario 02: Longitudinal control after leading vehicle's brake

- **Definition**: Leading vehicle decelerates suddenly due to an obstacle and ego-vehicle must react, performing an emergency brake or an avoidance maneuver.

Figure 14: Two of the several CARLA Autonomous Driving Challenge scenarios from [51]

2. Add sensors to ego vehicle including custom ground truth sensor

CARLA by default does not have a sensor capable of reading ground truth measurements of target objects. Thus, a custom developed Ground Truth sensor was developed for this purpose. It works by using CARLA's provided tick() function for sensors which records data for the sensor it is attached to at each simulated timestep. The ground truth sensor can be attached to the ego vehicle with a fixed mounting point similar to the default sensors. As seen in Figure 15, it is initialized using the sensor actor object, output path of its recorded data, the ego vehicle actor, and the target vehicle actors. Additionally, it inherits from a base Sensor class which has methods for recording longitudinal/lateral position, velocity, and acceleration measurements as well as heading angle.

```
120  class GroundTruth(Sensor):
121
122      def __init__(self, actor, output_path, vehicle, targets):
123          super().__init__(actor, output_path)
```

Figure 15: Ground truth sensors initialization function

3. Run scenario in CARLA and complete test by saving sensor data to Bag file.

Though listed here as one, this step has two parts to realize its execution. Leveraging CARLA's Python API, a single script is responsible for opening the CARLA world which acts as a server.

It also sets up the ego vehicle and other relevant actors in the scenario including configuring the sensors to record data. By the end of the simulation this data is saved in a bag file format.

4. **Run each Bag file scenario as input to UW-STF.**

With each scenario recorded as a separate bag file, it is now compatible with **UW-STF** as long as the corresponding **.sim file** is set up. When running system test level **Observers**, the ground truth **Topic** can be accessed as any other by specifying the topic name. UW-STF is capable of time-synching the ground truth and system output topics based on their timestamps. An illustration of the whole pipeline including gathering simulation inputs is shown in Figure 16.

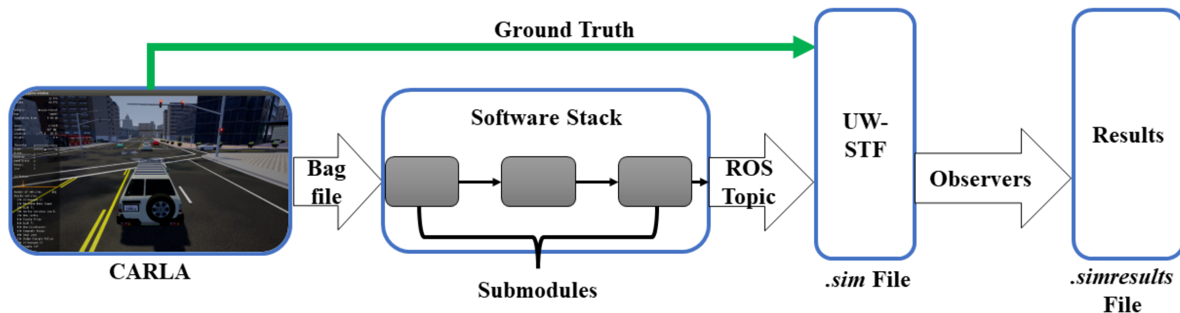


Figure 16: Illustration of the full testing pipeline including simulation input

As described previously, scenario setup and selection is done within the **CARLA** environment. The output of this is both ground truth data of the target objects as well as sensor data in which noise profile parameters have been accounted for. **Topic**s relating to both ground truth and sensor data are saved to a **Bag file**. From here onwards, UW-STF takes authority of the testing process by spinning up the required **ROS** nodes in the software **Stack**, the **Observer**s in the **.sim file**, as well as playing the bag file containing the simulation data. Once test execution is complete (either manually by a developer or automatically by a **CI** tool), relevant performance metrics are displayed in the **.simresults file**. If executed by a CI tool, the pass/fail status of these performance metrics determines the status of that testing pipeline. Because UW-STF handles cleanup of tests as well, multiple scenarios can be run in succession as long as they are entered in the desired order in the CI configuration file.

3.4 Chapter Summary

- Due to the extremely application specific nature of unit tests, a generalized guide (based on a review of literature) of six key recommendations to improve test quality is provided.
- At the integration and system test level, a custom test framework known as **UW-STF** is proposed to address shortcomings of previously mentioned frameworks. Advantages include a rapid and modular development approach, high flexibility in integration and an intuitive interface.
- Simulation data using CARLA is tied into the framework to show how UW-STF can function in a full scale **System testing** pipeline. The pipeline includes consideration of scenario prioritization and selection using **NHTSA**'s report on pre-crash typology.

4 Application of Framework to the UWAFT Stack

4.1 UWAFT Architecture

To validate the effectiveness of the proposed testing framework in a working development setting, this section utilizes the codebase developed by the **CAV** subteam of **UWAFT**. **EVC** requires the addition of **ADAS** features to a 2023 Cadillac Lyriq and at the time of writing, year 1 of the competition has been completed. Section 4.1 introduces the hardware and software **Stack** of the team while the following sections apply **UW-STF** to the stack at different levels of testing.

4.1.1 Software Architecture

For year 1 of **EVC**, each school's **CAV** subteam in the competition was tasked with developing a baseline perception architecture including considerations for both software and hardware implementation. Features that teams would have to implement by the end of the four-year competition include autonomous parking, V2X based intersection navigation, adaptive cruise control, and lane centering. The CAV team elected to go with an architecture design that focused on ease of implementation and debugging for the first year. Being the first year of competition, the skill levels of new recruits was largely at the beginner level. Thus, electing for design choices that members could more easily understand was a priority. Based on a review of the literature CAV members chose to modularize their design into the following to achieve **ADAS** L2/L3 functionality: perception, object tracking, planning, and controls.

Object Fusion: The perception module in Y1 was required to handle data from both stock Lyriq sensors, which output processed object detections over **CAN**, as well as conventional team added sensors, which output raw sensor data. Specifically, a team added front camera which outputs RGB images and overhead 32-channel LIDAR were proposed as key additions to the sensor suite to complement the existing stock sensors. Thus a full versioned perception module would have to include an object detection sub-module for the team added sensors. It would then be required to sync the data that was pre-processed with the output results of a custom object detection algorithm using a sensor fusion algorithm. For the initial version, or V0, of algorithm implementation which as mentioned, focused on simplicity of implementation and debugging, the scope of the perception algorithm was modified to exclude team added sensors. As a result, sensor fusion for V0 included fusing only the pre-processed object detections from stock sensors. Given the redefined scope, several traditional methods for sensor fusion were evaluated, from which a weighted-voting based approach was selected.

Tracking: Multiple variations of object trackers exist, from traditional cluster based approaches to probabilistic methods to newer machine learning based approaches. The full scale version of the object tracking module would be responsible for taking in fused sensor objects and being able to consistently associate them with a separate ground truth object, or track. Object tracking is also responsible for maintenance of the tracks in the case new objects are introduced into a scenario or if older objects are removed. Once updated, estimation filtering is usually conducted to provide an updated estimate of the state of the object(s). Lastly, traditional approaches to tracking include a gating step which helps to inform the bounds of assignment of new objects to

tracks in the next iteration. In the name of keeping the algorithm has easy to intuit for new developers in the team as possible, a nearest neighbours approach to object assignment was implemented. Track maintenance occurred through the update of an active status counter for each track, which was modifiable. State estimation was also implemented with the simplest of approaches found in literature – the vanilla Kalman Filter.

Planning: The planning module was defined by the **CAV** team to include behavioural and motion planning. Behavioural planning was set to include high level state actions that the ego vehicle should take given its current trajectory as well as the trajectories of all obstacle tracks provided by the object tracking module. Then motion planning could account for vehicle dynamics limits, collision avoidance, and drive quality, to make a decision on the optimal trajectory given a list of possible options. Behavioural planning via a Finite State Machine (FSM) approach was selected for V0 because of its ability to be modularly expanded to different feature requirements as well as its inherent transparency of current status. The 2019 Hyundai Autonomous Vehicle Competition, a project with a scope similar to EcoCAR, further justified this FSM based strategy [52]. The article incorporated such an approach for behavior planning of connected intersections in urban environments.

Controls: After lessons learned from previous implementation efforts in the previous EcoCAR competition as well as consultation with professors working on the subject matter, the team elected to go with a Model Predictive Controller (MPC) to realize the conversion of a target state from the planning **Node** to control signals to send to the propulsion controller. Similar to [52], the control of vehicle dynamics was decided to be split into decoupled controllers for longitudinal and lateral motion. A decoupled system benefits from being easier to debug and tune in different dynamic driving scenarios, with the drawback being a potentially less optimal solution than a single controller. Beyond the control module's outputs of acceleration, braking, and steering torque signals, the propulsion compute unit handles conversion of these to actuate vehicle dynamics. However, this is deemed to be out of scope for this thesis.

A complete software architecture diagram including the sensors, main **ROS** modules, as well as high level signals between them is shown in Appendix B. However, due to the fact that the planning and controls modules are yet to be integrated with the rest of the system, the scope of the examples in this thesis will just be limited to perception and tracking modules, which herein define our "system" under test. Furthermore, as integration tests validate a subset of nodes in the system, the system is further broken down into submodules for clarity of communication about where we are in the system with respect to the test (Figure 17). This will be of use in Section 4.3 when integration test examples are described at different points in the system.

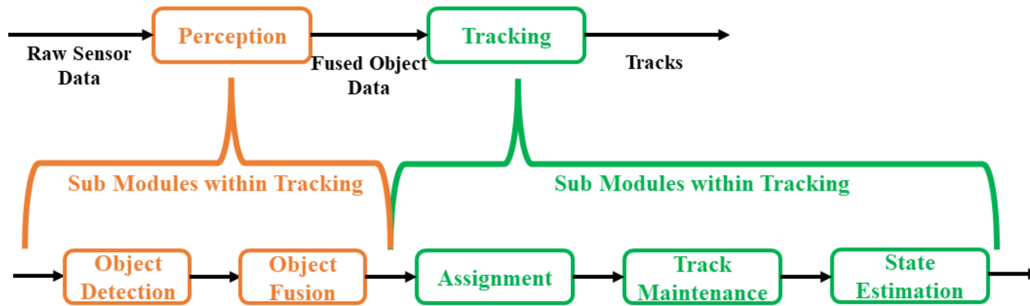


Figure 17: Definition of UWAFT system used in Section 4 examples

4.1.2 Hardware Architecture

To support the current and future algorithms running in vehicle, the **CAV** team also had to make decisions on sensor and compute hardware that would persist throughout the remaining years of the competition. In the way of sensor types the team chose the sensor suite presented in Table 1. Note that exact range and Field of View (FOV) specifications of the stock sensors are not shown due to General Motor's confidentiality requirements:

Table 1: Sensor Detail Overview

Sensor Type	Quantity	Stock or Team-Added	Range & Horizontal Field of View (FOV)
Front Long Range Radar (**LRR**)	1	Stock	Long range, narrow FOV
Front Short Range Radar (**SRR**)	2	Stock	Short range, wide FOV
Rear Short Range Radar (SRR)	3	Stock	Short range, wide FOV
Front Camera Module (**FCM**)	1	Stock	Medium range, medium FOV
Ouster LiDAR (OS1-32)	1	Team-Added	360° @90m
Pi Camera Module 3 Wide	1	Team-Added	120°

The additional LiDAR gives the team the advantage of a unique 360° field of view and higher-resolution detection that can be leveraged in obstacle dense environments such as heavy traffic and parking scenarios. The selected model also has a resolution matching that of large-scale online datasets (e.g. nuScenes), which make testing object detection feasible in simulation. Another team added sensor of high relevance to the perception algorithm is the Pi camera which provides raw RGB images instead of pre-processed detections, enabling team members to test machine learning (ML) based approaches to object detection. For the first iteration of algorithm development however, the team chose to focus on just the stock sensor inputs as they would not require an additional object detection step in the software **Stack**. The combined FOV is visualized in Figure 18.

