A Platform for Generating Anomalous Traces Under Cooperative Driving Scenarios

by

Donghyun Shin

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

Waterloo, Ontario, Canada, 2018

© Donghyun Shin 2018

I hereby declare that I am the sole author of this 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.

Abstract

As we allow more critical decisions to be made by software, it becomes increasingly necessary to ensure the decisions made are correct. One approach is to monitor the software for signs of an anomaly. The approach is an active area of research with many proposed methods. For validating anomaly detection techniques, two popular approaches exist: Using an existing data trace, such as Knowledge Discovery and Data (KDD) Cup 1999 data, or injecting attacks into a tool used in industry. Both approaches lack flexibility; using existing traces constrain the validation to the way they are captured, while injecting attacks into an industry tool may require prior knowledge or characterization of it. A well-characterized platform that is built for trace generation would address the flexibility problem. On the other hand, autonomous driving is a field that demonstrates the criticality of decisions software discussed in the beginning. Specifically, cooperative autonomous driving scenarios, due to interactions between cars, can generate complex traces that would be of interest for researchers seeking to validate their anomaly detectors.

In this thesis, we propose a cyber-physical system for collecting traces for testing anomaly detectors, based on cooperative autonomous driving. We provide the design and implementation of the proposed system. The three-tier design of the system allows researchers to generate different traces by extending the system on different levels, from different control and estimation methods to new cooperative driving scenarios. It also provides a suite of tools for introducing anomalies and collecting traces. A Failure Mode Effects Analysis (FMEA) of the system is done to guide the creation of new anomalies. Finally, traces generated by the proposed system is used on an existing anomaly detector to verify its usefulness while the trace tools and anomaly injection tools are tested for its interference with the main system.

Acknowledgements

I would like to thank my supervisor Sebastian Fischmeister for support giving me an opportunity to work on such a unique system. Your unwavering vision and enthusiasm inspired me to pull through when I felt lost.

Dedication

To my family, whom I cannot express enough gratitude for supporting me, no matter what decision I made.

To my significant other, Hajra, for being my other half.

Table of Contents

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Validating Anomaly Detection Algorithms 2

1.2 Motivation 2

1.3 Contributions 3

1.4 Organization of the Thesis 3

2 Requirements 4

2.1 Required Properties 4

2.1.1 Flexibility 4

2.1.2 Consistency 5

2.1.3 Robustness 5

2.1.4 Semblance to Reality 5

2.2 Functional Requirements 5

2.2.1 Characterization 5

2.2.2 Annotated Anomaly Injection 6

2.2.3 Trace Generation and Conversion 6

3 System Architecture **7**

3.1 Robot Operating System (ROS) 8

4 Main Platform **9**

4.1 Platform Layer 10

4.1.1 Obstacle Platform 11

4.1.2 Conveyor Platform 11

4.1.3 Car Platform 12

4.2 Model Layer 13

4.2.1 Conveyor Model 13

4.2.2 Car Model 14

4.2.3 Threat Model 16

4.3 Apps Layer 17

4.4 Notable Details 20

4.4.1 Car Consistency 21

4.4.2 Battery Depletion 22

4.4.3 Car Dynamics 26

4.4.4 State Estimation 28

5 Trace Tools **30**

5.1 Implementation 31

5.2 Proposed Workflow 32

6 Anomaly Injection Framework **33**

6.1 Anomaly Characterization 33

6.2 Anomaly Injection Suite 34

6.3 Example Implementations 36

7 Case Study 38
7.1 Inter-Arrival Curves Based Anomaly Detection 38
7.2 Setup 39
7.3 Result 39

8 Evaluation 41
8.1 Objectives 41
8.1.1 Trace Tools 41
8.1.2 Anomaly Injection Suite 41
8.2 Experimental Setup 42
8.2.1 Baseline 42
8.2.2 Trace Tools: Redis Streaming 43
8.2.3 Trace Tools: Delay 43
8.2.4 Trace Tools: Man-in-the-Middle 43
8.3 Results 43
8.3.1 Baseline 43
8.3.2 Trace Tools: Redis Streaming 44
8.3.3 Trace Tools: Delay 46
8.3.4 Trace Tools: Man-in-the-Middle 47

9 Future Work and Conclusion 48
9.1 Future Work 48
9.1.1 Dedicated Car Hardware 48
9.1.2 Improved Car Control 49
9.1.3 Obstacle State Estimation 49
9.2 Conclusion 50

References 51

APPENDICES **53**

A Car Circuit Board Schematic **54**

B FMEA **63**

List of Tables

6.1 An example FMEA row, transposed to fit the page 34

7.1 An example of counting inter-arrival for event c 38

8.1 Baseline latency values 44
8.2 Baseline resource usage 44
8.3 Redis latency values 45
8.4 Redis overhead values 45
8.5 Redis resource usage 46
8.6 Delay latency values (N = 2) 46
8.7 Delay resource usage (N = 2) 47
8.8 MitM latency values (N = 1) 47
8.9 MitM resource usage (N = 1) 47

List of Figures

3.1 High-level overview of the system 7

4.1 Hardware overview 9
4.2 The platform layer 10
4.3 Occupancy grid of obstacles, superimposed over frame data 11
4.4 The model layer 13
4.5 Control loop for the conveyor 14
4.6 Control loop for the longitudinal control of the car 15
4.7 Kinematic bicycle model for lateral control of the car 16
4.8 Threat model 17
4.9 The apps layer 17
4.10 Added visualization 18
4.11 Platooning 19
4.12 Obstacle avoidance calculation 20
4.13 Previous iteration car 21
4.14 Current iteration car 22
4.15 Battery discharge curve 22
4.16 Naive and normalized throttle versus time 23
4.17 Naive and normalized residual plotted versus time 24
4.18 Battery voltage versus time 25
4.19 Normalized values for a full battery after preheating 26

4.20 An oversteering bicycle model 27
4.21 One Euro filter 28

5.1 Trace tools workflow 32

6.1 Injection of a Man-in-the-Middle node 35
6.2 Dead spot 36

Chapter 1

Introduction

We are living in a world where many decisions are made for us by some form of software. At the same time, the criticality of said decisions is increasing. Autonomous driving is one example. As of October 2017, 67 cities worldwide are, or planning on, allowing companies to test their autonomous driving technologies on public roads; further, 33 cities are actively evaluating the impact of autonomous vehicles on the road [4]. Another recent example is Food and Drug Administration (FDA) allowing software to make a screening decision for diabetic retinopathy patients, whether or not they should see an eye care professional [10]. The stakes involved in such decisions are apparent, as making a wrong decision can cost people's health or even their lives.

Given the criticality of decisions modern software makes, it seems crucial to verify that it is behaving correctly. There exist many proposed methods, but one widely adopted by the industry is software testing. Test-driven methodologies do tend to improve code quality [3]; however, proving their coverage remains an open problem [2]. Another approach is model checking, where a finite software model representing the code is built, and then checked against properties to enforce [7]. In this approach, managing the number of states in a model, which can be enormous for real-world software, is an open problem that is an active research topic. It seems that, despite the need to ensure the correctness of software, there is no sure way to do so. The presence of third-party software that may potentially be closed-sourced can further complicate the issue.

Another approach, then, is used to replace often or complement methods above, such as Runtime monitoring. As its name implies, it is used to verify assumptions about the system in real time. For example, a runtime monitor may be used in conjunction with model checking to verify assumptions made in the model.

An instance of runtime monitoring is anomaly detection. This approach examines the data produced by the program, trying to verify that the data observed is as expected. While anomaly detection algorithms tend to scale better, and generally does not require access to the program's source, it has its own set of challenges.

1.1 Validating Anomaly Detection Algorithms

Once researchers implement an anomaly detection algorithm, it must be verified. The verification can happen in many ways. One technique is to utilize a previously collected and annotated trace. Example datasets include the KDD Cup 99 data [23] used for verifying unsupervised intrusion detection [9], or annotated electrocardiogram and sensor data used for validating time series anomaly detection [15]. A problem with using such dataset is that the definition of "normal" behavior may change over time, obsoleting the dataset [6].

Another method often used for validating an anomaly detection algorithm is to take advantage of existing code. Some researchers have previously used Java Secure Sockets Library (JSSL), as-is [13], while others inject known attacks into tools such as dhcpd. Unfortunately, such methods require an in-depth, or prior characterization of the subject code.

1.2 Motivation

As mentioned in Section 1.1, challenges in finding a useful data for validating anomaly algorithm are as follows. A dataset may age and become obsolete. Some anomalies may become less relevant as new ones are introduced. An already created dataset, by itself, cannot accommodate such change over time. Further, the dataset may not be in a suitable form for the anomaly detection algorithm of choice. Finally, the trace may turn out to not represent the real-life anomaly scenario accurately.

The challenges above highlight an opportunity for a platform specifically designed and built for validation of anomaly detection algorithms. Such platform would provide a flexible data source that can adapt for changing definitions of "normal" over time while providing a guide on adding new anomalies. Any issue with a dataset can be addressed by merely generating a new dataset.

At the same time, the proposed system would ideally represent real-world scenarios, connecting the anomaly detection algorithms to practical applications. In this regard,

autonomous driving seems to be a suitable choice; a wrong decision by the software would result in an accident that could claim people's lives. Cooperative autonomous driving is especially impressive, as the interaction between cars can generate complex traces. The simulation would involve a cyber-physical component, introducing physical realism, and an illustrative platform for real-time demonstrations.

1.3 Contributions

In this thesis, we describe a cyber-physical system that can be used to help validate anomaly detection algorithms. It includes a platform for simulating autonomous cooperative driving scenarios as a source for non-trivial traces. The platform has a three-tier architecture, guiding researchers in introducing changes of different degrees, from changing the control methodology to introducing a new cooperative driving scenario. The system also provides a framework for injecting anomalies into the platform. It consists of a characterization document for guiding the creation of anomalies, and a software suite to help implement them. Finally, the system provides a dedicated set of tools for exploring and streaming generated traces.

1.4 Organization of the Thesis

The thesis is organized as follows. Chapter 2 discusses the guiding requirements of the system. Chapter 3 provides a system-level architecture overview. Chapter 4 provides a brief design overview of the main platform, then delve into details that impacted the system's adherence to the requirements. Chapter 5 and 6 discuss designs of the trace tools and the anomaly framework. Then, Chapter 7 validates the system through a case study by using the trace generated by it to detect injected anomalies. Chapter 8 evaluates the performance of the trace collection and injection infrastructure. Lastly, Chapter 9 discusses future work that can improve the platform.

Chapter 2

Requirements

As mentioned in Section 1.2, the primary goal of a candidate system is to accommodate for anomalies injected into different cooperative driving scenarios, generating various traces for use with the anomaly detection algorithms. This chapter describes the requirements introduced to guide the design and implementation of the system towards the goal. The requirements have two categories: Required properties, and functional requirements.

2.1 Required Properties

This section discusses properties required of the proposed system. These properties are desired for a system that can generate traces useful for validating anomaly detection algorithms.

2.1.1 Flexibility

If the system cannot handle new driving scenarios and anomalies, it does not confer any advantage over existing methods for testing anomaly detection algorithms. Therefore, the system must be modular enough for researchers to introduce new behaviors into the system without changing many parts of it.

2.1.2 Consistency

Most anomaly detection algorithms must first determine a baseline behavior that is considered normal. The algorithm's classification of abnormal behavior depends on this baseline. Subsequently, if the system shifted the baseline behavior over time, or over multiple runs, it would cause false positives, rendering it useless. Presence of hardware components makes it harder to guarantee such consistency, with manufacturing uncertainties and observation noises.