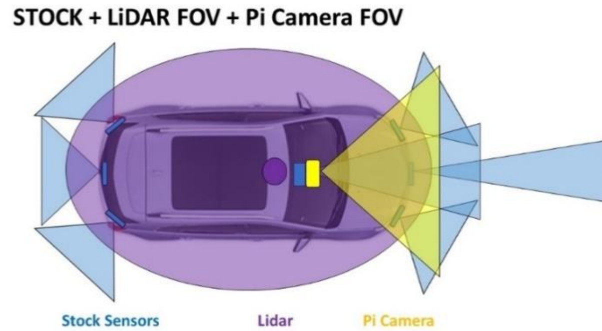


Figure 18: FOV depiction of stock sensors plus team added LiDAR and camera. Note that FOV and range are not to scale.

Beyond the sensor hardware which provides the inputs to the algorithm, the compute units required to run the algorithms themselves were also a decision for the **CAV** team to make. The network architecture chosen resembles a more centralized design with the dSPACE Autera taking on most of the computational load. Its extendable RAM and GPU capabilities allow the team to balance power, timeliness, and energy efficiency as needed. In addition, the team plans to add a Nvidia A5000 GPU to the AUTERA to further enhance its compute performance. The NavQPlus and Jetson were chosen as the main edge devices for work offloading. In addition to its competition sponsorship, the minimal footprint of the NAVQPlus allows for simpler offloading of tasks from the AUTERA with minimal increases in energy expenditure. Lastly, the Jetson TX2 was chosen because of the team's familiarity and experience with it in past competitions in which it has shown that it is capable of handling the power and timeliness requirements for data visualization. The proposed network diagram for the teams compute architecture is shown in Appendix C.

4.2 Unit Testing

In this section the recommendations from Section 3.1 *Unit Testing Guide* are applied to the **UWAFT** codebase to improve the quality of testing at this level.

First, we apply recommendation #1: consistent naming convention. Prior to the application of this guideline, no naming convention for tests existed on the team. The only guideline developers were told to follow is that tests for a specific package must be kept in that packages 'test' folder. This led to a quickly unorganized collection of unit tests that, though all related to a specific module, had relationships to the methods under test that were unclear. As seen in Figure 19, the relationship between tests and their corresponding source code scripts within the *obj_tracking* package is ambiguous as all the tests for the package are continually added to the same file in the order of when that test was conceived.

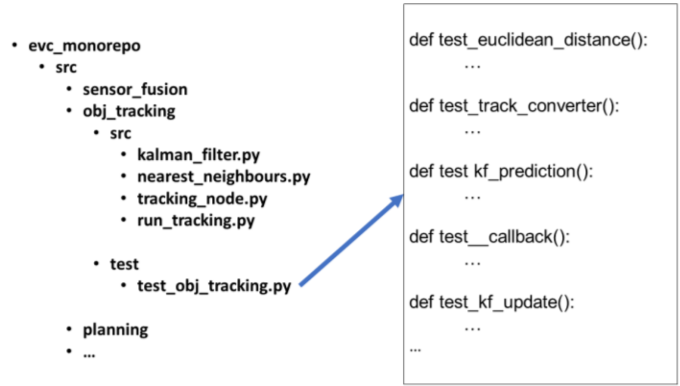


Figure 19: Example of old unit test organization for object tracking module

Taking direction from [34], the naming convention was changed to follow *UnitOfWork_StateUnderTest_ExpectedBehavior*. For instance, a test of the Kalman Filter's predict method in the kalman_filter.py file to ensure it returns the expected prediction can be named: *predictStep_validPrediction_returnPredictedState*. The same principle can be applied to the remaining tests and a new test file can be created to group tests for each of the *src* files in the *obj_tracking* module. This is shown in Figure 20.

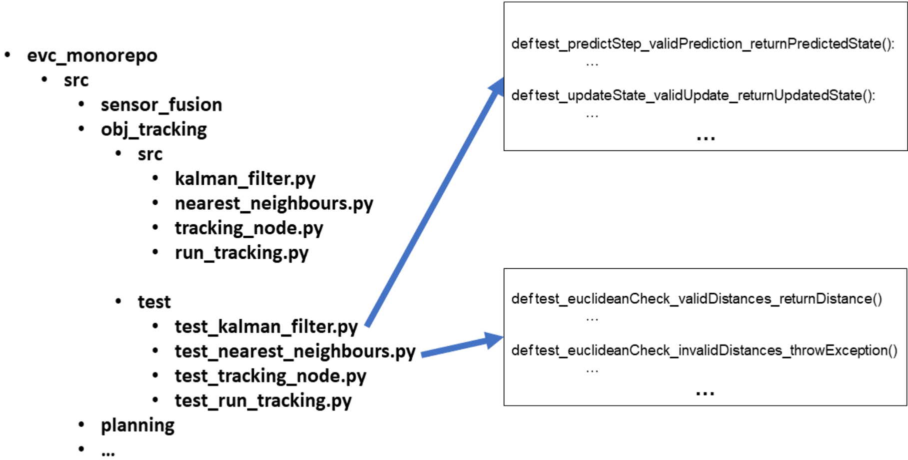


Figure 20: New folder structure and naming convention for unit tests as per recommendation #6

Contrary to the old folder structure, a test file for each corresponding src folder script leaves no question as to where the area under test is derived from. Within each test file, the new naming convention for unit tests also clarify the scope of each test. In the top right callout of Figure 20 it is evident that two completely different methods are being tested within the kalman_filter.py script whereas the bottom right callout shows how to differentiate between two states of the same function (in that case Euclidean distance function) being tested for different behaviour.

Recommendation #2 – to increase code clarity and documentation – was realized in two steps. The first was to explicitly include a documentation requirement as part of the official merge request process for any change to the teams codebase. Furthermore it was included as part of the teams merge request template checklist as seen in Figure 21.

Checklist:

- [ ] My code follows the style guidelines of this project
- [ ] I have performed a self-review of my code
- [ ] I have commented my code, particularly in hard-to-understand areas
- [ ] I have made corresponding changes to the documentation
- [ ] My changes generate no new warnings
- [ ] I have added tests that prove my fix is effective or that my feature works
- [ ] New and existing unit tests pass locally with my changes
- [ ] Any dependent changes have been merged and published in downstream modules

Figure 21: Highlighted text showing documentation requirement added to team's default merge request template

To improve clarity in the test code, the Arrange-Act-Assert (**AAA**) framework can be leveraged as recommended in several sources [34] [35] [36]. As an example, it can be applied to the existing test case for validity of the Euclidean distance function as seen in Figure 22.

```
11  def test_euclidean_distance():
12
13      nearest_neighbours = MyNearestNeighbours(0)
14
15      tracked_object = TrackedObject()
16      fused_object = FusedObject()
17
18      tracked_object.dx = 1.0
19      fused_object.dx = 0.0
20      tracked_object.dy = 0.0
21      fused_object.dy = 0.0
22
23      assert nearest_neighbours.euclidean_distance(fused_object, tracked_object) == 1.0
```

```
29  def test_euclidean_distance():
30
31      # Arrange
32      nearest_neighbours = MyNearestNeighbours(0)
33
34      tracked_object = TrackedObject()
35      fused_object = FusedObject()
36
37      tracked_object.dx = 1.0
38      fused_object.dx = 0.0
39      tracked_object.dy = 0.0
40      fused_object.dy = 0.0
41
42      # Act
43      euclidean_distance = nearest_neighbours.euclidean_distance(fused_object, tracked_object)
44
45      # Assert
46      assert euclidean_distance == 1.0
```

Arrange
Act
Assert

Figure 22: Old test code (top) and new test code after application of AAA (bottom)

Figure 22 is a trivial example that likely doesn't present a significant increase in readability. However when test cases being more comprehensive or involve multiple assert requirements as

in Figure 23, applying the **AAA** framework increases readability of the test code by standardizing the organization of its subsections.

```
125     # Assert
126     assert track_obj.header.stamp == fused_obj_array.header.stamp
127     assert track_obj.active_counter == tracked_object.active_counter + 1.0
128     assert track_obj.dx == fused_object.dx
129     assert track_obj.dy == fused_object.dy
130     assert track_obj.vx == fused_object.vx
131     assert track_obj.vy == fused_object.vy
132     assert track_obj.ax == fused_object.ax
133     assert track_obj.ay == fused_object.ay
134     assert track_obj.head == fused_object.head
```

Figure 23: The benefit of AAA is more pronounced for more involved tests such as those with multiple assert requirements which are grouped together as opposed to spread throughout

Recommendation #3 entails avoiding interdependence between tests as well as minimizing the use of external dependencies. As seen in Figure 24, one way to do this is avoid the use of external processes / modules. In this figure the **CAV** team has developed a test to verify proper functionality of the callback method of the object tracking **Node** when it receives data from the upstream sensor fusion node. The former test strategy seen at the top of the figure leverages Pythons subprocesses in which the sensor fusion node is run. This requires the creation of processes that are external to the program and also requires that the process lifecycle is managed to ensure it does not remain active after test completion. Improper termination of processes increases memory consumption as well as possibly affects the outcome of future tests. As seen from the bottom of the figure the alternative method reduces dependencies (evident from less imports) by creating its own **ROS** publisher required to trigger the callback. Doing so using ROS's rclpy library also ensures proper cleanup of the nodes after use.

```
10  import subprocess
11  from subprocess import DEVNULL, STDOUT
12  import os
13  import signal
14
15  def test_callback_old():
16
17      # Arrange
18
19      # Act
20      start_command = f"ros2 run sensor_fusion run_sensor_fusion"
21      pro = subprocess.Popen(start_command, stdout=subprocess.PIPE,
22                             shell=True, preexec_fn=os.setsid)
23      time.sleep(5)
24
25      os.killpg(os.getpgid(pro.pid), signal.SIGTERM)
```

External Imports

External subprocess

(a)

```python
33  from rclpy.node import Node
34  import rclpy
35
36  def test_callback_new():
37      rclpy.init()
38      my_node = Node('sensor_fusion_node')
39
40      # Arrange
41      fused_obj_pub = my_node.create_publisher(FusedObjectArray,
42                                               'fused_object_topic',
43                                               qos_profile=10)
```

Node and publisher internal to test

(b)

Figure 24: Test that relies on more external dependencies (a) and test with fewer dependencies (b)

Per recommendation #4, the tradeoff between the time required to write tests versus the coverage that more tests usually provide is a balance for any development team. The self-prompts of 1 - "What does it mean when the test passes?" and 2 – "what does it mean when it fails?" from [40] are very useful in this context to discern value added tests cases versus development for the sake of meeting an arbitrary coverage goal from management (as pointed out in [5], management requests are often a primary motivation for test development). One tool that is suitable for the CAV team's efforts is coveralls.io because it works with ROS modules including support for Python and C++ (Figure 25). Many alternatives tools exist but the key takeaway as cited by Microsoft's case study is to pick a single tool that can be used teamwide [12]. This greatly increases the ease and likelihood of tool adoption among team members.

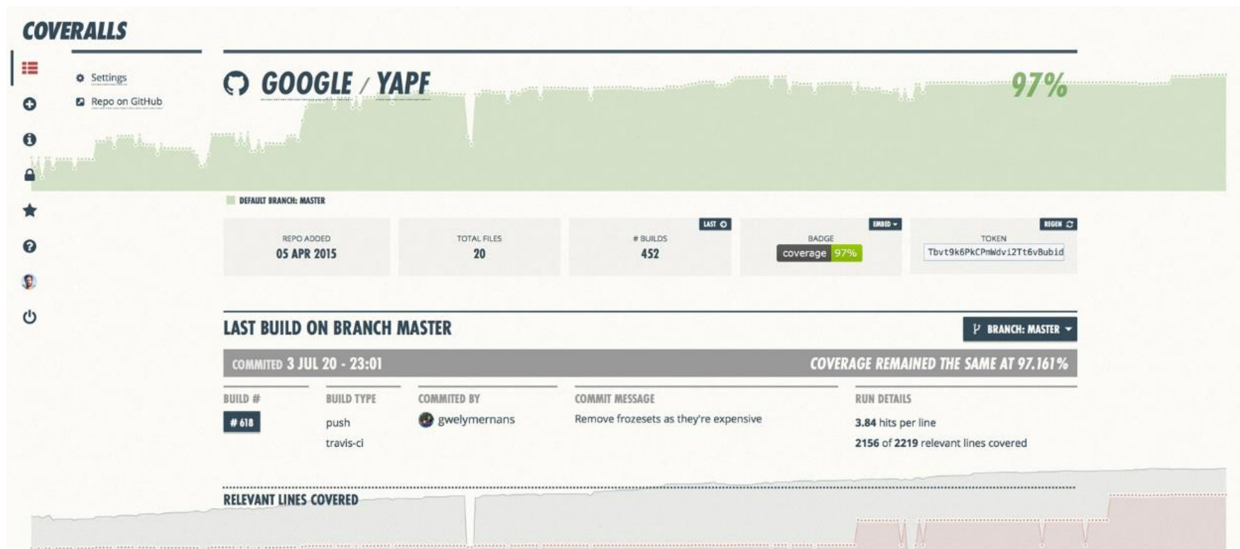


Figure 25: Example code coverage dashboard from coveralls.io

TDD as an approach to development (recommendation #5) is not something that can be suddenly ingrained in the teams processes with results that are immediately available. Thus, at this point we do not have conclusive data of the before and after effects of TDD within UWAFT. That said it is planned to be implemented in year 2 of EVC where sufficient internal team documentation will be added to the team documentation repository regarding TDD.

Lastly, and of highest importance to an efficient test development and usage pipeline, is recommendation #6, consideration of timeliness of the test suite including its ability to be run by automatically. Given its usage of Gitlab for hosting of the codebase, the **CAV** team leveraged Gitlab's built-in **CI** tools for test suite running. The team also benefitted from a design team dedicated server cluster built and maintained by WATOnomous to house dedicated **Test runners** [53]. Test runners have been added at the group and specific repository level so runners can be distributed evenly among ongoing **CAV** projects with some redundancy in case one of the servers are down. The assignment and prioritization of Gitlab runners are all set fairly intuitively in Gitlab's **CI** settings section (Figure 26). Following the setup of a runner, a YAML based CI configuration file (.gitlab-ci.yml) was set up using the online documentation provided as a reference [54].

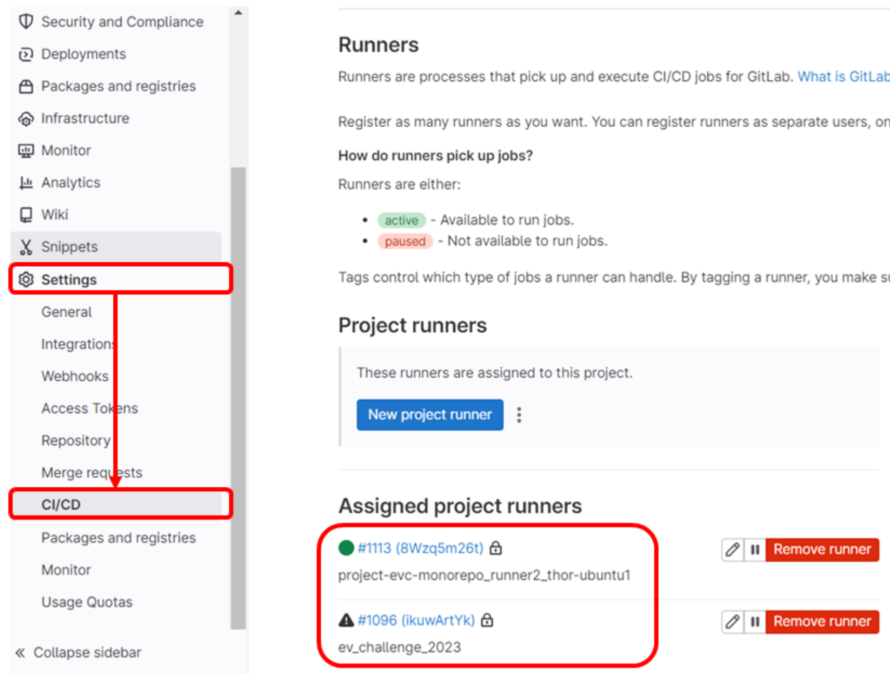


Figure 26: Project test runners are assigned using Gitlab's CI settings screen

4.3 Integration Tests

Integration tests utilize **UW-STF** as described in Section 3.2 to validate different sections of a system in parallel by using a single source of data stored in a **Bag file**. For the purpose of **CAV** related work, this bag file contains recorded sensor data collected from simulation (though real-world data is also a possibility). The scenario used in this section is an approach of a stationary lead vehicle which as seen from **NHTSA**'s pre-crash report was the most common crash scenario for a two light-vehicles. A video of the scenario can be found at this link with a screenshot of it shown in Figure 27.

Figure 27: Stationary approach scenario seen in CARLA

In this section several **Observers** developed for **Integration testing** are demonstrated using the aforementioned scenario. A summary of the subsequent integration test examples that will follow are listed below:

- **Heartbeat Observer for Sensor Inputs**
- **In Range Observer for Fusion Output**
- **Frequency Observer for Tracking Output**
- **Max Observer for Number of Tracks Detected**:

Heartbeat Observer for Sensor Inputs

The first test demonstrates the use of a Heartbeat Observer which checks for initial communication between sensor **Topic**s and its interfacing **ROS Node** in the system (Figure 28). It is one of the quickest tests to validate because it simply checks that data has been published but does not check the quality of its contents. Note that because the teams first iteration of their algorithm doesn't require object detection, the object fusion node is the furthest upstream. In this example we use it to verify that the connection between the **Bag file** and the start of the system works as expected.

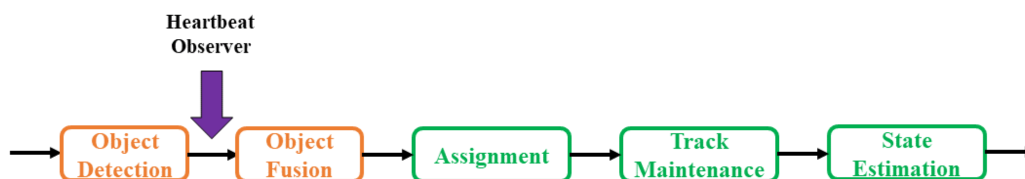


Figure 28: Location of Heartbeat Observer within system

The beginning of the **.sim file** for integration tests is shown on the lefthand side of Figure 29. The **Launch file** starts up the sensor fusion **Node** while the published and required **Topics** are also set up. Note that topics not directly related to sensor fusion appear (e.g. /tracked_obj) but

this is so the same *.sim* fle can be used for additional integration tests that will be described in later sections.

```
1  launch_files:
2    - package: structured_testing
3      file: my_system_launch.launch
4
5  published_topics:
6    - /fused_object_topic
7    - /tracked_obj_array
8    - /GroundTruth_topic_converted
9
10 required_topics:
11   - /FCM_topic
12   - /FrontLRR_topic
13   - /GroundTruth_topic
14   - /FrontLeftSRR_topic
15   - /FrontRightSRR_topic
16
17 bag_files:
18   - /system_testing/sf_scen1_input/sf_scen1_input.db3
19
20 test_duration:
21   - 20
```

```
23 observers:
24
25   - name: FrontCameraIsAlive                              Observer 1
26     observerClass: HeartbeatObserver
27     topic: "/FCM_topic"
28     msgType: "custom_sensor_interface.msg.FrontCam"
29
30   - name: FrontLRRIsAlive                                 Observer 2
31     observerClass: HeartbeatObserver
32     topic: "/FrontLRR_topic"
33     msgType: "custom_sensor_interface.msg.FrontLRR"
34
35   - name: FrontLeftSRRIsAlive
36     observerClass: HeartbeatObserver                      ...
```

Figure 29: Setup parameters of .sim file (left) including Observer section (right)

Once the setup parameters have been configured, the **Observers** can be added (righthand side of Figure 29). In this case a Heartbeat Observer is added for each sensor on the ego vehicle. As per the stock sensor suite (Section 4.1.2) this includes 1 **FCM**, 1 Front **LRR**, 1 Front Left **SRR**, and 1 Front Right SRR. A check of the ground truth sensor is also added. With a simple copy and paste of the standard Heartbeat Observer schema and some minor modifications, 5 separate integration tests are set up in a matter of minutes. The corresponding results file after running the simulation is shown in Figure 30. As per the figure, the front camera, long range radar, and ground truth sensors have passed but the left and right radars have not established **ROS** communications with the software **Stack**. In general miscommunication issues may be due to incorrect **Topic** naming, an issue with the **Bag file** itself or improper specification of a message type. In this case the left/right short range radars were purposely disabled in simulation to validate that the issue would be caught. Due to the purely longitudinal direction of the scenario, it is not critical that they be used.

```
1   - name: FrontCameraIsAlive
2     status: PASS
3     observerClass: HeartbeatObserver
4     topic(s):
5     - /FCM_topic
6     observerType: TRUE_ONCE
7
8   - name: FrontLRRIsAlive
9     status: PASS
10    observerClass: HeartbeatObserver
11    topic(s):
12    - /FrontLRR_topic
13    observerType: TRUE_ONCE
14
15  - name: FrontLeftSRRIsAlive
16    status: FAIL
17    observerClass: HeartbeatObserver
18    topic(s):
19    - /FrontLeftSRR_topic
20    observerType: TRUE_ONCE
21
22  - name: FrontRightSRRIsAlive
23    status: FAIL
24    observerClass: HeartbeatObserver
25    topic(s):
26    - /FrontRightSRR_topic
27    observerType: TRUE_ONCE
28
29  - name: GTSensorIsAlive
30    status: PASS
31    observerClass: HeartbeatObserver
32    topic(s):
33    - /GroundTruth_topic
34    observerType: TRUE_ONCE
```

Figure 30: Results file from running Heartbeat Observer on sensor topics

In Range Observer for Fusion Output

Having verified communications between the sensor data and the fusion **Node**, we can move further downstream to test the sensor fusion output itself. This **Observer** verifies that the output of a **Node** responsible for object fusion is within reasonable bounds (Figure 31). For high level performance gauging, the In Range Observer is of great use given a priori knowledge of the node under test's behavior.

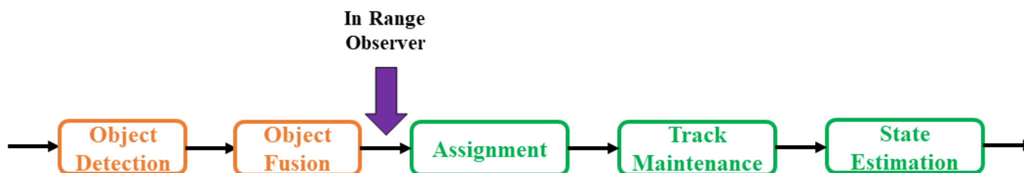


Figure 31: Location of In Range Observer within system

For example, we can first observe a plot of the ground truth of the *relative* longitudinal distance between ego and target vehicle, identified as DX (distance in the +x direction) as seen in Figure 32. As evident in the scenario video/figure, two target vehicles are present in the vicinity of the ego vehicle with only one being in the current lane.