2.1.3 Robustness

While the impact of a misbehaving system based on a pure software is limited, such is not the case for a cyber-physical system; any misbehavior may cause damage to hardware that may need replacing. Worse yet, the damaged hardware may behave differently, which would have a negative impact on the system's consistency.

2.1.4 Semblance to Reality

The implementation of the proposed system is a scaled simulation. Such nature allows only a limited simulation of actual cooperative driving scenarios. However, the proposed system must make best efforts to bear some resemblance to reality. A system that presents a radical departure from reality is less attractive for researchers, especially compared to using existing tools or libraries.

2.2 Functional Requirements

The properties above let the proposed system generate data suitable for anomaly detection algorithms. However, the proposed system must also provide functions to access data and introduce anomalies into the system.

2.2.1 Characterization

The proposed system is designed to produce non-trivial traces; that is, traces that are generated by interacting components. It follows, then, that the implementation of such

system is non-trivial. Without any guidance, introducing new anomalies into the system may require an in-depth analysis of the system; this defeats the perceived advantage of the system over using existing traces or tools. Therefore, the system must provide some form of documentation describing its behavior, especially under anomalous circumstances.

2.2.2 Annotated Anomaly Injection

Since the proposed system focuses on verifying anomaly detection algorithms, it must provide facilities to inject anomalies. A dedicated facility would prevent ad-hoc changes, which would affect the reliability of the system. Further, the generated trace must have an annotation indicating if the anomaly is active, as many anomaly detection algorithms use labeled data for training.

2.2.3 Trace Generation and Conversion

A complex cooperative driving scenario with injected anomalies would have little meaning if users cannot collect data. To this end, the proposed system must provide tools to collect data from the live system. Further, it must support conversion of the data for three different use cases: Exploratory analysis, offline verification, and online verification. Extracting a small set of CSV files from a large trace would be useful for exploratory analysis, where being able to stream data real-time would be used for online verification.

Chapter 3

System Architecture

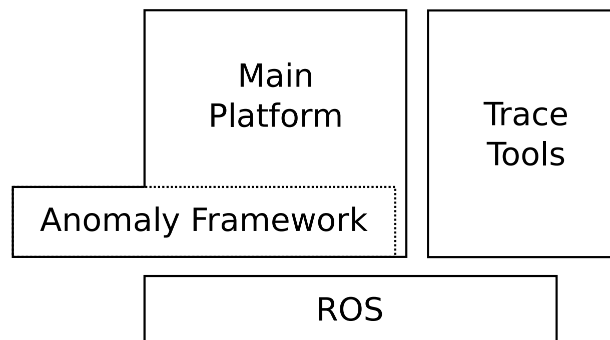


Figure 3.1: High-level overview of the system

The system utilizes ROS, a publish-subscribe framework designed for robotics [19]. ROS has multiple properties that ease the effort for satisfying the requirements. These properties are discussed in Section 3.1. The proposed system consists of three parts: The main platform, trace tools, and the anomaly injection framework. The structure is shown in Figure 3.1. The main platform is the environment for running autonomous cooperative driving scenarios. The trace tools allow researchers to collect and convert data as needed. It also provides a tool to transform the collected data into a ROS-independent format. Finally, the anomaly framework consists of two parts. The anomaly injection suite is a toolset for introducing a shim into the main platform to introduce anomalies. The suite also contains a characterization document to guide the creation of new anomalies.

3.1 Robot Operating System (ROS)

ROS is a framework of tools and libraries for writing robot software [19]. At its core, ROS allows users to implement a publish-subscribe infrastructure. A system implemented using ROS consists of many processes (or threads), called nodes (or nodelets). The network of nodes communicates by subscribing or publishing to a set of topics. For example, a node that controls a car would subscribe to topics that provide current position, velocity, and the goal, using them to publish appropriate throttle and steering values to a topic that would be subscribed to by another node that interfaces with the car hardware. Such design allows changing behavior by providing a new node that publishes and subscribes to same sets of topics the node it is replacing. As well, ROS provides both C++ and Python bindings, allowing researchers to use Python for fast prototyping.

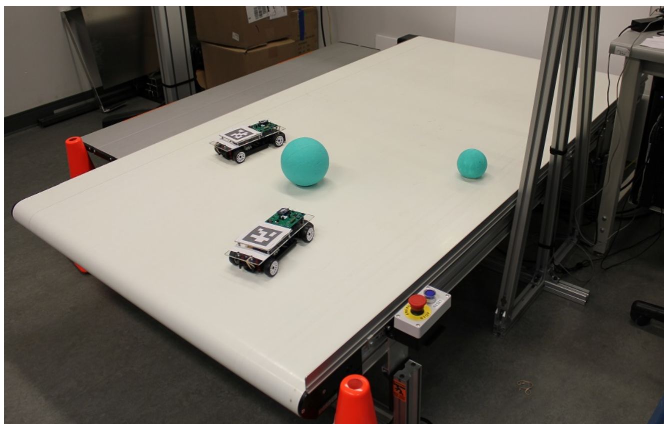
All topics in ROS are public. Any node is free to publish or subscribe to any topic. As long as the system is broken down into a smaller set of nodes implementing parts of behavior, gathering insight into the system becomes the matter of examining the right topics. Additionally, ROS provides a command line tool to record specified topics in a binary format, called "bag." However, researchers would need to install ROS to interact with the file.

Finally, regarding anomaly injection, the publish-subscribe infrastructure shifts the burden of analyzing the code to the topics, and how they interact. Anomaly injection, then, concerns manipulating the messages from the topics, such as inducing a delay into the car's position data. Therefore, characterization of the system focuses on topics, not code.

There are features of ROS that make it suitable for anomaly injection. First, as mentioned, all topics are public, and therefore eavesdropping and injecting messages are trivial. Second, ROS establishes one TCP connection for each publisher-subscriber pair, enabling the use of well-established network testing tools for anomaly injection. Finally, names of topics a node publishes and subscribes to can be remapped into other names when the node launches. Section 6.2 discusses how these features are exploited to provide the anomaly injection framework.

Chapter 4

Main Platform



(a): Conveyor with cars and obstacles

(b): Top mounted equipments

Figure 4.1: Hardware overview

The main platform, hardware-wise, is actualized as a conveyor belt simulating an infinitely long highway. Modified off-the-shelf remote-controlled cars run on the conveyor, with a top mounted camera tracking their positions. Finally, another camera tracks obstacles on the conveyor. Figure 4.0(a) and 4.0(b) show the hardware setup. Most software components reside within a high-performance PC.

The platform's software stack has three layers: platform, model, and the apps layer. The model layer provides the "world" that upper layers can observe and manipulate. The platform layer contains the control and estimation components that provide a higher level view into the world. Finally, the apps layer comprises high-level functionalities that

control the overall behavior of the system, such as different driving scenarios, visualizers, and monitors to prevent hardware damage.

4.1 Platform Layer

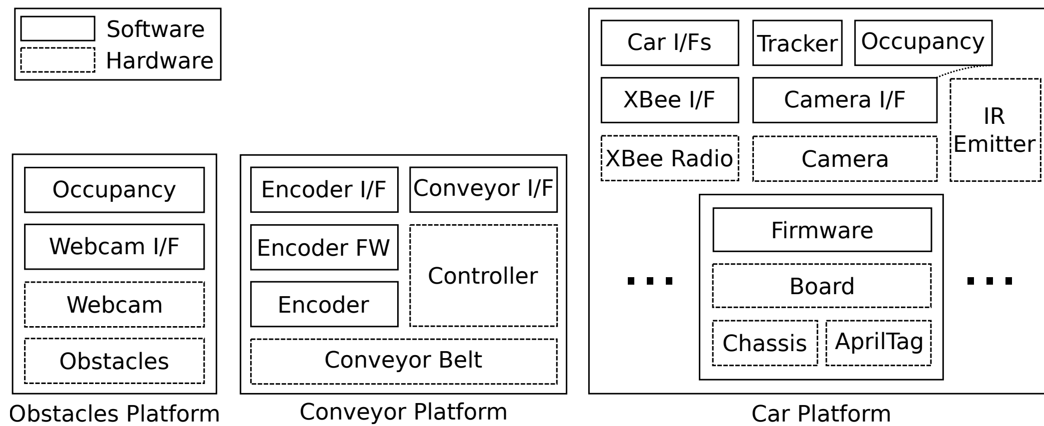


Figure 4.2: The platform layer

The platform layer provides the system with ways to observe and manipulate the world at a low level. Changing the underlying hardware or implementing a simulation of the hardware would require changes in this layer. It divides into three components, named after the physical environment they observe and manipulate: Car, Conveyor, and Obstacle. Block diagram for the layer is in Figure 4.2.

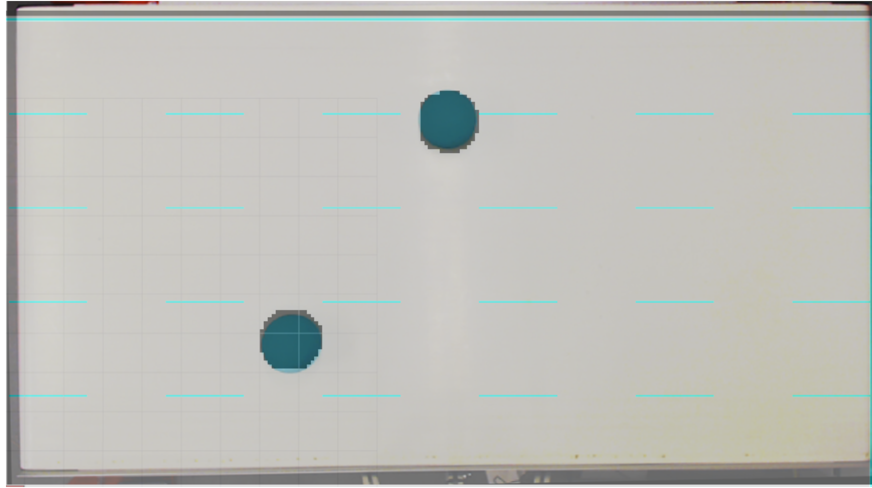


Figure 4.3: Occupancy grid of obstacles, superimposed over frame data

4.1.1 Obstacle Platform

In the system, an obstacle is defined to be an object of the specified color. Color-based obstacle classification is a departure from reality; however, the system's purpose is not to implement a sophisticated perception algorithm but to use this data for cooperative driving. To this end, the realism tradeoff is deemed acceptable. The obstacle layer captures the top-down view of the environment, then builds an occupancy grid based on it. Occupancy grid is a 2D matrix of integers representing the environment, where each cell represents the probability of the corresponding location being occupied. The occupancy grid is shown superimposed over the camera frame data in Figure 4.3. By definition, obstacles are outside of the system's control and are not manipulable through the system, unlike car and conveyor platforms.

4.1.2 Conveyor Platform

Manipulating the conveyor platform involves the conveyor assembly and the motor controller. Both are standard, off-the-shelf industrial products. The motor controller allows control of the conveyor through Ethernet. The software side then uses the Ethernet interface to set the throttle, given in percent, provided by the upper layer.

The conveyor velocity is observed through a custom encoder assembly, connected to a microprocessor platform to count the pulses from it. The microprocessor connects to the

PC, sending the pulse count periodically. The software interface uses the pulse count to calculate unfiltered velocity values.

4.1.3 Car Platform

Manipulating and observing the cars involve two components: Tracking and communications.

The tracking component allows the upper layers to observe the cars by uniquely identifying each car and providing its position and orientation, also known as the pose. AprilTag [18] is mounted on each car for this purpose. However, the tracking works in the Near-Infrared (NIR) light range; a camera, equipped with a NIR pass filter, and an IR emitter are mounted close to each other at the top. This setup allows consistency over differing light conditions. The IR emitter illuminates the scene. The tags are constructed with retroreflective material, reflecting high-intensity light to the camera. The software layer identifies these high-intensity regions as regions of interest. The AprilTag tracking algorithm is applied to the regions to provide pose data of each car. Another node uses pose data to build an occupancy grid for cars.

The communication component uses XBee, allowing the upper layers to control the car by giving command values. Command values consist of throttle, in percent, and steering, in radian. XBee is used for wireless communication, as its frequency hopping functionality allows robust data exchange in a saturated environment. The commands are mapped to integers understood by the lower-level firmware. XBee interface collects commands for all cars and broadcasts them using the base station XBee. Then, the firmware on car's custom circuit board directly applies the given command value and replies with the battery voltage level. The schematic for the circuit board exists in Appendix A. The reply is processed by the XBee interface, which makes the voltage value available on ROS.

4.2 Model Layer

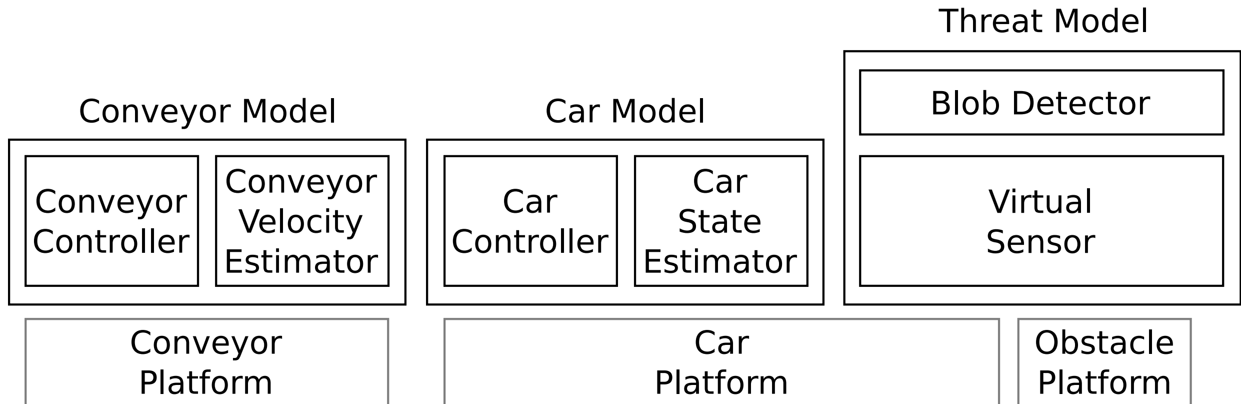


Figure 4.4: The model layer

The model layer is implemented above the platform layer and is shown in Figure 4.4. It serves two purposes: To refine the observations given by the platform layer and to provide a high-level control to the upper layer by closing the loop. The model layer divides into three parts: The conveyor model, car model, and the threat model. Changing this layer would cause changes in the per-car driving behavior and perception.

4.2.1 Conveyor Model

The conveyor model smooths raw velocity values and uses it to implement a velocity control loop. For smoothing the velocity, the system employs a simple, discrete moving average filter:

$$v_{filtered}[n] = (1 - \alpha) * v_{filtered}[n-1] + \alpha * v_{unfiltered}[n]$$

Where $0 < \alpha \leq 1$ is the filtering coefficient. As α approaches zero, the smoothing becomes more aggressive; however, it also makes the signal lag behind the actual value. The lag is acceptable, as the conveyor velocity changes slowly.

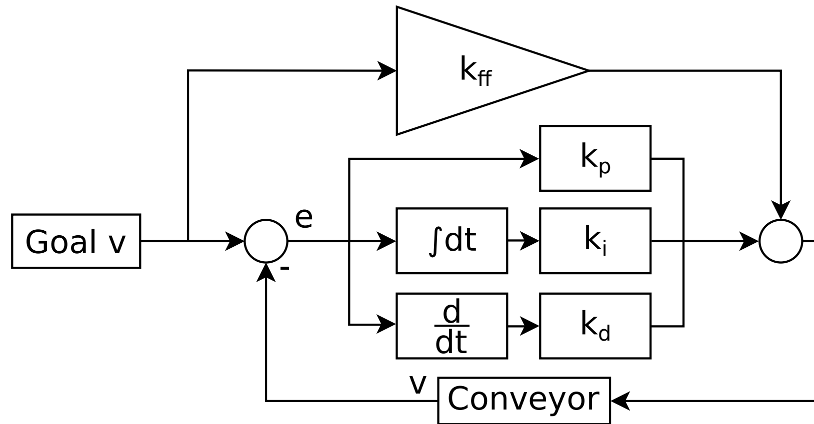


Figure 4.5: Control loop for the conveyor

The velocity loop uses values from the above filter to implement a Proportional-Integral-Derivative controller, as shown in Figure 4.5. The PID controller calculates the error; that is, the difference between the desired velocity, provided by the upper layer, and the actual velocity. A portion of the error (Proportional), the rate of change of it (Derivative), and the accumulation (Integral) of it are used to calculate the appropriate control signal. PID controller, by itself, will eventually provide an appropriate control signal to track the desired velocity; however, such design relies on the accumulation of the error, which slows the response time. Therefore, a feedforward gain, k_{ff}, is introduced to decrease the response time of the control loop.

4.2.2 Car Model

The car model works similarly to the conveyor model. Raw pose values are smoothed and then used to implement a control loop.

Unlike the conveyor velocity, the car's pose values can have a substantial change over a short time. Therefore, the system uses a different filter, called One Euro Filter [5]. The details are discussed in Section 4.4.4. The smoothed pose values are derived to obtain the velocity values.

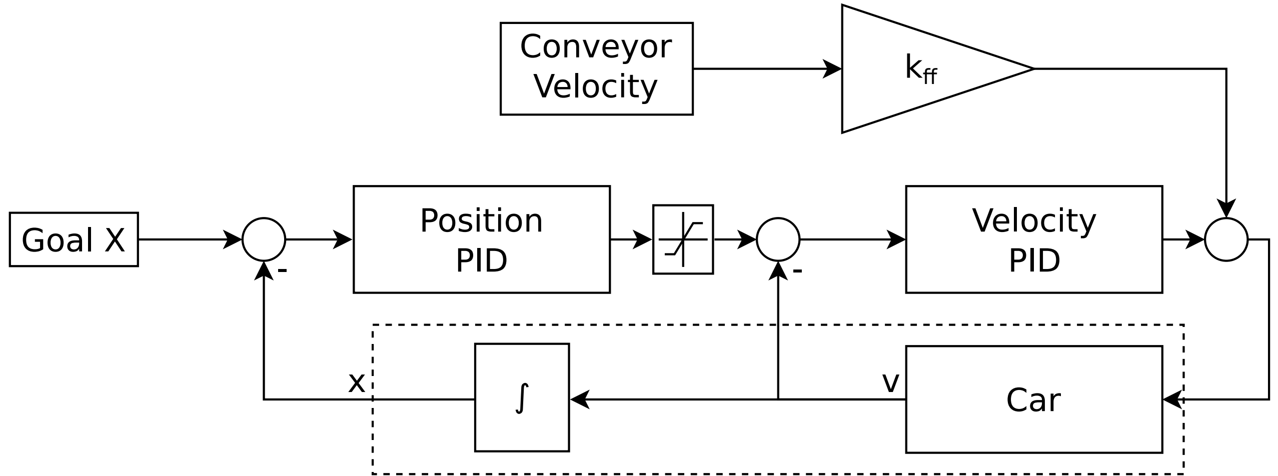


Figure 4.6: Control loop for the longitudinal control of the car

The car's longitudinal and lateral control are separate implementations. The longitudinal controller commands the car's throttle to move the car to the correct longitudinal position. It is implemented by two cascaded PID controllers, as shown in Figure 4.6.

At first, only one controller was used to generate the throttle value from the position error. However, this strategy led to a high throttle output when the longitudinal position error was significant. This unrealistic, as drivers generally respect speed limits, regardless of how far they are from the destination. Since the throttle is proportional to the acceleration, there is no intuitive way to clamp it. Therefore, the control loop divides into two PID controllers. The first PID controller uses the position error to calculate the velocity to approach the goal point. Then, this value is clamped to a configurable range. Finally, another PID controller calculates the throttle to achieve the desired velocity.

The lateral controller is a non-linear controller named Stanley controller [22]. The implementation follows the implementation by Autonomoose [8], University of Waterloo's full-scale autonomous vehicle project. Stanley Controller is a kinematic controller. So, it does not consider the dynamics in trying to keep the car on a defined path. The vehicle is simplified to a bicycle, as shown in Figure 4.7. δ denotes the steering angle of the car. Stanley controller attempts to drive two errors, e_h and e_c, to zero by commanding δ. By driving e_h, the heading error, towards zero, the vehicle heads parallel to the path. By driving e_c, the cross-track error, to zero, the vehicle goes on the path. Implementation details and tradeoffs made are discussed in Section 4.4.3.

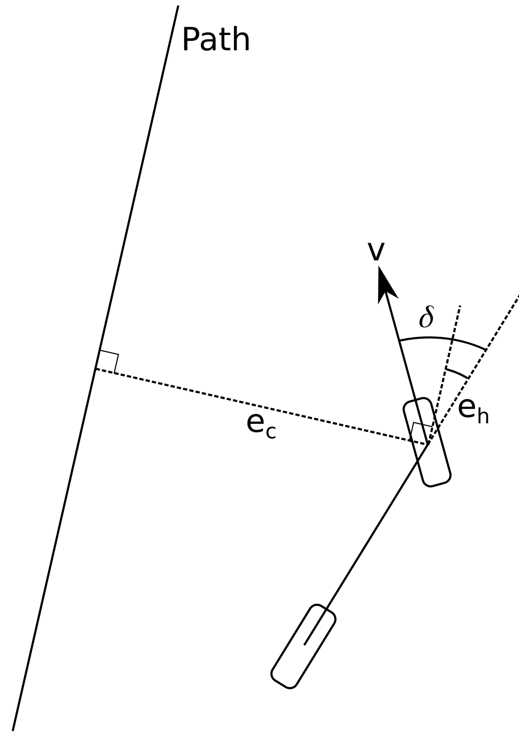
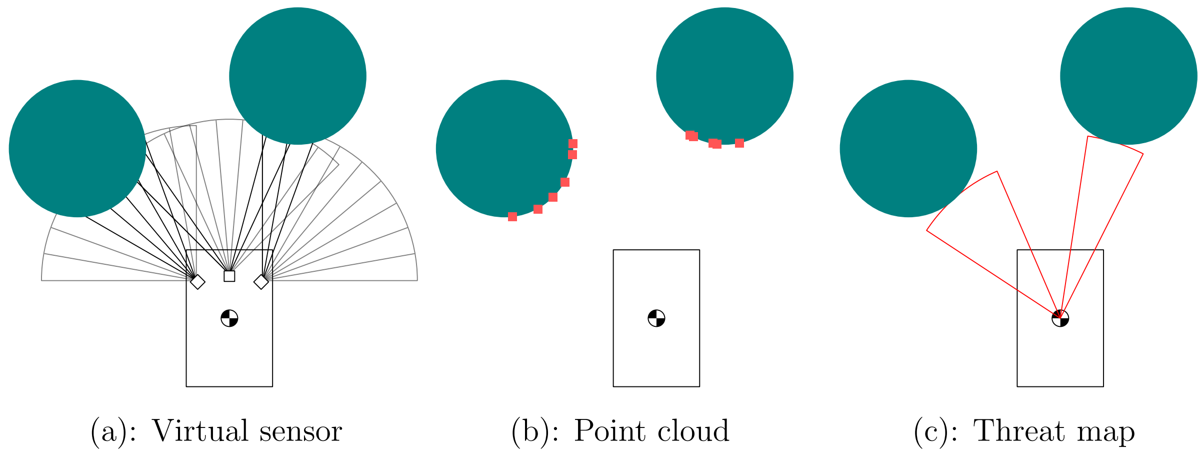


Figure 4.7: Kinematic bicycle model for lateral control of the car

4.2.3 Threat Model

As-is, the platform layer provides a top-down view of obstacles and cars on the conveyor. However, this is unrealistic as no car has such view. The virtual sensor provides a degree of realism by transforming the top-down view. It combines the pose of the car and the mounting position of the sensor to project scan lines in the occupancy grid, generating a LIDAR-like scan, as shown in Figure 4.7(a). An arbitrary number of differently configured virtual sensors can exist on a car. Blob detector combines readings from all virtual sensors, generating a point cloud shown in Figure 4.7(b). Then, neighboring points are combined to create a threat map around the car, providing the angle, size, and distance of each blob around the car, as shown in Figure 4.7(c).