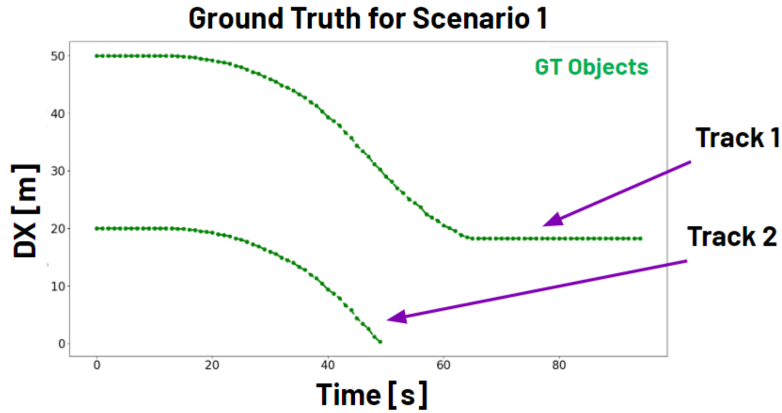
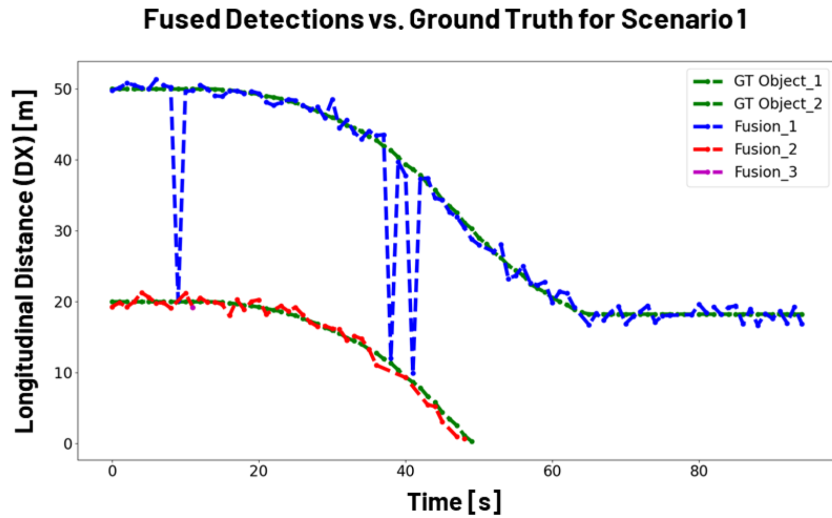


Figure 32: Ground truth plot for two target vehicles in scenario 1

Given that the range of ground truth for both vehicles is between 0 to 50m, we can add a rough 20% error tolerance to the **Observer** definition of the max value. Because only the frontal field of view is relevant for this scenario no tolerance less than 0m is required. Overlaying the fused detection results over ground truth produces the plot at the top of Figure 33. The bottom of the Figure shows the Observer definition as part of the generated results file.




```
33    - name: LeadVehicleInWindow
34      status: PASS
35      observerClass: InRangeObserver
36      topic(s):
37      - /fused_object_topic
38      field: fused_objects
39      observerType: ALWAYS_TRUE
40      minValue: 0.0
41      maxValue: 60.0
```

Figure 33: Sensor fusion algorithm results overlayed on ground truth for scenario 1 (top) and the corresponding the Observer entry in the .simresults file.

Evident in the plot and the results file, the test passes as detection results came nowhere close to the 60m maximum. It is important to note that the 20% error tolerance is not a hard rule. Tolerances must be defined by the user based on their understanding or expectations of system performance. It is not meant to be a target metric either and is better used as a "minimum" expectation for the system to pass.

Frequency Observer for Tracking Output

A higher data rate often benefits downstream nodes which are then more quickly able to respond to changes in inputs. For the use case where data must be transmitted at a minimum frequency, the Frequency **Observer** is an ideal choice for testing. Note that though this observer is located at the output of the system, it is not considered a system test because it specifically relates to the quality of data of being output from the last submodule (state estimation) as seen in Figure 34, and is not a reflection of the whole systems performance as a whole.

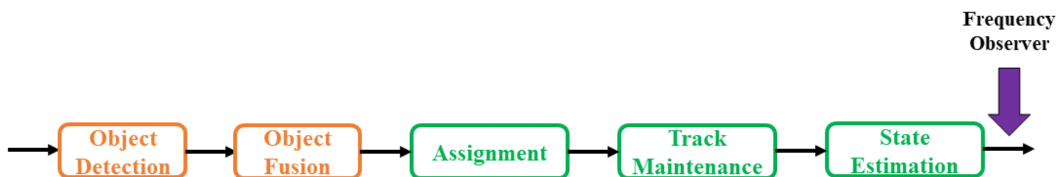


Figure 34: Location of frequency observer within system

Thus this example utilizes the Frequency **Observer** to ensure the output of the tracking module is transmitted at a minimum frequency. The definition of the Observer can be seen in Figure 35.

```
59    - name: TrackedMsgFrequency
60      observerClass: FrequencyObserver
61      topic: "/tracked_obj_array"
62      msgType: "obj_tracking_interface.msg.TrackedObjectArray"
63      minFreq: 8
```

Figure 35: Frequency Observer definition for tracked object data specifying 8Hz

Because the team does not possess the vehicle and thus stock sensors to make determinations about a reasonable data transfer rate, 10Hz was used as a conservative estimate when setting the ROS **Node** spin rate for the different modules. This is based on the lowest frequency sensor for the previous competition vehicle in the EcoCAR Mobility Challenge, where stock radars and cameras were also present. Using the 20% tolerance rule, the minimum frequency defined to be considered a pass for the test was 20% less than 10Hz. This specification is evident by the 8Hz minimum frequency defined in Figure 35. Then running the test as part of the integration test suite we have been building upon thus far produces the *.simresult* entry seen in Figure 36. The performance of the system is close to the 10Hz requirement that was set for the system and thus definitely passes the more lenient 8Hz requirement defined by the **Observer**.

```
40  - name: TrackedMsgFrequency
41    status: PASS
42    observerClass: FrequencyObserver
43    topic(s):
44    - /tracked_obj_array
45    observerType: TRUE_AT_END
46    minFreq: 8.0
47    actualFreq: 10.116752598639687
```

Figure 36: Output of frequency Observer test when an 8Hz minimum frequency is specified

Max Observer for Number of Tracks Detected

Another relevant metric to evaluate from the output of the object tracking **Node** is the tracks themselves. At a high level, the number of tracks detected provide information of the algorithms ability to distinguish target objects and match them continuously to their previous frames over time. In general the Max/Min **Observer** verifies that a tested message is above or below a certain threshold reflecting a minimum pass criteria for the system performance (Figure 37). In this case, we use the Max Observer.

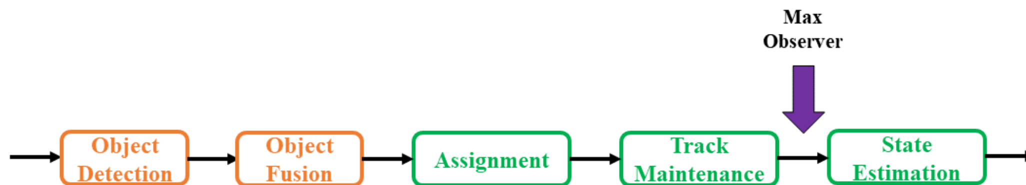


Figure 37: Location of Max Observer within system

Based on the ground truth plot for this scenario (Figure 32), it is known that the correct number of tracks throughout the simulation is 2. When setting a pass limit for this Observer, the 20% rule may not suffice because the number of tracks needs to be an integer. For this reason, we'll set the max to be 3 to provide a proper tolerance for error. This is seen in the lefthand side of Figure 33. The righthand side on the other hand shows the results once the test is completed. It can be seen that the actual value did reach the maximum of 3 tracks but did not surpass it.

```
65  - name: NumberOfTracks
66    observerClass: MinMaxObserver
67    observerType: ALWAYS_TRUE
68    topic: "/tracked_obj_array"
69    msgType: "obj_tracking_interface.msg
70    field: "num_of_tracks"
71    value: 3
72    isMin: False
```

```
48  - name: NumberOfTracks
49    status: PASS
50    observerClass: MinMaxObserver
51    topic(s):
52    - /tracked_obj_array
53    field: num_of_tracks
54    observerType: ALWAYS_TRUE
55    isMin: false
56    targetValue: 3.0
57    actualValue: 3.0
```

Figure 38: Max Observer definition for max tolerable number of tracks (left) and corresponding results file output showing that output was equal to max limit (right)

When plotting the number of tracks detected over the course of the simulation the team can identify where the shortcomings in detections occur as seen in Figure 39. Thus, **Integration testing** here plays the role of a quick identifier of weaknesses in the system. Though it does not provide exact feedback on areas that need improvement it gives developers a start to their 'search area' when debugging an **ADAS** algorithm.

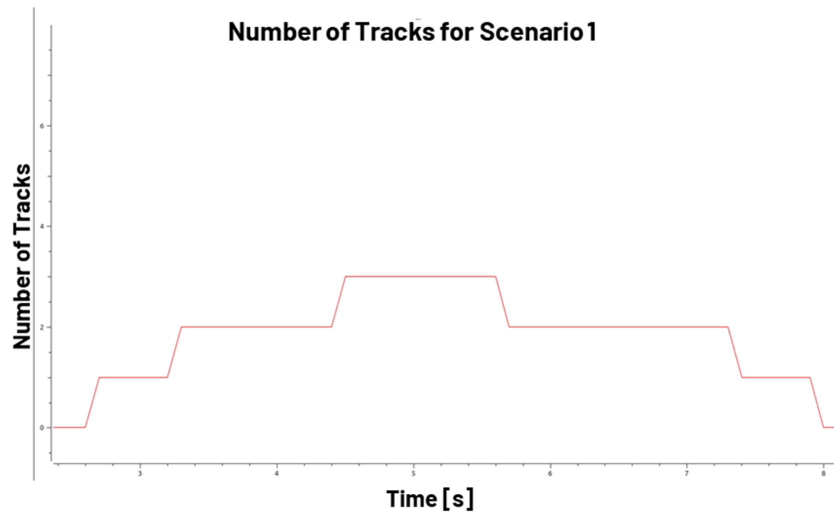


Figure 39: Number of tracks detected over course of scenario 1

4.4 System Tests

System tests involve the full scale of the software **Stack** and is the highest level of testing before acceptance testing (out of scope for this thesis). Thus, the defined **Observer**s reflect more standardized metrics for system evaluation of the **CAV** teams algorithm in year 1. The metrics used by **UWAFT** are expanded on in Section 4.4.1. Following an explanation of the metrics, Section 4.4.2 shows how they are applied to scenarios selected based on NHTA's report on pre-crash typology [47]. Note that the purpose of **UW-STF** is not to improve metric performance of an **ADAS** system but rather to objectively evaluate it. The metric results presented in Section 4.4.2 represent the teams first iteration of a simple algorithm and is not an evaluation of UW-STF itself. Section 4.4.3 however evaluates UW-STF itself as a test framework and thus provides more useful analysis as to how well the framework actually met its design intents.

4.4.1 Metrics

The first metric used at the system test level is **F1-score**. It is known as the harmonic mean of precision and recall, indicating its ability to convey information about a system in terms of both its ability to capture true positives amidst all its perceived detections (recall) as well as accurately signal a detection whenever a true object is present (precision) [55]. In the context of the **CAV** team's development efforts, a True Positive (TP) is defined as a detection that corresponds to a ground truth object (while a False Positive, FP, does not) and a True Negative (TN) is defined as a lack of detection output given that no ground truth object exists (while a False Negative, FN, occurs given a ground truth object actually does exist). This is visually

presented in Figure 40. Precision and Recall are then fractions of these binary combinations as per Equations 4 and 5 while F1-core combines both previous measures as in Equation 6.

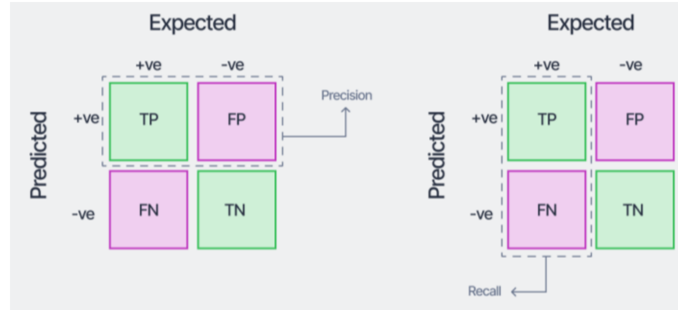


Figure 40:Visualization of precision and recall as seen in [56]

$$Precision\ (P) = \frac{TP}{TP + FP} \qquad (4)$$

$$Recall\ (R) = \frac{TP}{TP + FN} \qquad (5)$$

$$F1\ Score = \frac{2 \cdot P \cdot R}{P + R} \qquad (6)$$

To realize an interpretation of the F1 score that was applicable to vehicle detection in **ADAS** scenarios, a tolerance bound was set for what was considered a TP. Unsurprisingly, expecting detections to be exactly at ground truth positions is unrealistic given realities such as sensor noise and other unideal real world driving conditions. The tolerance for the first iteration of the **CAV**s algorithm was set to be 4m based on the average length of a light vehicle.