(a): Virtual sensor (b): Point cloud (c): Threat map

Figure 4.8: Threat model

4.3 Apps Layer

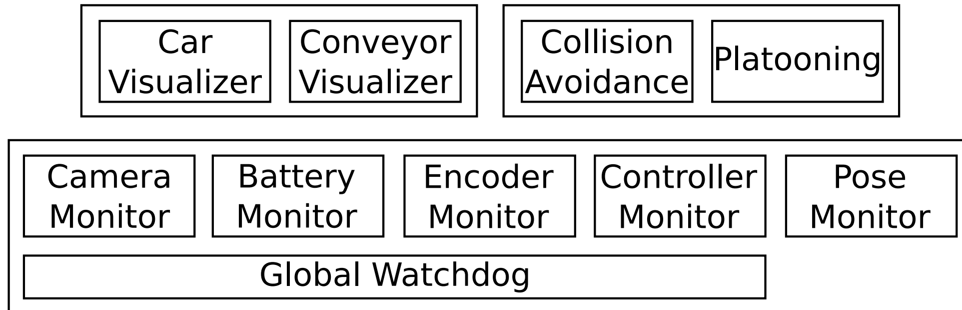


Figure 4.9: The apps layer

The apps layer utilizes the information to the two lower layers to dictate or demonstrate the high-level system behavior. There are currently three significant components: Visualizers, monitors, and scenarios. These components are shown in Figure 4.9. Change in this layer would cause changes in the high-level behavior, such as driving strategies.

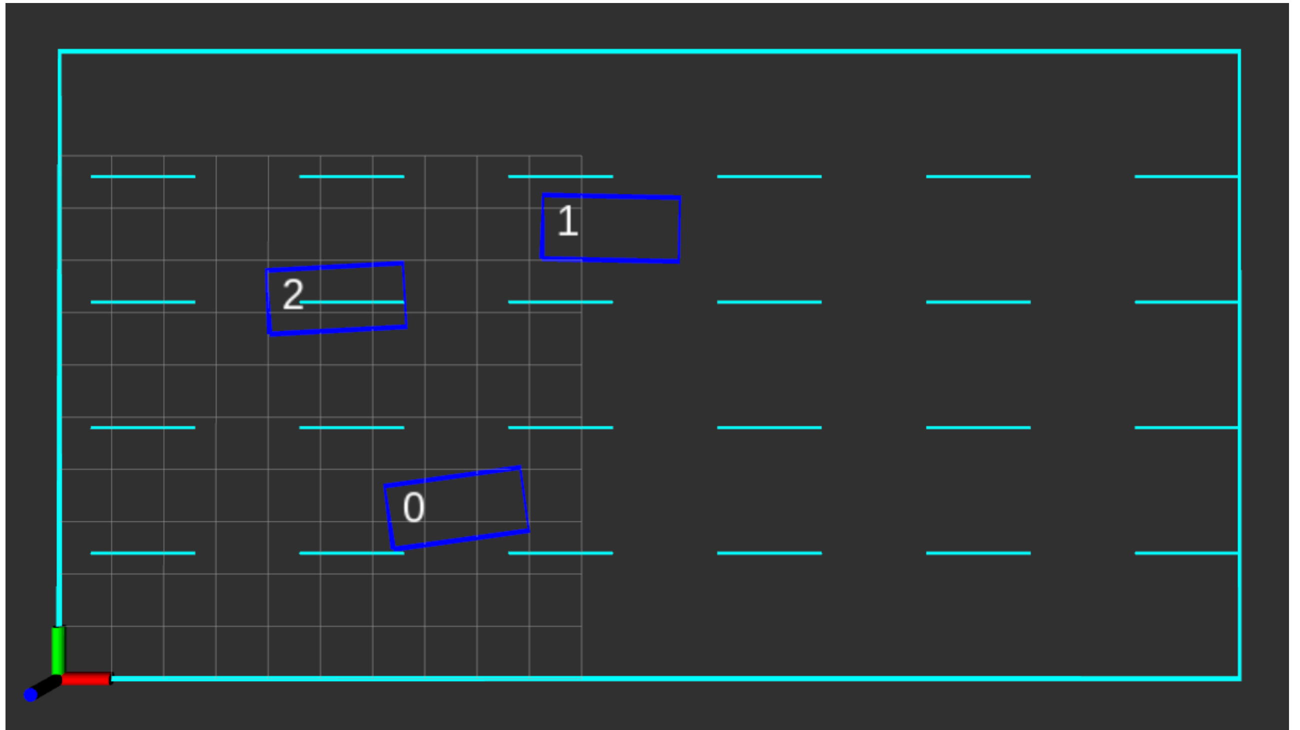


Figure 4.10: Added visualization

The visualizers act to extend the visualization available in RViz, a graphical application included with ROS. While low-level details, such as poses of each car, and obstacle and car occupancy grids, can be visualized on RViz by default, details specific to the system, such as each car's bounding box and lane markers of the conveyor are not. The added visualizations are shown in Figure 4.10.

The monitors check the system for unusual activities that may cause damage to the system. Critical monitors check the system for error conditions that may cause a car to fall off the conveyor or otherwise cause unrecoverable damage to the system. They report to a watchdog, which has direct control over the platform layer. The watchdog expects a periodic safety report from the critical monitors; if a report is missing, or an unsafe condition is reported, the system is in a fault condition. Then, the watchdog issues an overriding command to the platform layer, causing all hardware to go into the idle state.

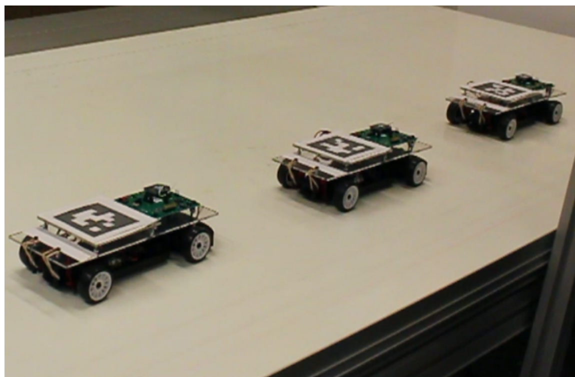
An example of a critical monitor is the battery monitor. Overdischarging the battery damages it irreversibly; therefore, voltage values produced by the platform layer ensures that it is above a configurable threshold value. Missing voltage values will also cause a fault. Other critical monitors verify the periodicity of data produced by hardware and the

control signals.

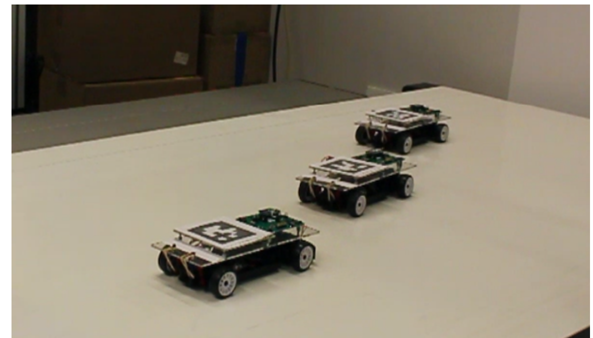
The monitor that checks the periodicity of pose data produced is non-critical, as taking a car off the conveyor is not a critical failure. The monitor sets the command values of the car missing pose updates to idle, preventing the vehicle from moving outside the conveyor.

Finally, the scenario layer implements high-level driving scenarios. Two provided scenarios are platooning and obstacle avoidance.

The platooning scenario utilizes the threat model. It examines the threat map of the car and chooses the object in front as a platooning target. The car aligns itself to the platooning target laterally. The longitudinal distance is maintained by a non-linear controller, issuing a highly aggressive backoff behavior in case the follow target slows down drastically. The car at the front is the lead car and is controlled manually through a controller input. Three cars in a platoon are shown in Figure 4.10(a). Because no car communicates directly with the lead car, lateral and longitudinal changes are propagated with visible lag, as shown in Figure 4.10(b).



(a): Default behavior



(b): Lag in lateral position propagation

Figure 4.11: Platooning

Another implemented scenario is obstacle avoidance. The implementation utilizes a blob detector and implements a behavior like the potential field path planning method. In this method, obstacles close to the vehicle exert a repulsive force. The implementation picks the closest obstacle, and calculate forces that three vectors. The first two vectors move perpendicularly to the obstacle. The last vector, on the other hand, points away. Their magnitudes depend on the distance to the obstacle. The forces are shown in Figure 4.12. The perpendicular vectors are denoted as $v_{perp,1}$ and $v_{perp,2}$, and the away factor is denoted as v_{away}. The perpendicular vectors and the away vector are added to generate two candidate vectors, $v_{cand,1}$ and $v_{cand,2}$. Finally, the candidate vector that directs away

from most other obstacles is selected. In Figure 4.12, this is $v_{cand,1}$.

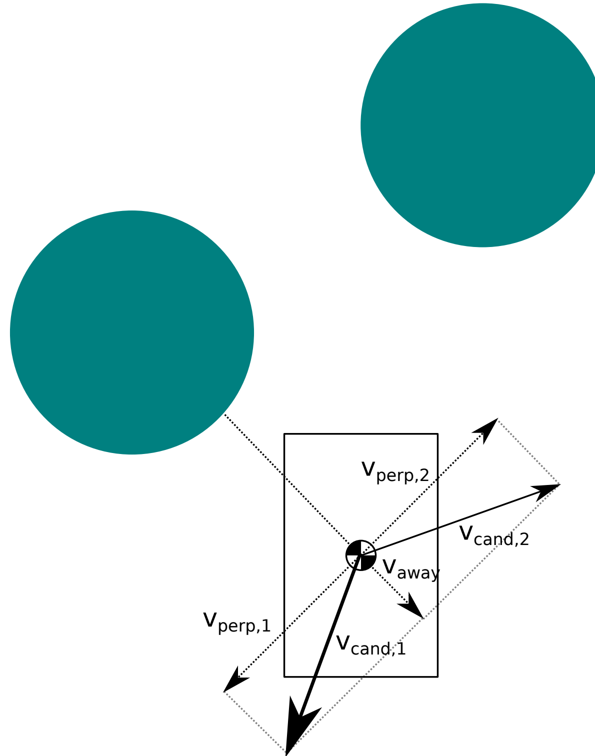


Figure 4.12: Obstacle avoidance calculation

A vehicle under this scenario can dodge some obstacles aimed at any car; however, the performance of the algorithm is not as realistic as it could be. Obstacles do not track over time, and therefore, the algorithm has no information over its motion. As a result, the vehicle would try to move away from an obstacle that is rolling away from the car.