The second metric of use to the **CAV** team for algorithm evaluation was Optimal Sub Pattern Assignment (**OSPA**). While the F1-score is based on binary classification of detections at a certain timestamp, OSPA is oriented towards the evaluations of tracks. Tracks take into account potential temporal disruptions in detections by representing a labeled sequence of state estimates over said period of time [57]. OSPA was developed to address several shortcomings of other metrics in multi-object tracking [58]. The Hausdorff metric for instance, though very suitable to measure dissimilarity between binary images is insensitive to differing cardinalities of sets. The Optimal Mass Transfer (**OMAT**) metric partly addresses the cardinality shortcoming of Hausdorff but introduces its own problems such as inconsistency when there are different numbers of points assigned to ground truth objects as well as behavior dependent on the

geometry of the ground truth objects themselves. OMAT is also undefined if the cardinality is zero. Thus OSPA is introduced which is described and defined in Figure 41 and Equation 7.

Denote by $d^{(c)}(x,y) := \min(c, d(x,y))$ the distance[1] between $x, y \in W$ cut off at $c > 0$, and by Π_k the set of permutations on $\{1,2,\ldots,k\}$ for any $k \in \mathbb{N} = \{1,2,\ldots\}$. For $1 \leq p < \infty$, $c > 0$, and arbitrary finite subsets $X = \{x_1,\ldots,x_m\}$ and $Y = \{y_1,\ldots,y_n\}$ of W, where $m,n \in \mathbb{N}_0 = \{0,1,2,\ldots\}$, define

Figure 41: Notation description for Equation 4 from [58]

$$\bar{d}_p^{(c)}(X,Y) := \left(\frac{1}{n}\left(\min_{\pi\in\Pi_n}\sum_{i=1}^{m} d^{(c)}(x_i, y_{\pi(i)})^p + c^p(n-m)\right)\right)^{1/p} \quad (7)$$

Parameters *p* and *c* are set by the user. Increases in *p* are interpreted as higher leniency to outlier objects. Cut-off *c* on the other hand determines how much cardinality errors are penalized. Applying this to the teams use case, *p* is set as 2 as recommended by [58] while the cutoff *c* is set to be 4 which is approximately an average car length [59].

4.4.2 Results

Scenario 1 – Stationary Approach

The stationary approach of a lead vehicle, besides being the most common on **NHTSA**'s list of light vehicle crashes, was sought after by the team because of the focus of year 1 on ease of implementation over accuracy. Thus, scenarios that were simpler to interpret, also meant quicker debugging and validation of the system (albeit not extremely challenging for it performance wise). As described in earlier sections, **CARLA** was used to generate the simulation data leveraging the development work for simple scenario generation described in Section 3.3.2. As with **Integration testing**, the **Observer**s must be defined and added to the **.sim file** prior to the start of the simulation. On a new separate file than what was used for integration testing, both the **F1-score** and **OSPA** metric Observers are defined. Their definitions are shown in Figure 42.

```
23  observers:
24    - name: SystemF1Score
25      observerClass: F1ScoreObserver
26      topic:
27        - "/tracked_obj_array"
28        - "/GroundTruth_topic_converted"
29      msgType:
30        - "obj_tracking_interface.msg.Tra
31        - "ground_truth_interface.msg.Gro
32      field:
33        - "track_array"
34        - "gt_objects"
35      num_targets: 2
36      output_type: variable
37      tol: 4
38      f1_pass_score: 0.8
```

```
23  observers:
24    - name: OSPA
25      observerClass: OSPAObserver
26      topic:
27        - "/tracked_obj_array"
28        - "/GroundTruth_topic_converted"
29      msgType:
30        - "obj_tracking_interface.msg.Tracke
31        - "ground_truth_interface.msg.Ground
32      field:
33        - "track_array"
34        - "gt_objects"
35      num_targets: 2
36      cut_off: 4
37      sens: 2
38      ospa_pass_score: 3
```

Figure 42: Key fields for each system test Observers is highlighted in their corresponding .sim file. F1-score is shown on the left while the OSPA is on the right.

As described in Section 4.4.1, custom metric parameters such as the true positive tolerance for **F1-score** and the outlier sensitivity and cut-off parameters for **OSPA** are defined in a user friendly YAML format under the **Observer** like any other field. Note the addition of a "_pass_score" field that was not mentioned earlier. This defines a minimum/maximum metric score that must be achieved for the test to be considered a PASS/FAIL. This is so the **CI** pipeline simply needs to parse the result to make a determination on overall pipeline status. For F1-score a higher score is better so the pass score is a minimum. Alternatively, because OSPA is a loss distance, a lower value is better and thus the pass score is a maximum. Similarly to the pass scores for integration tests, the users should consider this a minimum performance goal rather than an ideal target. **UWAFT** elected to choose their minimum requirements on numbers grounded in a literature review of state of the art performance results. State of the art machine learning techniques for F1-scores for example may achieve values in the range of 90-97% such as in the work by Alqarqaz et el. [60]. Considering the team was focused on ease of implementation over accuracy, and thus would not be pursuing state of the art algorithms in year 1, a pass criterion of 80% for F1score was chosen. A similar literature review in the use of OSPA for **ADAS** related performance evaluation showed scores < 2m such as the approach used by Lee, Kim and Lim [61]. By the same reasoning as for F1 score, UWAFT**UWAFT**'s pass criterion was set slightly more lenient than this at 3m.

Because both longitudinal and lateral position is considered, RViz was used to visualize detections alongside the simulation. A video of the visualization efforts is seen at this link. An annotation of the actors in it is seen in Figure 43. The data is then saved to a **Bag file** where it can now be used as an input to **UW-STF**. The results of the simulation for just the longitudinal direction (DX) is shown in Figure 44 where ground truth as well as the output of the tracking module is displayed.

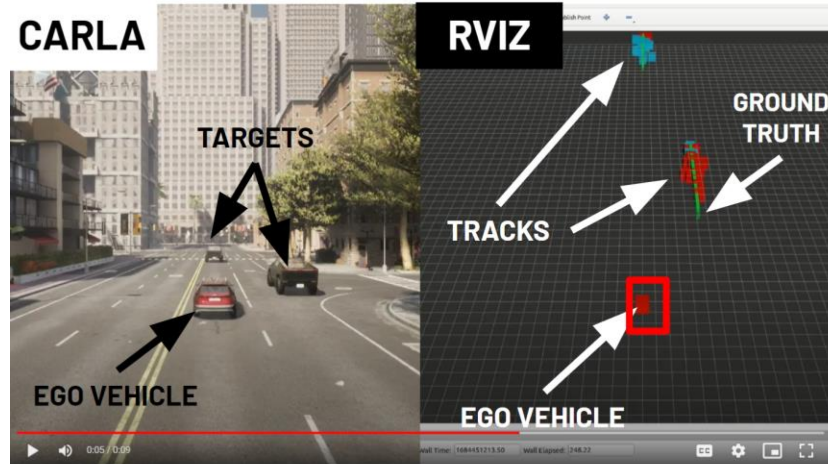


Figure 43: Annotation of CARLA and RViz visualization consisting of the ego vehicle, targets (tracks) and ground truth lines (green)

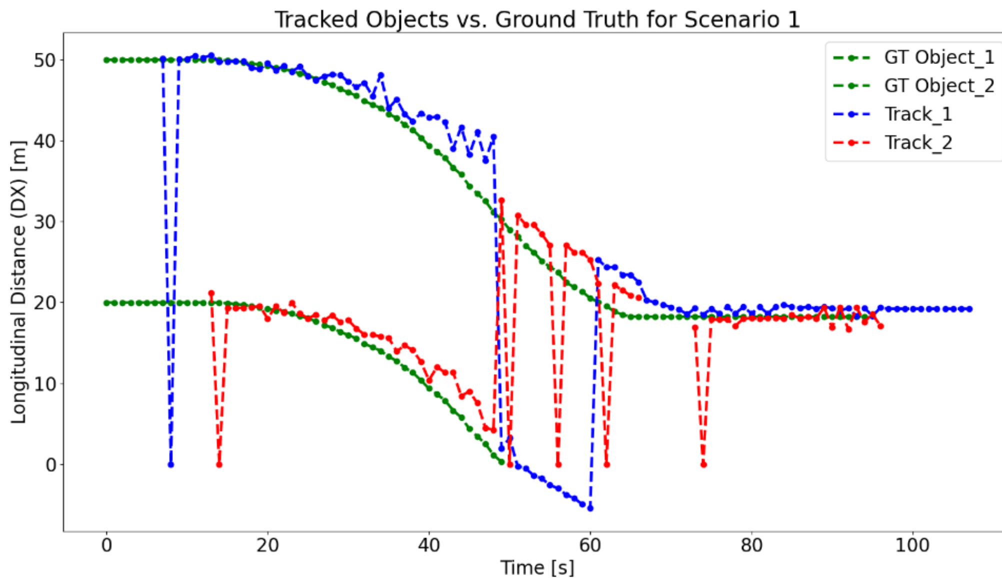


Figure 44: Ground truth and system tracking results for scenario 1

It is evident in the video of the simulation and by the corresponding plot that track switching occurs once the object in the adjacent lane goes out of sight. Other than this section however, the tracking algorithm is able to separate the objects and relatively consistently follow the ground truth objects over time. Finally, running the **UW-STF** pipeline yield the **.simresults file** in Figure 45. It can be seen that both F1 score and **OSPA** pass criterion is met. This was expected given the simple nature of the scenario.

```
- name: SystemF1Score
  status: PASS
  observerClass: F1ScoreObserver
  topic(s):
  - /tracked_obj_array
  - /GroundTruth_topic_converted
  field:
  tol: 4
  f1PassScore: 0.8
  precision: 0.84
  recall: 0.76
  fScore: 0.8
```

Actual score >= pass score so F1-score test is a PASS

```
- name: OSPA
  status: PASS
  observerClass: OSPAObserver
  topic(s):
  - /tracked_obj_array
  - /GroundTruth_topic_converted
  field:
  observerType: TRUE_AT_END
  cutoff: 4
  sensitivity: 2
  ospaPassScore: 3
  meanOSPA: 2.23
```

Actual score <= pass score so OSPA test is a PASS

Figure 45: System test Observer results for scenario 1. F1-score shown on the left and OSPA shown on the right.

Scenario 2 – Lange Change

The second scenario chosen is a lane change scenario from a target vehicle in an adjacent lane to that of the ego vehicle. This was selected from the list of top 5 **NHTSA** pre-crash scenarios as well because of its inclusion of lateral movement by the target vehicle (whereas scenario 1 only involved relative longitudinal movement). A video of the scenario including its side by side RViz tracking visualization is seen in at this link. The moment a lead vehicle turns into the ego vehicle lane is captured in the screenshot in Figure 46.

Figure 46: The moment in simulation where a vehicle turns into the ego vehicles lane (Scenario 2)

Similar to scenario 1, **F1-score** and **OSPA** are used as the measures of evaluation. The results of these generated by **UW-STF** are seen in Figure 47.

```
- name: SystemF1Score
  status: FAIL
  observerClass: F1ScoreObserver
  topic(s):
  - /tracked_obj_array
  - /GroundTruth_topic_converted
  field:
  - track_array
  - gt_objects
  observerType: TRUE_AT_END
  tol: 4
  f1PassScore: 0.8
  precision: 0.75
  recall: 0.72
  fScore: 0.74
```

```
- name: OSPA
  status: PASS
  observerClass: OSPAObserver
  topic(s):
  - /tracked_obj_array
  - /GroundTruth_topic_converted
  field:
  - track_array
  - gt_objects
  observerType: TRUE_AT_END
  cutoff: 4
  sensitivity: 2
  ospaPassScore: 3
  meanOSPA: 2.6
```

Figure 47: F score and OSPA results of scenario 2. F1-score is worse than the pass score so the test fails. OSPA however is better than the pass score so it passes.