4.4 Notable Details

Some implementation details of the main system involve tradeoffs that put the limit on how closely the system follow the requirements. We discuss such details in this chapter.

4.4.1 Car Consistency

A major issue that occurred during prolonged testing is the car's performance drift over time. The issue was significant in the previous iteration of the car hardware. The previous iteration, as shown in Figure 4.13, is a 1:28 scale car. It had two major issues. First, the build quality was low. For example, its gearbox connecting the motor and the driveshaft were open to the air, making it prone to debris ingestion. Second, the added weight of the custom circuitry caused the motor to overheat. Upgrading the motor or the gear ratio was impossible due to its non-standard components.

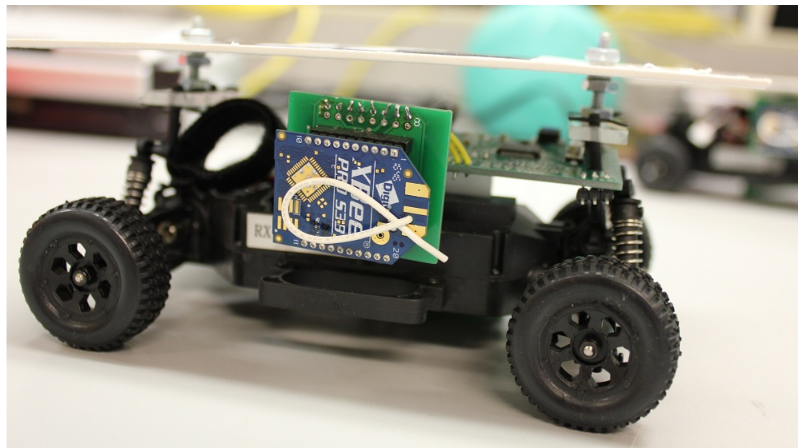


Figure 4.13: Previous iteration car

Combination of the two issues mentioned above resulted in the car drifting in its performance, requiring different tuning parameters for each vehicle. Even then, the performance of each car would drift over time and different runs. The drift makes it impossible to establish what is considered normal in the system.

The solution was to utilize a bigger scale car of higher build quality, as used in the current iteration. The current iteration car is shown in Figure 4.14. Not only is the build quality better, by for example having a closed gearbox, it also uses the motor that is considered standard. With the current iteration, all cars share the same set of tuning parameters to achieve similar tracking performances. However, the current iteration car is still off-the-shelf hardware, and the underlying issue of consistency persists. For example, slack in the steering assembly causes the wheel to change its angle even when the steering servo is locked. The slack causes lateral oscillation at the steady state.

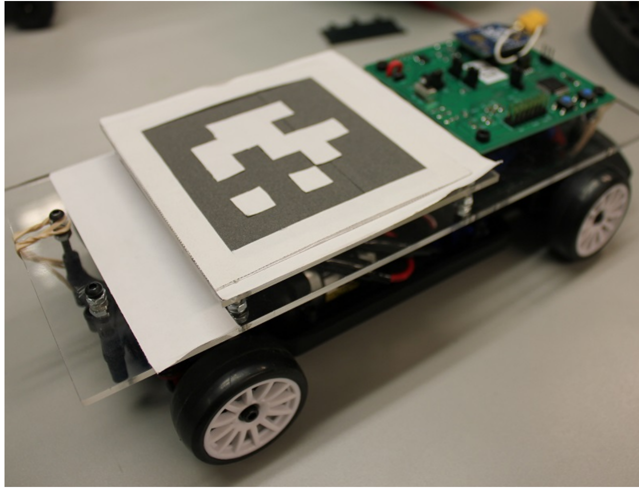


Figure 4.14: Current iteration car

4.4.2 Battery Depletion

Another issue involving the car is how the battery depletes. As the battery is discharged, its voltage follows a non-linear curve [21], as shown in Figure 4.15. V_{nom} is the nominal voltage the battery is advertised for, like 1.5 [V] for an AA battery. However, a fully charged battery has a higher voltage, defined as V_{max}. As the battery discharges, the voltage decreases to around V_{nom}. Towards the end of the discharge, the voltage decreases drastically. In the case of the LiPo battery, discharging the battery below V_{min} permanently damages it.

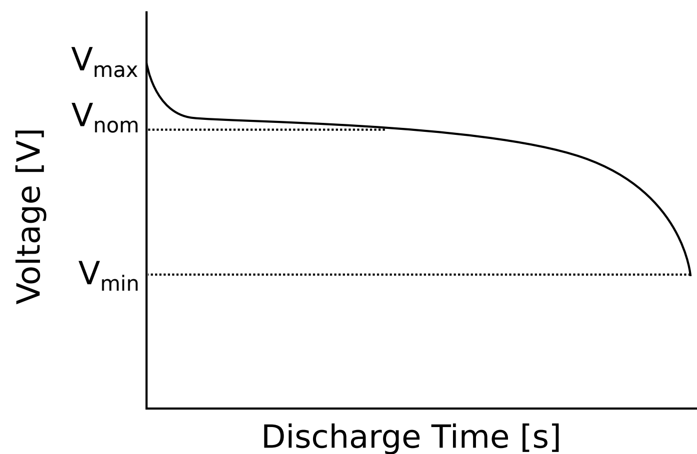


Figure 4.15: Battery discharge curve

As the battery voltage decreases nonlinearly, so does the voltage across the motor. Assuming that the mechanical load on the motor does not change (A reasonable assumption for a car at a steady state), the motor speed is proportional to the applied voltage. The implication is that, as the battery discharges, the car's speed given the same throttle value changes. The closed-loop controller then issues a higher throttle value to keep the car on the goal point. As a result, the nominal value for the vehicle at a steady state drifts.

The current mitigation is to provide a feedback path for the battery voltage, enabling anomaly detection algorithms to take the changing battery voltage into account. Use of larger car also allowed a larger capacity battery to be used, stretching the discharge curve over an extended period.

To mitigate the drift, scaling of the throttle by the voltage is done, by taking a voltage ratio, $V_{curr}/V_{max} = V_{curr}/8.4$, scaling it around 1.0, and multiplying it to throttle values. As seen in Figure 4.16, The normalized throttle value generally stays consistent over time. However, during the first phase of the run, highlighted in blue, the throttle drifts more. Moreover, the residual versus time plot during the phase is not consistent with other phases, as seen in Figure 4.17.

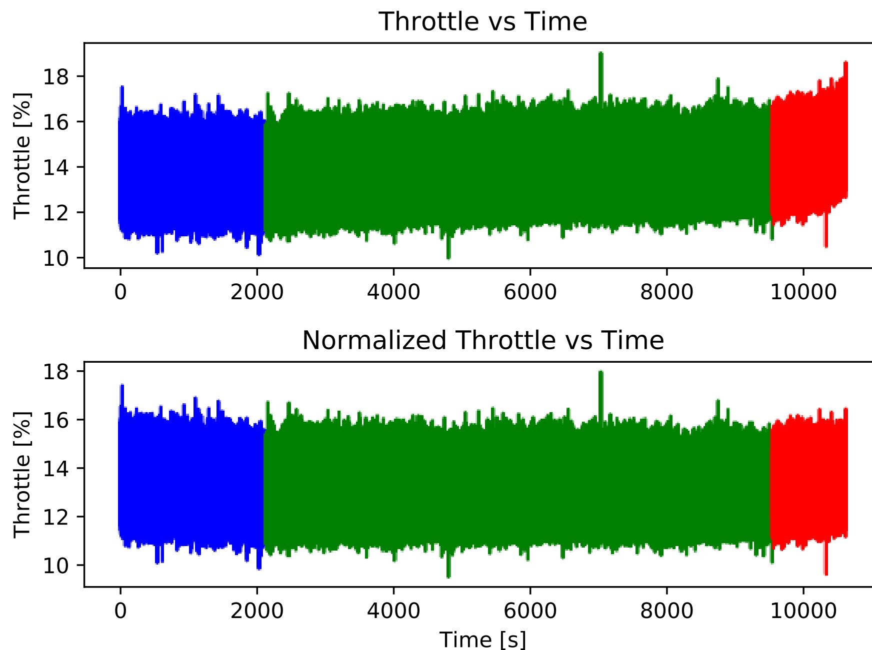


Figure 4.16: Naive and normalized throttle versus time

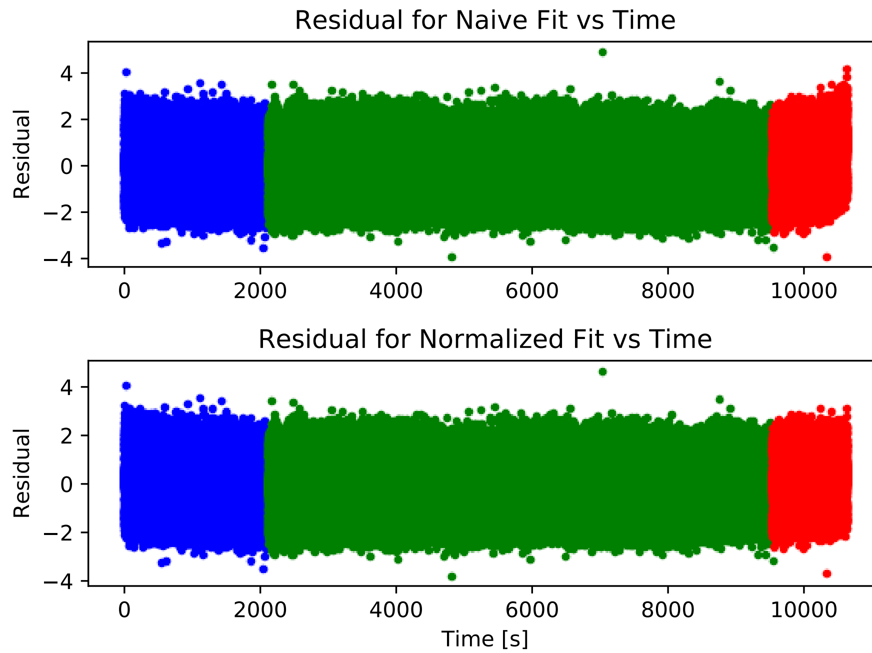


Figure 4.17: Naive and normalized residual plotted versus time

The problem stems from the actual battery discharge curve. As seen in Figure 4.18, the curve's shape differs from the model seen in Figure 4.15. The discharge curve assumes a constant current draw; it seems that this is not the case for the vehicle. The suspicion is that the motor heating up to the equilibrium temperature and changing the current draw is the likely cause.

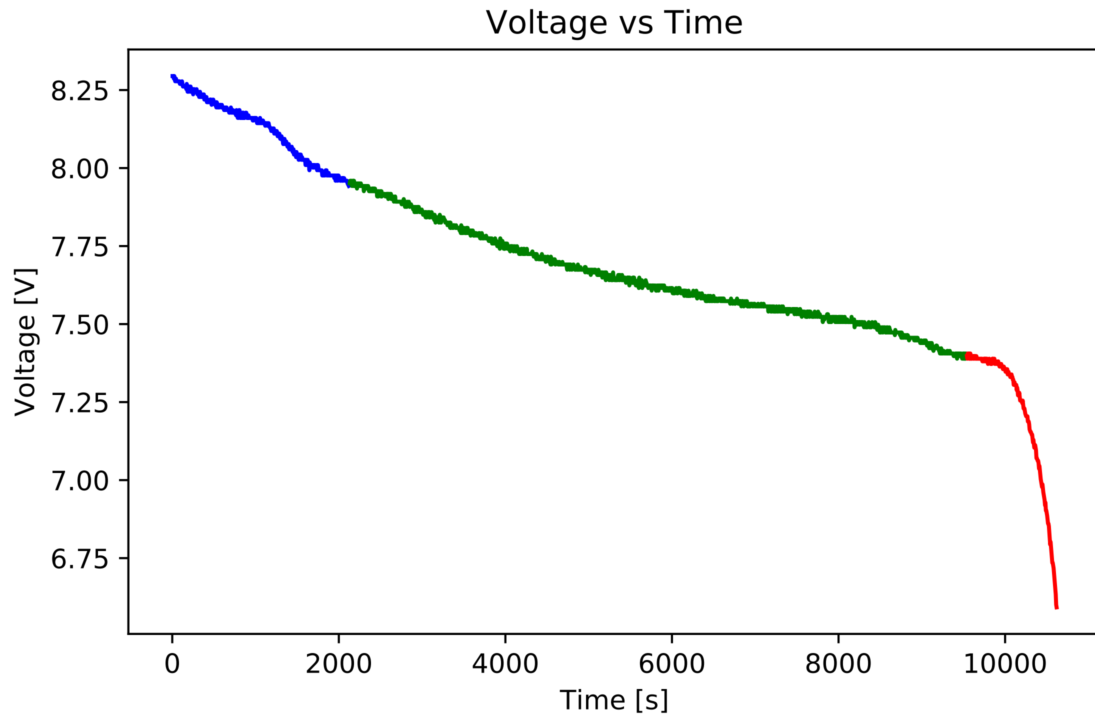


Figure 4.18: Battery voltage versus time

One of the possible mitigations is to let the motor preheat before running an experiment. To verify this, we make the car run for 40 minutes, and then collect data after replacing the battery with a fully charged one. Normalization of throttle values on this data is more consistent, as shown in Figure 4.19. Reducing the heat level, such as installing a motor heatsink, is another possible mitigation.

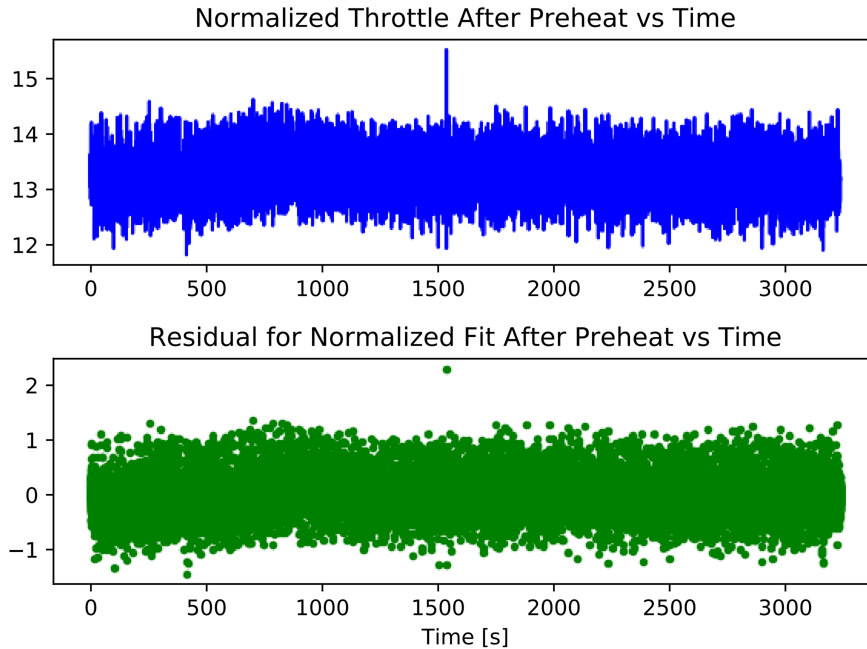


Figure 4.19: Normalized values for a full battery after preheating

4.4.3 Car Dynamics

As mentioned in Section 4.2.2, the lateral controller for a car is a kinematic controller named Stanley controller. Because the kinematic controller ignores dynamics, it makes unrealistic predictions under some circumstances. For example, the kinematic model does not consider a spinout caused by excessive steering for given velocity.

When the implementation from the original implementor of Stanley controller [22] is used as-is, with the cars of previous iterations, cars exhibited severe lateral oscillation at the steady state. Only after updating the controller with Autonomoose's iteration, which adds a damping term [8], the lateral oscillation disappeared.

With the current iteration of cars, removing the damping term does not cause the car to oscillate noticeably at the steady state. Instead, creating both large lateral and forward movements does cause the oscillation to manifest that does not manifest once reaching the steady state. However, the oscillation does not occur with the same command when the conveyor is slowed down.

The oscillation seems to be caused by oversteer. Such inconsistency is from slip, where the velocity vector generated do not match the vector that would be generated by a purely rolling wheel [12]. Slip is demonstrated in Figure 4.20: $v_{roll,f}$ and $v_{roll,r}$ are velocity vectors of wheels if they were purely rolling. These vectors are aligned perpendicularly to each wheel's rolling axis. However, the lateral force exerted on each tire may cause the actual travel velocity of each tire to deviate, like v_f and v_r. The slip angles, α_f and α_r, quantify the deviation between the pure rolling and the actual travel velocity vectors. When $\alpha_f < \alpha_r$, the vehicle exhibits oversteer, where the actual turn radius is smaller than what is suggested by the kinematic model. Oversteer causes a vehicle utilizing Stanley controller to overcompensate and overshoot the desired heading, leading to oscillation observed in the system.

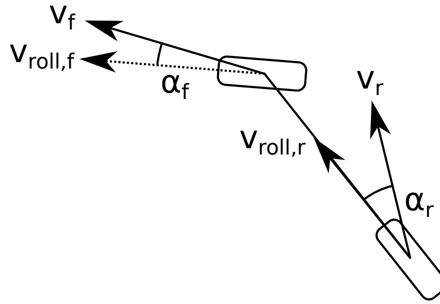


Figure 4.20: An oversteering bicycle model

Multiple factors lead to oversteer. Possible factors include Drivetrain configuration, low vertical loads on tires, low friction between a tire and the road, and large steering, lateral acceleration, or forward velocity.

Considering the tire size and overall weight of the previous iteration cars, it follows that they would be more prone to oversteer. Additionally, they are also modified to be rear-wheel drives to reduce mechanical load leading to overheating. Rear-wheel drive configuration is known to worsen oversteer [14].

There are multiple possible mitigations. First is to prevent an excessive steering value from being generated. Such damping is the solution used by Autonomoose and adapted into the system for its simplicity. However, we note that Autonomoose also implements a separate dynamic controller for aggressive emergency movements. Another approach is to account for the deviation between the yaw angle and the path curvature. This approach is what the original implementors of Stanley controller [24] uses. The mapping required is non-linear and requires manual calibration. Finally, a dynamic controller accounting for forces on the vehicle can be used, allowing extreme maneuvers without uncontrolled oscillation. Liniger et al. employ this approach to control 1:48 scale cars for autonomous

racing [16]. This approach requires hardware modification to measure accelerations on the vehicle.

The current approach has two implications to the system. The primary impact is that the upper layers should not order the car to perform extreme maneuvers, which may cause secondary oscillations. Under collision avoidance scenarios, however, avoiding an imminent collision may require such movements, ideally without leading to secondary crashes from the oscillation. A broader implication is that the implicit assumption of using a kinematic controller of low speed and acceleration may not hold in the system.

4.4.4 State Estimation

As discussed in Section 4.2.2, the moving average filter used for the conveyor velocity filtering is unsuitable for car state estimation, due to its fast changing nature. At first, consider a non-linear Kalman filter, an approach employed by full-sized autonomous cars, such as the Stanley vehicle [24] and Autonomoose [8]. The family of non-linear Kalman filters utilizes a known model to predict the state of the given system and updates the predicted state with the observation from sensor values. Therefore, a Kalman filter's quality of estimation depends on the accuracy of the model. Given the inconsistency discussed in Section 4.4.1, such approach is undesirable.

However, use of a moving average filter, as discussed in Section 4.2.2, is unsuitable due to its inflexibility. Instead, we consider One Euro Filter [5]. It is an adaptive filter utilizing two moving average filters, as shown in Figure 4.21. The rate of change of the signal is filtered through a moving average filter of constant aggressiveness. This value is used to adjust the aggressiveness of the other filter; if the rate of change of the signal is high, the aggressiveness is scaled back to reduce the lag. Otherwise, the signal is smoothed aggressively to achieve low noise.

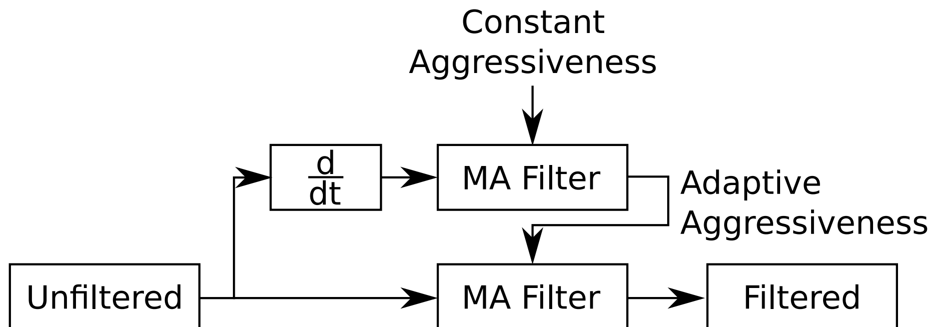


Figure 4.21: One Euro filter

Even if the state estimation does reflect methods used in reality, the current state does provide a reasonable estimate for controlling the cars.

Chapter 5

Trace Tools

As discussed in Chapter 2, the system must provide a facility for generating traces and converting them to formats used by researchers. Since all topics in ROS are public, it is trivial to query them for messages. In fact, ROS provides a tool for recording specified topics into a binary file called a bag file. However, there is one problem using the tools as-is; they are a part of ROS, and can only feed back into the ROS infrastructure. Third-party conversion tools exist, such as CSV converters, but they still depend on ROS. For researchers who merely wish to access the trace, a specific framework like ROS provides little value.

Three use cases exist for accessing trace: Exploratory analysis, offline validation, and online validation. Utilization of the system starts with an exploratory analysis. In this phase, all potentially relevant topics are collected, and then analyzed for their potential application for anomaly detection. Due to the number of topics recorded, it is desired for researchers to extract topics of interest selectively. At the end of the exploratory analysis, researchers select a subset of topics recorded for exploratory analysis. The next step, offline analysis, would involve collecting a trace of the specified topics in a format the anomaly detection algorithm understands. For these two steps, CSV seems to be the desired format, as it is easily readable by humans as well as a wide range of programming languages and frameworks. The final step is online anomaly detection, where data streams to the anomaly detection algorithm in real time.

5.1 Implementation

The trace tools provided in the system provides a ROS-independent workflow for all use cases considered above. The workflow requires the use of generic third-party libraries and tools. One such library is MessagePack. It is a binary serialization specification that allows the encoding of general data structures such as maps [11]. Another tool is Redis, an in-memory database that supports the publish-subscribe infrastructure [1].

The binary generated by MessagePack has a close resemblance to the bag file. Both contain a list of tuples, where each tuple consists of the timestamp received, topic name, and the binary data. The bag file contains serialized data that can only be decoded by importing the message type, making it ROS dependent. By converting each message to a generic map, where a message's member variables map to key-value pairs, the ROS coupling breaks. The bag to MessagePack tool utilizes a custom library that takes advantage of facilities provided by ROS and Python. ROS allows introspection of a message's subfields. The collected subfields are extracted from messages with the use of Python's introspection facilities.