For scenario 2, the F1 score is below the pass criterion but the **OSPA** shows that the system passes. Similar to **Integration testing** Observers, these results do not provide an in depth root cause analysis of what went wrong, but there are learnings to be made. The failure of the F1 score may point at weaknesses in the perception module as that is what generates the detections that tracks are based on. Alternatively, perhaps the pass score set for OSPA was too lenient as well causing it to detract from shortcomings in the tracking algorithm. Having tied this into our **CI** pipeline the system status is thus currently set as a failure. The benefit of UW-STF however is that any further changes to the code do not require any modifications to the Observer **.sim files** unless they are **Topic** definition changes. This means alternative algorithms, techniques, and even bug fixes can be validated quickly and automatically.

4.5 Quantitative Performance Analysis

Now that the functionality of **UW-STF** when applied to a real world use case has been proven, a reasonable next step is to objectively evaluate its performance at doing so. Accordingly, we compared UW-STF's performance against **ROS**'s Standard Integration Testing (**SIT**) approach described in [24] on the basis of several metrics. Specifically, the analysis compared usability (based on lines of code required for a test) as well as timeliness/efficiency (based on the metrics of runtime, **CPU** - Central Processing Unit - and memory consumption) as shown in Tables 2 and 3 [62]. In this analysis the same input scenario, compute hardware and test types were used. The only difference is that in UW-STF predefined **Observer**s (In Range Observer and Minimum Frequency Observer) are used to carry out the tests while for **SIT** we replicate the functionality of these Observers from scratch using conventional Python development. The comparison is expressed as a percentage difference between the two frameworks.

Table 2: Lines of code comparison

Test Type	Lines of Code		Reduction
	SIT	*UW-STF*	
In Range	112	36	68%
Minimum Frequency	114	44	61%

Table 3: Difference in runtime, CPU and memory requirements of UW-STF compared to SIT

Test Type	Delta (UW-STF Minus SIT)		
	Runtime(s)	*CPU (%)*	*Memory (%)*
In Range	-0.00299	-6.2	-0.690
Min. Freq.	-0.0891	-6.0	-0.684

As evident by the data, UW-STF had a significant reduction in the lines of code required to implement both test types as well as statistically significant improvements in CPU and memory consumption based on t-test results [62]. Runtime improvements were deemed not to be of statistical significance however.

4.6 Chapter Summary

- To validate its functionality with source code, the **UWAFT** codebase is used. The team is developing **ADAS** features for a 2023 Cadillac Lyriq as part of **EVC** and thus has made decisions on the software and hardware architecture that will define the automated features of the prototype vehicle.
- The proposed **Unit testing** guide is used to highlight areas for improvement in the teams codebase and corresponding organization/code changes are made in some cases.
- **UW-STF** is applied at the integration test level where sample tests are used to check communication and data quality for different nodes in the architecture
- At the system test level, the full-scale pipeline from Section 3 is utilized in which scenarios are chosen from **NHTSA**'s report, and then the simulated data is realized in **CARLA**. Next, UW-STF is applied in a similar way to the previous level except with **Observer**s that generate **F1-score** and **OSPA** metrics for the system that are based on algorithm performance in each of the scenarios.

5 Conclusion

5.1 Summary of Work and Current Limitations

The proposed test framework, validated by the performance metric results in Section 4.5, as well as the case study of the **UWAFT** codebase, demonstrates success at the initial objective of the completion of a full-scale **open-source test framework** to facilitate **rapid and modular testing**. These features also give the way for potential adoption by stakeholders in both industry and academia.

Specifically, this thesis has provided a thorough framework for validation of **ADAS** software systems including unit, integration and system level testing. At the **Unit testing** level, the most commonly expressed sentiments across the reviewed online sources on improving test quality were succinctly tied to six recommendations with actionable items that any development team could integrate in their own processes. At the **Integration testing** and **System testing** level, a custom test framework known as **UW-STF** was developed to address many of the shortcomings of existing testing pipelines. Through another literature review, these shortcomings were found to include that existing frameworks were mainly geared towards comprehensive testing of the system and not rapid validation for **CI** purposes. With this in mind, UW-STF was designed with the intent to: 1 – provide an intuitive user interface for developers to make quick changes to testing, 2 – integrate easily with any existing CI tool for automated test running, 3 – be adaptable to multiple levels of testing, and 4 – be easily integrable with any teams software platform that uses **ROS**. Through a YAML file based test specification schema and a monitoring system that uses "**Observer**" nodes, these goals were addressed. In addition at the system test level, a simulation pipeline was proposed to address major hurdles in existing system testing in relation to scenario prioritization and generation in simulation. The combination of research done by the **NHTSA** for scenario prioritization and use of **CARLA** and ScenarioRunner to realize these scenarios was selected as the best fit to work with the rest of the UW-STF pipeline.

To verify the efficacy of the proposed framework, we apply it to the codebase of the **CAV** subteam within the **UWAFT**, a team competing in the EcoCAR challenge to, among other goals, automate a 2023 Cadillac Lyriq. Starting with **Unit testing**, the CAV codebase is analysed for prospective improvements based on the unit testing guide. Multiple improvements for cleaner and more maintainable code are made including leveraging the Arrange-Act-Assert framework and reducing the external dependencies of the codebase. At the integration level, checks for prerequisite **ROS** communications and minimum acceptable data quality are all made using UW-STF's **Observers**. At the system level, the proposed simulation pipeline using **NHTSA**'s pre-crash report and CARLA to generate scenarios greatly simplified the process to test the CAV teams algorithms. At the end of their first year, the algorithm saw reasonable performance in both stationary lead vehicle and moving lane change scenarios based on metrics of an average **F1-score** of 0.77 and average **OSPA** of 2.42. However, much work remains to be done to improve the accuracy and reliability of the system to be able to test their algorithms in vehicle. Lastly, a quantitative analysis comparing UW-STF to **SIT** is made. The analysis shows a greater than 60% reduction in lines of code required to write an equivalent test using UW-STF as

compared to SIT. UW-STF also demonstrated improvements in approximately 6% reduction in **CPU** and 0.7% reduction in memory usage for a given test.

Despite the initial success in establishing a test framework that meets the teams baseline goals, there still exist limitations on its application as well as opportunities for improvement. In regard to unit testing, a limitation of the literature review was the assumption of equal weighting given to the different sources surveyed. Despite the recommendation guide being arranged based on the number of mentions across the different sources, each source was not of the same quality. For example, large scale case studies consider more input than individual developer experiences. Providing weightings to the sources based on relative comprehensiveness may be a better approach to tacking this. The next limitation of significance is the lack of specific direction provided to the user when a test fails. As seen during the evaluation of **UWAFT**'s **Stack**, test failures were simply denoted on the **.simresults file** as a FAIL but without further description. This is especially true when UW-STF is tied into a **CI** pipeline because the *.simresults* file will not be visible to the user, only the console output. Gitlab's CI tool only provides the overall pipeline status as well as which stage in the pipeline failed. Another shortcoming of the proposed framework from a technical standpoint is the lack of determinism for Observers that require multi-**Topic** synching like the F1-score and OSPA Observers. Both require that the ground truth topic and the topic under test are time-synched so corresponding values can be compared to generate the metrics. Currently this is implemented using ROS2's *ApproximateTimeSynchronizer* message filter object which allows that messages within a certain time tolerance are associated with each other and passed to the same callback [63]. The problem with its current implementation is that different trials result in slightly different associations between ground truth and detection messages which manifests in slight variations in the final metric calculations. The validation of UW-STF also falls short in the domain of robust testing environments. That is, the framework has only been applied in the software-in-the-loop domain and was not designed for considerations of hardware and vehicle-in-the-loop testing. These higher fidelity testing environments often require consideration for communication interfaces such as CAN which is not handled by this testing framework. That said however, the benefit of using bag files is that they can be recorded from in-vehicle testing itself, as long as the topic names and format match up with its counterpart in simulation.

5.2 Future Work

To address the issue of a lack of test of test transparency future considerations should include the addition of lower-level debugging outputs, especially when **UW-STF** is integrated into a **CI** pipeline. As mentioned in Section 5.1, the current CI pipeline outputs represent very high level information and thus do not provide the user much additional support for debugging. More detailed print statements for failed test cases should be added to an **Observer** class such that specific failure points are transparent to the user also. In addition, if common reasons for test failure are documented by a team over the course of the project, corresponding resolution items can be documented as well. Furthermore, they can be output as suggestions available to the user to provide a direction for their debugging efforts.

The lack of determinism for some current system test metrics also represents a major priority for development. An alternative to the current approach using ROS2's *ApproximateTimeSynchronizer* is to implement a custom method for time synching of multiple **Topic**s using traditional Subscribers. A Cache message filter can be used to store messages from the separate topics and then pass them to be processed by the **Observer** once the timestamps are within an acceptable tolerance.

To further improve the ease and simplicity with which UW-STF is integrated into a teams software stack, an open repository of Observers would be beneficial to have. Alongside corresponding documentation of the Observers schema, users of UW-STF could simply pull Observers that best meet their needs from the repository and modify it to their needs. The additional open source aspect of sharing and using Observers made by other development teams would decrease the workload for both initial developer and end user.

A feature that would further improve the quality of the system status readout would be a **GUI** similar to the one used in [25] for **ROS** introspection capabilities. **CI** logs are not necessarily user friendly and do not provide a clear overview of the status of the system or specific submodules. An interface that users are able to view for specific test failures as well as what part of the system the test covered would be a value added feature for debugging. Furthermore, mapping failures to their source scenario when conducting **System testing** may provide insights into which type of driving scenario the **ADAS** algorithm is most challenged by and thus help justify future prioritization of test cases.

Lastly, and most importantly, to truly validate that the framework has achieved its design intent of modularity, it should be used in teams and software systems outside of **UWAFT**. Given that it was designed with the intent to be adopted by both industry and academia, having relationships with groups on both sides willing to trial it and provide feedback is crucial. Doing so provides the opportunity to receive feedback from users who may use the framework in ways different than initially intended, which increases the likelihood of rooting out bugs or additional limitations with the framework. Specific to UW-STF, continual and frequent updates of the source code are expected at the early stages of release, which will only help to improve the quality and adoption rates of it in the long run.

References

[1] Market.Us, "Autonomous Vehicles Market Size [+USD 3,444.1 Bn] | Expands Steadily at a CAGR of 38.8% by 2032, States Market.us," GlobeNewswire, 30 May 2023. [Online]. Available: https://www.globenewswire.com/news-release/2023/05/30/2678245/0/en/Autonomous-Vehicles-Market-Size-USD-3-444-1-Bn-Expands-Steadily-at-a-CAGR-of-38-8-by-2032-States-Market-us. [Accessed 30 May 2023].

[2] J. Becker, "A Brief History of Automated Driving — Part Two: Research and Development," Apex.AI, 5 October 2020. [Online]. Available: https://www.apex.ai/post/a-brief-history-of-automated-driving-part-two-research-and-development. [Accessed 30 May 2023].

[3] Porsche AG, "A Modern Car Runs on 100 Million Lines of Code — but Who Will Write Them in the Future?," Medium, 10 December 2021. [Online]. Available: https://medium.com/next-level-german-engineering/porsche-future-of-code-526eb3de3bbe. [Accessed 31 May 2023].

[4] AVTC, "Advanced Vehicle Technology Competitions," AVTC, [Online]. Available: https://avtcseries.org/. [Accessed 31 May 2023].

[5] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," *IEEE 25th International Symposium on Software Reliability Engineering,* pp. 201-211, 2014.

[6] G. Lou, Y. Deng, X. Zheng, M. Zhang and T. Zhang, "Testing of Autonomous Driving Systems: Where Are We and Where Should We Go?," in *ESEC/FSE 2022*, Singapore, 2022.

[7] Canonical Ltd., "What is ROS," Canonical Ltd., 2023. [Online]. Available: https://ubuntu.com/robotics/what-is-ros. [Accessed 6 July 2023].

[8] M. Luqman, "Understanding ROS Nodes," Open Robotics, 18 October 2022. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes. [Accessed 7 June 2023].