For streaming ROS messages to Redis, a custom node is provided. The node subscribes to topics specified by users. Then, each ROS topic maps to a Redis channel, an equivalent concept. One problem encountered is that ROS messages are, in general, nested maps; however, there is no equivalent concept in Redis. Therefore, it a JavaScript Object Notation (JSON) string representation is instead published to Redis. The representation is suitable because most languages support JSON parsing with reasonable performance. The custom library discussed in the previous paragraph for converting ROS messages into dictionaries is utilized, then Python's JSON library, included with Python by default, is used to convert dictionaries into JSON strings.

5.2 Proposed Workflow

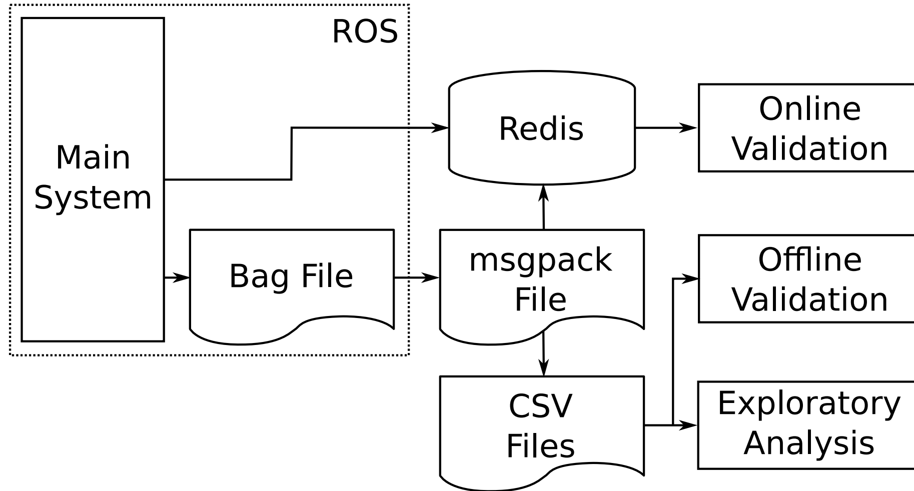


Figure 5.1: Trace tools workflow

Using the tools discussed above, a ROS-independent workflow, as specified in Figure 5.1, is proposed. During the exploratory analysis, all topics of interest are collected into a bag, then converted to a msgpack file. Researchers would use the msgpack to CSV conversion tool to selectively analyze subsets of recorded messages. Once researchers select the final set of topics, additional bag files can be collected, and the workflow used for the exploratory analysis can be applied. If the researchers desire online verification, they will interface with Redis, subscribing to appropriate channels and decoding the JSON strings. The infrastructure can be tested by streaming the msgpack file. Once the researchers verify the infrastructure, the system is launched with the Redis streaming node for live verification.

Chapter 6

Anomaly Injection Framework

The system described so far enables users to execute non-trivial cooperative driving scenarios, and collect and stream data from it. However, the system also needs to provide a way for researchers to inject anomalies into the system. To this end, anomaly injection framework is introduced, consisting of two components: A characterization document to guide the creation of anomalies and a suite for implementing them.

6.1 Anomaly Characterization

The creation of characterization document follows an industry standard process, Failure Mode Effects Analysis (FMEA). As the name implies, the document allows users to infer effects of different failures in various parts of the system. It is a tabular document, where each row is a combination of a system component and a failure mode. Relevant columns not only give researchers an insight into the system without in-depth analysis but also defines the thought process for any additional component and failure mode combination. The first column indicates the function of the system under analysis. Since anomalies are injected, as discussed in Section 3.1, into ROS topics, this column consists of ROS topics. The failure mode column describes types of anomalies to introduced. Details of the anomaly can be flexible, but in general, it falls under one of the following categories: Intermittent loss, injection of nonsensical messages, a transformation of incoming messages, and hardware failure. Effect of the failure is assessed in the next column. For the proposed system, it divides into three parts: Effect to nodes that directly subscribes to the failing topic, effects to nodes that subscribe to that node, and effects to the system as a whole. Such breakdown is allowed as there are clear dependencies between nodes. Then, a score

between 1 to 10 is assigned for severity. A score of 10 is the most severe, assumed to cause behavior that may damage the hardware. A severity of 1 indicates an inconsequential failure mode. The next column outlines potential causes for the given row. Analysis of causes allows researchers to consider how realistic a given failure mode may be. A potential cause can be in the context of the system as-is, or the simulated driving scenario. Overall, the FMEA document provides not only a systematic breakdown of a failure mode but also a template for performing it for new topics. An example of a row is shown in Table 6.1. The full document is shown in Appendix B.

Topic	Car Pose
Failure Mode	Transformation - Constant offset of position
Effects	1. Controller believes the offset pose is true pose 2. Vehicle commands issued to compensate for perceived perturbation 3. Vehicle position offset, opposite to the offset
Severity	3-8, Depends on amount of offset
Causes	• Driving scenario: Spoofing attack • System as-is: Mount position offset from config

Table 6.1: An example FMEA row, transposed to fit the page

Of note is that the monitors, discussed in Section 4.3, are implemented as a result of this document, to handle severe failure modes. Such failures, mostly hardware failures, are mitigated by the use of critical monitors.

6.2 Anomaly Injection Suite

The anomaly injection suite, as mentioned, provides a set of tools and libraries to implement a new anomaly. The tools consider three anomaly scenarios: Message injection, message delay and Man-in-the-Middle (MitM). Message injection is a trivial case, since, as mentioned, ROS makes all topics available publically. The latter cases are not, so tools are provided to ease the implementation.

The tool for inducing a delay into a specified topic achieves functionality by taking advantage of the fact that ROS utilizes standard TCP sockets. As mentioned in Section 3.1, one pair of publisher and subscriber on a topic has one corresponding TCP connection. Therefore, an established Linux traffic control tool, tc, and its subset utility, NetEm, are utilized to inject the delay. They redirect the traffic between specified source and

destination ports into a particular channel with an induced delay. The delay injection tool ensures intuitiveness by using the ROS infrastructure to infer the ports used based on a specified topic and subscriber name, instead of requiring the users to figure them out.

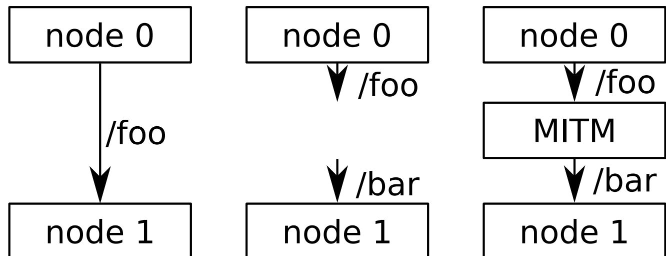


Figure 6.1: Injection of a Man-in-the-Middle node

As mentioned in Section 3.1, one can remap the names of topics a node looks for, at the time the node is being launched. Using the remapping functionality, one should be able to disrupt a publisher-subscriber relationship, and inject a middle node, as shown in Figure 6.1. In this example, $node0$ publishes to foo and $node1$ subscribes to it. When launching $node1$, researchers can specify any occurrence of foo to be replaced with bar, causing $node1$ to subscribe to bar. At this point, a new node that subscribes to foo and publishes to bar is injected, becoming a man in the middle. Such node can achieve many anomaly scenarios involving drop or transformation of messages.

The above example, however, is a trivial case; nodes in the system are numerous and are launched by a tree of launch files specifying information about nodes to launch. The anomaly injection suite alleviates having to navigate through all launch files by providing a custom launcher script. The script takes the root launch file and a remapping configuration file. The remapping configuration file contains the names of nodes to perform remappings to, and those remappings are. The launcher script then navigates the launch file tree, automatically applying the specified remapping to each node.

Finally, the suite provides a library for annotating the anomaly. The annotation is published as a topic, which then is recorded for exploratory analysis. A message from the topic contains information such as the name, enable status, and configuration of the anomaly. The configuration field is flexible and implementation-defined. For example, an anomaly injector node that drops messages probabilistically may publish the annotation containing information on how many messages are dropped so far.

6.3 Example Implementations

Some example anomalies are provided in the system as a reference for using the anomaly injection suite. Implemented anomalies are camera delay, unintended acceleration or deceleration, camera dead spot, and GPS spoofing.

Camera delay simulates an anomalous scenario where the data channel used by the tracker camera is saturated, delaying frame data from being transmitted. Ultimately, the system would be working with delayed information on vehicles' poses. As a result, the control loops for cars become less stable.

Unintended acceleration or deceleration is inspired by a real-life case where hackers were able to wirelessly cut the transmission of a car driving on a highway [17]. An injection node periodically publishing the car command data with a scaled throttle value simulates such behavior. Depending on the scaling, the car periodically jolts forward or backward, with the amount depending on how much the injected messages flood and overwhelm the legitimate messages.

Figure 6.2: Dead spot

Camera dead spot stems from a real scenario experienced in the earlier iteration of the system. At that point, the system used the visible, instead of Near IR, light for tracking cars. This approach was sensitive to changes in ambient light conditions. A spotlight that shines on the conveyor surface, for example, would degrade the tracking reliability over the highlighted region. The phenomenon is shown in Figure 6.2. The vehicle on the left side of the image is outside the spotlight and is detected, as indicated by the

superimposed tracking information. On the other hand, the car on the right, which rests over the spotlight, is not detected. Any car within such dead spot is uncontrollable until the car exits the dead spot. The scenario is now simulated by probabilistically dropping pose data of a car within the specified region.

GPS spoofing is a more involved application of the MitM attack, where an offset into the car's pose data is introduced. The implication is already discussed in Table 6.1. The injector extends the injection functionality by enabling users to specify the desired position to send the car to. Additionally, the offset values are introduced gradually, to avoid trivial detection caused by a sudden jump in the pose.

Chapter 7

Case Study

In this section, we discuss the validation of the system by running an existing anomaly detection algorithm on a trace generated by the system and ensuring that any instance of injected anomaly is detected. The case study uses inter-arrival curves based anomaly detection [20] to detect unintended acceleration of a car. This section discusses the background of the anomaly detection algorithm, scenario setup, and the results.

7.1 Inter-Arrival Curves Based Anomaly Detection

An inter-arrival curve builds from a trace of discrete events. Two parameters are required: The event of interest and the maximum sliding window to apply. The curve consists of discrete points, with each point showing the number of the specified event's inter-arrival count for given window size. The inter-arrival count for a given window size is counted by finding the specified event and counting the occurrence of that event within the sliding window starting from it. An example arrival count for event c, with selected window sizes are shown in Table 7.1.

Event	a	a	b	c	a	a	b	b	c	a	a	a	b	b	a	c	c	c	a	a	a...
$IA(c, 4)$	-	-	-	1	-	-	-	-	1	-	-	-	-	-	-	3	2	1	-	-	-...
$IA(c, 6)$	-	-	-	2	-	-	-	-	1	-	-	-	-	-	-	3	2	1	-	-	-...
$IA(c, 10)$	-	-	-	2	-	-	-	-	4	-	-	-	-	-	-	3	2	1	-	-	-...