[9] R. Blake, R. Ward, M. Woods, T. Jorden, E. Allouis, B. Maddison, S. Gunes-Lasnet and H. Schroeven-Deceuninck, "A Facility For Verification & Validation of Robotics & Autonomy for Planetary Exploration," in *44th Lunar and Planetary Science Conference*, The Woodlands, 2013.

[10] T. Bhat and N. Nagappan, "Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, Rio de Janeiro, 2006.

[11] N. Nagappan, E. Maximilien, T. Bhat and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Software Engineering,* vol. 2008, no. 13, pp. 289-302, 2008.

[12] L. Williams, G. Kudrjavets and N. Nagappan, "On the effectiveness of unit test automation at Microsoft," in *2009 20th International Symposium on Software Reliability Engineering*, Mysuru, 2009.

[13] D. Kumara and K. Mishrab, "The Impacts of Test Automation on Software's Cost, Quality and Time to Market," *Procedia Computer Science,* vol. 79, no. 1877-0509, pp. 8-15, 2016.

[14] E. Soares, G. Sizilio, J. Santos, D. Alencar and U. Kulesza, "The Effects of Continuous Integration on Software Development: a Systematic Literature Review," *Empirical Software Engineering,* vol. 27, no. 3, 2022.

[15] P. Duvall, S. Matyas and A. Glover, Continuous integration improving software quality and reducing risk, Upper Saddle River: Addison-Wesley, 2013.

[16] M. Fowler, "Continuous Integration," martinfowler.com, 1 May 2006. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html. [Accessed 11 June 2023].

[17] I.-C. Donca, O. Stan, M. Misaros, D. Gota and L. Miclea, "Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects," *Sensors,* vol. 22, no. 12, 2022.

[18] J. Ernits, E. Halling, G. Kanter and J. Vain, "Model-based integration testing of ROS packages: A mobile robot case study," *European Conference on Mobile Robots (ECMR),* pp. 1-7, 2015.

[19] A. Bihlmaier and H. Worn, "Robot Unit Testing," in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2014.

[20] M. Bures, "Framework for Integration Testing of IoT Solutions," in *International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, 2017 .

[21] F. Lasaca, Z. Fu, A. Remazeilles, A. Wąsowski, J. Azpiazu and A. Alami, "Testing-Based Validation Infrastructure for ROS," ROSin, 2017.

[22] M. Brito, S. Souza and P. Souza, "Integration testing for robotic systems," *Software Quality Journal,* vol. 30, pp. 3-35, 2020.

[23] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Conference on Software Maintenance*, San Diego, CA, USA, 1990.

[24] Autoware, "Integration Testing," Autoware.Auto, 2022. [Online]. Available: https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto/index.html#autotoc_md58. [Accessed 14 June 2023].

[25] A. Bihlmaier and H. Worn, "Increasing ROS Reliability and Safety through Advanced Introspection Capabilities," in *Proceedings of the INFORMATIK 2014*, 2014.

[26] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. Jackel and U. Muller, "Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car," *CoRR,* vol. 1704.07911, pp. 255-266, 2017.

[27] R. Butler and G. Finelli, "The infeasibility of experimental quantification of life-critical software reliability," in *Conference on Software for Citical Systems*, 1991.

[28] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE Internation Journal of Transportation Safety,* vol. 2016, no. 1, pp. 15-24, 2016.

[29] R. Lattarulo, J. Pérez and M. Dendaluce, "A complete framework for developing and testing automated driving controllers," *IFAC-PapersOnLine,* vol. 50, no. 2405-8963, pp. 258-263, 2017.

[30] M. Tatar, "Enhancing ADAS Test and Validation with Automated Search for Critical Situations," in *Driving Simulation Conference & Exhibition*, Tübingen, 2015.

[31] M. Jiang, "Continuous Integration and Testing for Autonomous Racing in Simulation," University of Illinois, Urbana-Champaign, Urbana, 2021.

[32] T. Son, A. Bhave and H. Auweraer, "A simulation-based testing and validation framework for ADAS development," in *Proceedings of 7th Transport Research Arena TRA*, Vienna, 2017.

[33] P. Kaur, S. Taghavi, Z. Tian and W. Shi, "A Survey on Simulators for Testing Self-Driving Cars," in *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD)*, Detroit, 2021.

[34] Progress, "Unit Testing: The Complete Guide," 2020. [Online]. Available: https://www.telerik.com/docs/default-source/whitepapers/unit-testing-the-complete-guide.pdf. [Accessed 19 June 2023].

[35] E. Vartanian, "All about unit testing: 11 best practices and overview," educative, 30 May 2022. [Online]. Available: https://www.educative.io/blog/unit-testing-best-practices-overview#unit-testing-best-practices. [Accessed 30 March 2022].

[36] A. Ataman, "10 Unit Testing Best Practices in 2023," AIMultiple, 20 January 2023. [Online]. Available: https://research.aimultiple.com/unit-testing-best-practices/. [Accessed 19 June 2023].

[37] Upwork Global Inc., "What Is Unit Testing? Frameworks, Examples, and Best Practices," Upwork, 15 June 2021. [Online]. Available: https://www.upwork.com/resources/unit-testing#unit-testing-best-practices. [Accessed 19 June 2023].

[38] S. Kolodiy, "Unit Testing and Coding: Why Testable Code Matters," Toptal, 2015. [Online]. Available: https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters. [Accessed 19 June 2023].

[39] J. Reese, "Unit testing best practices with .NET Core and .NET Standard," Microsoft, 11 April 2022. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices#characteristics-of-a-good-unit-test. [Accessed 20 June 2023].

[40] Parasoft Corporation, "Top 5 Unit Testing Best Practices," 2017. [Online]. Available: https://markt.all-electronics.de/uploads/documents/32b668cc6c49249da290a8f5d6d0ac41869582ac.pdf. [Accessed 20 June 2023].

[41] V. Khorikov, Unit Testing Principles, Practises and Patterns, Shelter Island: Manning Publications Co., 2020.

[42] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[43] R. Martin, Agile Software Development, Principles, Patterns, and Practices, Pearson, 2002.

[44] S. Ambler, "Introduction To Test Driven Development (TDD)," Agile Date, 2022. [Online]. Available: http://agiledata.org/essays/tdd.html#TraditionalTesting. [Accessed 21 June 2023].

[45] Microsoft, "Unit Testing," Microsoft, 15 December 2022. [Online]. Available: https://microsoft.github.io/code-with-engineering-playbook/automated-testing/unit-testing/. [Accessed 21 June 2023].

[46] Open Robotics, "Creating a launch file," Open Robotics, 2023. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Intermediate/Launch/Creating-Launch-Files.html. [Accessed 22 June 2023].

[47] National Highway Traffic Safety Administration, "Pre-Crash Scenario Typology for Crash Avoidance Research," U.S. Department of Transportation, Springfield, 2007.

[48] B. Clayton and A. Gans, "CAV Simulation Documentation," 22 December 2022. [Online]. Available:

https://uwaterloo.atlassian.net/wiki/spaces/UWAFT/pages/43347738813/CAV+Simulation +Documentation. [Accessed 25 June 2023].

[49] CARLA , "CARLA Documentation," June 2023. [Online]. Available: https://carla.readthedocs.io/en/latest/. [Accessed 25 June 2023].

[50] CARLA, "ScenarioRunner," CARLA, November 2022. [Online]. Available: https://carla-scenariorunner.readthedocs.io/en/latest/. [Accessed 25 June 2023].

[51] CARLA, "CARLA Autonomous Driving Challenge," CARLA, [Online]. Available: https://carlachallenge.org/challenge/nhtsa/. [Accessed 25 June 2023].

[52] C. Jung, D. Lee, S. Lee and D. Shim, "V2X-Communication-Aided Autonomous Driving: System Design and Experimental Validation," *Sensors,* vol. 20, no. 10, 2020.

[53] Watonomous, Watonomous, 2022. [Online]. Available: https://www.watonomous.ca/. [Accessed 30 June 2023].

[54] Gitlab, "The .gitlab-ci.yml file," Gitlab, September 2022. [Online]. Available: https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html. [Accessed 30 June 2023].

[55] S. Allwright, "What is a good F1 score and how do I interpret it?," 20 April 2022. [Online]. Available: https://stephenallwright.com/good-f1-score/. [Accessed 2 July 2023].

[56] R. Kundu, "Precision vs. Recall: Differences, Use Cases & Evaluation," 19 September 2022. [Online]. Available: https://www.v7labs.com/blog/precision-vs-recall-guide. [Accessed 2 July 2023].

[57] B. Ristic, B. -N. Vo, D. Clark and B. -T. Vo, "A Metric for Performance Evaluation of Multi-Target Tracking Algorithms," *IEEE Transactions on Signal Processing,* vol. 59, no. 7, pp. 3452-3457, 2011.

[58] D. Schuhmacher, B.-T. Vo and B.-N. Vo, "A Consistent Metric for Performance Evaluation of Multi-Object Filters," *IEEE Transactions on Signal Processing,* vol. 56, no. 8, 2008.

[59] T. Harbid, "How Long Is A Car? What Is The Average Length Of A Car?," Cash Car Buyer, 5 October 2020 . [Online]. Available: https://www.cashcarsbuyer.com/how-long-is-a-car-what-is-the-average-length-of-a-car/. [Accessed 2 July 2023].

[60] M. Alqarqaz, M. Younes and R. Qaddoura, "An Object Classification Approach for Autonomous Vehicles Using Machine Learning Techniques," *World Electric Vehicle,* vol. 14, no. 2, 2023.

[61] C. Lee, K. Kim and M. Lim, "Sensor fusion for vehicle tracking based on the estimated probability," *Intelligent Transportation Systems,* vol. 12, no. 10, pp. 1386-1395, 2018.

[62] S. Fernando, A. Khan, R. Fraser and W. Melek, "Structured Testing Framework for ADAS Algorithm Development," in *IEEE International Automated Vehicle Validation Conference 2023*, Austin, 2023.

[63] Willow Garage Inc., "Message Filters — Chained Message Processing," Willow Garage Inc., 2009. [Online]. Available: http://docs.ros.org/en/kinetic/api/message_filters/html/python/index.html. [Accessed 3 July 2023].

[64] Advanced Vehicle Technology Competitions, "EcoCAR EV Challenge Home," Advanced Vehicle Technology Competitions, 2022. [Online]. Available: https://ecocarevchallenge.org/ecocar-ev-challenge/. [Accessed 22 11 2022].

[65] T. Andrews, "Computation Time Comparison Between Matlab and C++ Using Launch Windows," California Polytechnic State University, San Luis Obispo, 2012.

Appendix A – Template Observers

Frequency Observer:

Used to ensure topic is published above an average frequency over the simulation.

This is useful to add to critical control signals are topics that are known to be slow to compute ensuring that upstream nodes are getting data at the expected rate.

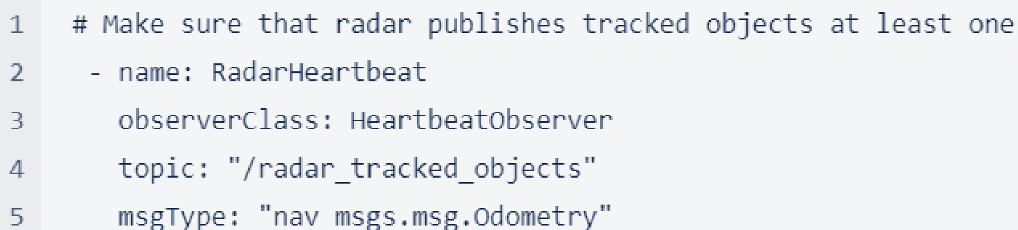
Example Schema:

```
1  #Ensures topic "/cmd_vel" is published at 10Hz (or more)
2    - name: CmdVelFrequency
3      observerClass: FrequencyObserver
4      topic: "/cmd_vel"
5      msgType: "geometry_msgs.msg.Twist"
6      minFreq: 10
```

Heartbeat Observer:

Used to ensure a topic is published at least once during a simulation. Generally used when you want to make sure the dependencies for a required topic have been met and it is able to publish something without caring about its contents.

Example Schema:



```
1  # Make sure that radar publishes tracked objects at least one
2   - name: RadarHeartbeat
3     observerClass: HeartbeatObserver
4     topic: "/radar_tracked_objects"
5     msgType: "nav_msgs.msg.Odometry"
```

In Range Observer:

Ensures that a numeric value is within an acceptable range. Paradigms of {`ALWAYS_TRUE`, `TRUE_ONCE`, `TRUE_AT_END`, `TRUE_AT_START`} apply and are used to determine the result.

Example Schema:

```
1  # Make sure cmd vel in x dir is always within [0, 10]
2  - name: CmdVelX
3      observerClass: InRangeObserver
4      topic: "/cmd_vel"
5      msgType: "geometry_msgs.msg.Twist"
6      field: "linear.x"
7      observerType: ALWAYS_TRUE
8      minVal: 0
9      maxVal: 10
```

Min Max Observer:

Similar to the in range observer but only checks value in one direction. (Value is always greater than min OR value is always less than max).

Example Schema:

```
1   # Ensure radar sequence number is always less than 120
2   - name: RaderObjectsIDObserver
3     observerClass: MinMaxObserver
4     observerType: ALWAYS_TRUE
5     topic: "/radar_tracked_objects"
6     msgType: "nav_msgs.msg.Odometry"
7     field: "header.seq"
8     value: 120 #in m
9     isMin: False #Max Value (if true becomes min val observer)
```

Monotonic Observer:

Used to ensure that a value is always increasing or decreasing throughout the simulation. Useful to determine if fragmented data is coming in order.

Example Schema:

```
#Ensure sequence number of radar msg is always increasing to confirm data i
- name: RaderObjectsIDObserver
    observerClass: MinMaxObserver
    observerType: ALWAYS_TRUE
    topic: "/radar_tracked_objects"
    msgType: "nav_msgs.msg.Odometry"
    field: "header.seq"
    isIncreasing: True #Check monotonic increase if false checks for decrea
```

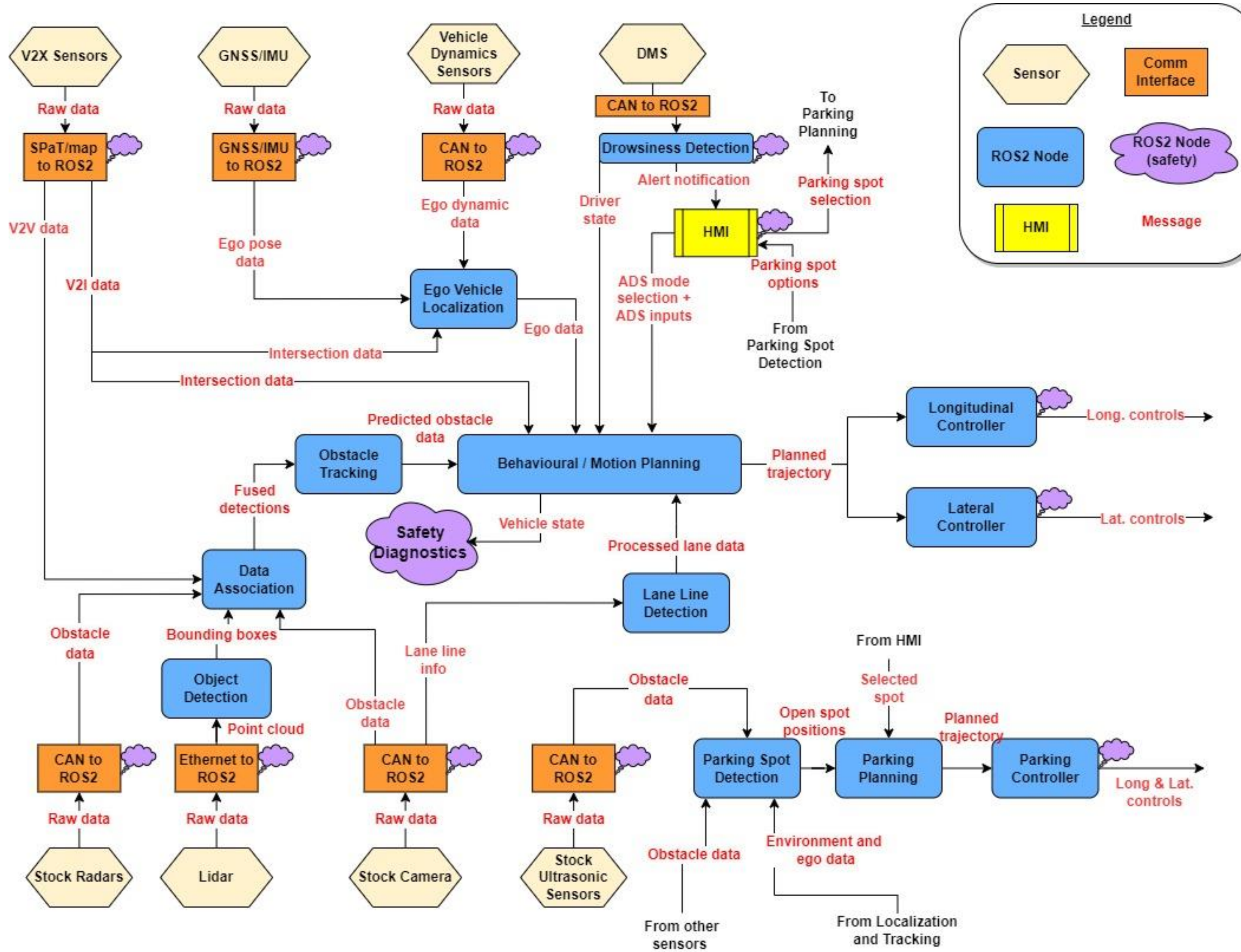
PrecisionRecall Observer:

Compares target topic to ground topic that is being published simultaneously. Uses dx/dy position locations and euclidean distance function for comparison

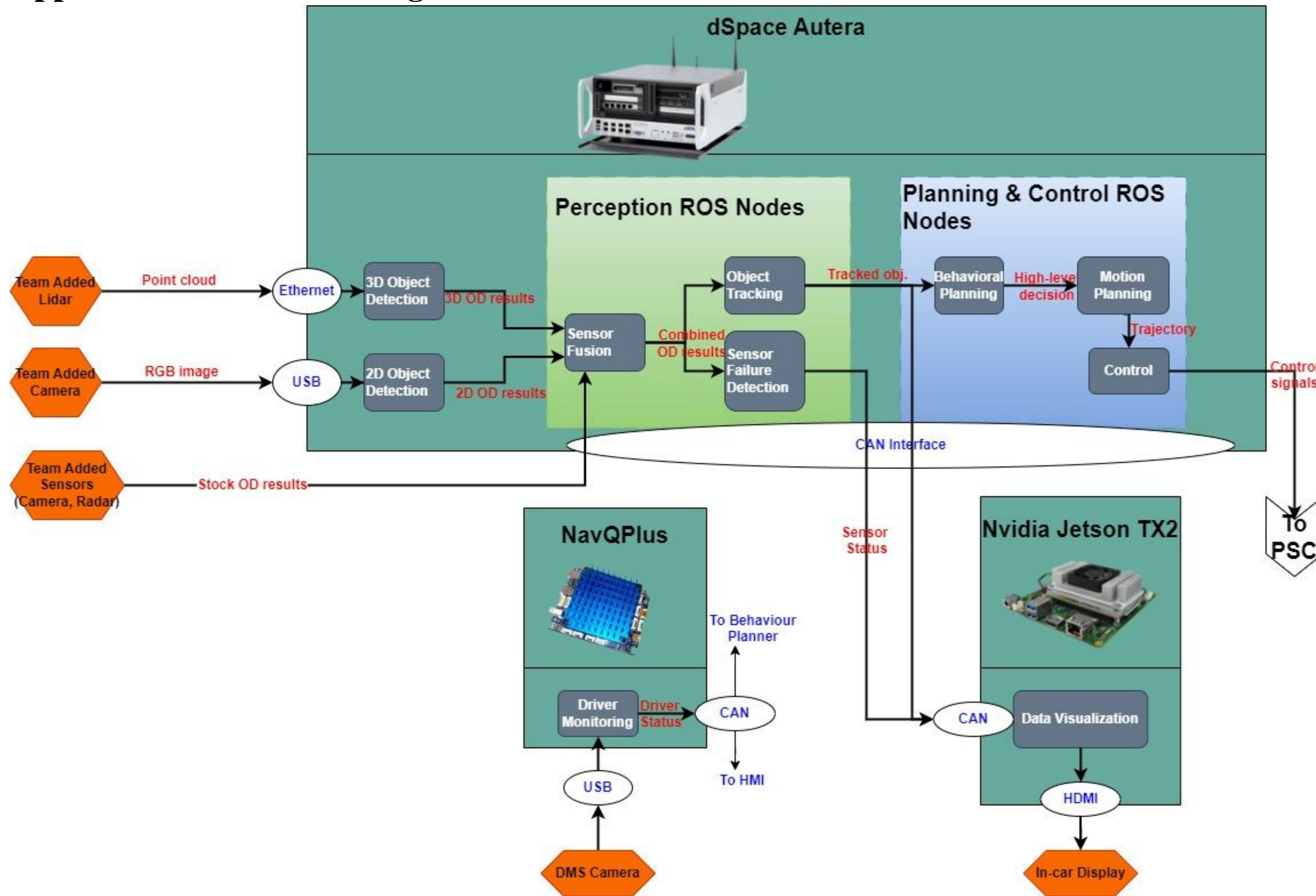
Example Schema:

```
- name: SystemPrecisionRecall
    observerClass: PrecisionRecallObserver
    topic:
      - "/tracked_obj" # Detection topic
      - "/Ground_Truth" # Ground truth topic
    msgType:
      - "common.msg.TrackedOutputMsg" # Detection message type
      - "common.msg.Ground truth" # Ground truth message type
    field:
      - "obj_dx" # Detection fields
      - "obj_dy"
      - "dx" # Ground truth fields
      - "dy"
    num_targets: 2 # actual number of ground truth objects
    output_type: fixed # whether output array is of a static size or variable
    tol: 4 # Tolerance between GT object and detection
    precision_pass_score: 0.6 # Minimum precision score that is deemed a PASS
    recall_pass_score: 0.6 # Minimum recall score that is deemed a PASS
```

Appendix B – Software Architecture Diagram



Appendix C – Network Diagram



Glossary

Bag file	File format used in ROS to save and playback topic data. In UW-STF it is used as the input for the software system under test (e.g. sensor data).
CARLA	Open source simulator for autonomous driving. Abbreviation here: **CARLA**.
F1-score	Harmonic mean of precision and recall. It is a measure of a model's accuracy.
Integration testing	Testing of a subset of nodes of the entire 'system'.
Launch file	File (can be in XML) used to configure and start up multiple nodes simultaneously.
Node	An executable which uses a ROS client library to communicate with other nodes. Data is transferred from one node to another via a publish-subscriber model of communication.
Observer	Name given to designated "monitor" node within UW-STF. When configured, they represent the different tests that can be run on nodes. Observers have the ability to monitor a topic, compare incoming data with expected values and provide a pass or fail status for the test.
RViz	3D visualization software for ROS.
SIT	ROS standard framework for integration testing as described in [24].
Stack	Components of a software system including the operating middleware, modules, and other libraries and tools that it is composed of.
System testing	Testing of the entire system (all nodes) as a whole.
Test runner	Tool that runs the commands when triggered as specified in a CI configuration file. These commands are usually related to building and running the source code so that this doesn't have to be done manually by a developer.
Topic	Name given to a communication bus from which data can be exchanged. The topic from which multiple nodes subscribes from and publishes to must be the same in order for the data to be exchanged.
Unit testing	Testing of individual functions or methods within a node.

.sim file A YAML file that defines how the University of Waterloo Structured Testing Framework runs. All important parameters for a suite of tests are configured in this file including the launch file(s), topic data, ***Bag file***(s), test duration, and Observers.

.simresults file A YAML file automatically generated at the end of the University of Waterloo Structured Testing Framework's execution. Each file corresponds to one .sim file and it is given the same name except with the *.simresults* extension.