Table 7.1: An example of counting inter-arrival for event c

A pair of inter-arrival curves is plotted to characterize a program trace. One curve counts the maximum number of inter-arrivals for given sliding window sizes. The other curve tracks the minimum. For the three inter-arrival counts in Table 7.1, $IA_{min}(c, N) = 1, N \in \{4, 6, 10\}$, where $IA_{max}(c, 4) = IA_{max}(c, 6) = 3$, and $IA_{max}(c, 10) = 4$. For a given program, at a steady state, an event trace generated would follow a pattern, which is encoded into the min-max inter-arrival curves. An anomalous behavior would break the pattern, and change the shape of the curves. As such, the anomaly detector based on it works as follows. First, a model pair of inter-arrival curves is built from nominal traces generated by the specified program. Then, a runtime pair of inter-arrival curves is calculated during the program runtime. If shapes of the two curves classify as a close match, the program is running nominally.

7.2 Setup

The scenario chosen for inter-arrival curves anomaly detection is an unintended acceleration scenario under a cruise condition. Nominally, two cars are running on the conveyor, both tracking to static goal points. The specified goal points are front to back, with some space between them. An anomaly is introduced into the car at the back, by inducing unintended accelerations. The acceleration causes the rear vehicle to jolt forward, possibly colliding into the front car.

The topic monitored is the command value for the rear car. Nominally, the throttle should be relatively stable, with small amounts of drift caused by factors such as uneven conveyor surface and tracking noise. The values are binned into a histogram. Under nominal scenarios, one bin would have a dominating count. Bins adjacent to it would have lower counts. For the anomaly detector used, each bin is one discrete event, with the main bin being the event of interest. Unintended acceleration would be classified into a completely different bin, therefore changing the pattern of event traces.

7.3 Result

Given the adequate amount of nominal traces, we were able to build a suitable inter-arrival curves model that classified unintended accelerations correctly. The first experiment was carried out with the earlier iteration cars, where battery feedback was unavailable. As discussed in Section 4.4.2, this caused the throttle value to drift over time. The drift

would result in the throttle histogram shifting, leading to false positives. The scaling carried out in Section 4.4.2 should mitigate the issue.

Chapter 8

Evaluation

In this section, we evaluate the trace tools and the anomaly injection suite by assessing their latencies and resource usages. Because calculation of latency would require timestamping the messages, purpose-built ROS nodes are used instead of the main system.

8.1 Objectives

All components discussed below run alongside the main system; therefore, their CPU and memory usage should not be excessive as not to interfere.

8.1.1 Trace Tools

The conversion segment of the trace tools already has automated tests to verify correctness. The experiment focuses on the ROS to Redis streaming capability. The primary concern is latency, which breaks down into two components. Since the Redis streaming functionality enables online verification, it should not incur excessive latency. Moreover, it should not slow down other subscribers depending on the topics streamed.

8.1.2 Anomaly Injection Suite

As discussed in Section 6.2, three approaches exist for injecting anomalies: Injection, delay, and Man-in-the-Middle. The injection suite provides specific procedures for the latter two to be verified.

Like the Redis streaming tool, both injection approaches run alongside the system, so it should not consume excessive computational resources.

Specifically for delay injection, the correctness should be verified by ensuring that the latency increases as specified. For the Man-in-the-Middle remapping approach, we must ensure that remapping does not induce excessive latency that would render the system unstable.

8.2 Experimental Setup

8.2.1 Baseline

The experiment features publisher-subscriber pairs. Publishers generate data that the timestamp at that time. When subscribers receive a message, it calculates and records the difference between the timestamp in the message and the current time. This value is latency.

The publishers subscribe to a node akin to a clock generating a set number of ticks. The publishers will only publish one message per one tick message. Ticks simulate the actual system, where most messages depend on data related to cars, which rely on the car tracking camera. The camera runs at 30 frames per second, so the clock node generates 30 ticks per second.

There are two types of messages exchanged in the experiment. Both are one of ROS' default message types: Time and PoseStamped. Time represents the smallest size message while being able to calculate latency. PoseStamped, on the other hand, contains many 64-bit floating point values and is considered a large message in the main system. There are larger data published in the system; however, such data is large because it is unprocessed, like the camera frame data. We assume that users will opt to work with smaller messages produced by processing such data.

The overall memory and CPU usage of the system is measured while any experiment is running, using Linux commands free and mpstat. All scenarios discussed below will also record the resource usage to compare with the baseline.

To establish a baseline, we run the setup above with a different number of publisher-subscriber pairs $N \in 1, 5, 10, 20, 30$. For all experiments, 900 pulses are generated, which would cause each publisher-subscriber pair to create 900 messages. The experiment runs on the PC used for the main system, with Intel Xeon 6-core 2.40 GHz CPU and 256 GiB RAM.

8.2.2 Trace Tools: Redis Streaming

In addition to the setup discussed in the previous section, Redis streaming is enabled, for all messages between each publisher-subscriber pair. Latency values calculated by subscribers are recorded as usual. These values are compared against the baseline to measure the impact of the streaming on the system.

Additionally, a single Python script queries Redis for streamed data and calculates latency by extracting the timestamp from each message. Because many messages generated from multiple processes funnels into one script, latency values would be more significant. Excessive latency here would cause a delay in anomaly detection.

8.2.3 Trace Tools: Delay

This scenario involves two publisher-subscriber pairs. One pair has delays induced of different values. Latency values in the affected pair should follow the delay specification, while ones in the unaffected pair should match the latency values in the baseline scenario.

8.2.4 Trace Tools: Man-in-the-Middle

The Man-in-the-Middle scenario uses one publisher-subscriber pair. The pair uses the remapping to allow a man-in-the-middle node. This node directly passes the message to the subscriber. The subscriber's latency values are recorded. Because the man-in-the-middle node is a pass-through, the latency value, when compared against the baseline values, represent the minimum latency incurred by the remapping strategy.

8.3 Results

8.3.1 Baseline

The baseline latency result is shown in Table 8.1, as 3σ confidence intervals. The maximum latency at $N = 20$ with large messages is under 1 [ms], less than 3 [%] of the 30 [Hz] interval. No conclusion can be made if there are significant differences between latency values, as their confidence intervals all overlap.

N	Latency ($\mu \pm 3\sigma$) [μs]	
	Small Message	Large Message
1	332 ± 46.5	385 ± 77.2
2	309 ± 82.0	375 ± 92.8
5	310 ± 120	373 ± 127
10	316 ± 138	391 ± 161
20	318 ± 159	396 ± 198
30	324 ± 197	407 ± 245

Table 8.1: Baseline latency values

CPU and memory usages, as shown in Table 8.2, do increase as N increases.

N	CPU Usage ($\mu \pm 3\sigma$) [%]		Memory Usage ($\mu \pm 3\sigma$) [KiB]	
	Small Message	Large Message	Small Message	Large Message
1	0.387 ± 0.102	0.389 ± 0.133	1300 ± 3.75	1300 ± 3
2	0.376 ± 0.086	0.512 ± 0.229	1460 ± 6.09	1470 ± 7.41
5	0.345 ± 0.148	1.21 ± 0.216	1770 ± 5.55	1770 ± 5.22
10	1.31 ± 0.157	1.24 ± 0.18	2350 ± 6.45	2350 ± 8.22
20	1.97 ± 1.488	2.35 ± 0.96	3520 ± 5.79	3530 ± 10.6
30	3.17 ± 3.21	3.67 ± 2.409	4690 ± 6.15	4700 ± 11.2

Table 8.2: Baseline resource usage

8.3.2 Trace Tools: Redis Streaming

Latency values for the latency on the Redis side is shown in Table 8.3. At $N = 1$, latency confidence intervals for both small and large messages do not overlap with the baseline confidence intervals. Therefore, we conclude that messages streamed through Redis does incur a latency penalty; however, it is still around 1 [ms]. As N increases, latency means and standard deviations rapidly increase. The increase in standard deviation values makes sense, as messages from multiple processes direct into one Python script. The script becomes a bottleneck, where some messages may get processed promptly, but not others.

N	Latency ($\mu \pm 3\sigma$) [μs]	
	Small Message	Large Message
1	776 ± 244	999 ± 182
5	1290 ± 1449	1910 ± 3270
10	1980 ± 2949	3070 ± 4800
20	3190 ± 6240	5220 ± 9450
30	4430 ± 8730	7110 ± 13470

Table 8.3: Redis latency values

Table 8.4 shows the overhead induced by running the Redis streaming script to the subscribers. The confidence intervals overlap with baseline, so no significant conclusion about the latency increase can be made.

N	Latency ($\mu \pm 3\sigma$) [μs]	
	Small Message	Large Message
1	360 ± 108	422 ± 122
5	391 ± 230	462 ± 252
10	407 ± 280	469 ± 318
20	392 ± 257	455 ± 288
30	390 ± 279	467 ± 324

Table 8.4: Redis overhead values

The resource usage is shown in Table 8.5. As N increases, there are significant differences CPU and memory usages when Redis streaming is enabled; however, the maximum CPU and memory utilization are less than 15 [%] and 5 [MiB].

N	CPU Usage ($\mu \pm 3\sigma$) [%]		Memory Usage ($\mu \pm 3\sigma$) [KiB]	
	Small Message	Large Message	Small Message	Large Message
1	0.398 ± 0.157	0.421 ± 0.189	1360 ± 7.89	1360 ± 11.1
5	0.805 ± 0.131	1.85 ± 0.18	1820 ± 5.31	1830 ± 5.43
10	2.29 ± 0.201	2.6 ± 0.208	2410 ± 7.89	2420 ± 6.87
20	4.58 ± 1.2	5.32 ± 1.11	3590 ± 7.77	3590 ± 7.59
30	6.54 ± 2.95	8.34 ± 3.21	4770 ± 17.6	4770 ± 12

Table 8.5: Redis resource usage

8.3.3 Trace Tools: Delay

Table 8.6 shows the latency values of messages both affected and unaffected by the induced delay. As seen, the latency values of unaffected messages do not seem to change between different delays. On the other hand, latency values of the affected messages do seem to follow specified delay values.

Delay ($\mu \pm 3\sigma$) [ms]	Unaffected Latency ($\mu \pm 3\sigma$) [μs]		Affected Latency ($\mu \pm 3\sigma$) [μs]	
	Small Message	Large Message	Small Message	Large Message
5 ± 0	319 ± 130	383 ± 183	5330 ± 139	5400 ± 162
5 ± 6	314 ± 88.5	372 ± 74.4	5360 ± 6000	5390 ± 5910
15 ± 0	311 ± 83.1	389 ± 154	15300 ± 113	15400 ± 101
15 ± 15	316 ± 142	374 ± 133	15400 ± 15060	15400 ± 15090

Table 8.6: Delay latency values (N = 2)

As shown in Table 8.7, no conclusion can is made about the resource usage compared to the baseline.

Delay ($\mu \pm 3\sigma$) $[ms]$	CPU Usage ($\mu \pm 3\sigma$) [%]		Memory Usage ($\mu \pm 3\sigma$) $[KiB]$	
	Small Message	Large Message	Small Message	Large Message
5 ± 0	0.221 ± 0.176	0.663 ± 0.363	1430 ± 29.6	1430 ± 28.9
5 ± 2	0.715 ± 1.94	0.721 ± 0.318	1430 ± 47.4	1430 ± 31.5
15 ± 0	0.4 ± 0.235	0.599 ± 0.262	1430 ± 30	1430 ± 29.7
15 ± 5	0.41 ± 0.223	0.576 ± 0.315	1430 ± 30.3	1430 ± 30

Table 8.7: Delay resource usage (N = 2)

8.3.4 Trace Tools: Man-in-the-Middle

No conclusion can be made about the latency introduced by using the remapping strategy for man-in-the-middle anomalies, compared to the baseline. The result is shown in Table 8.8.

	Small Message	Large Message
Latency ($\mu \pm 3\sigma$) $[\mu s]$	346 ± 113	385 ± 137

Table 8.8: MitM latency values (N = 1)

Table 8.9 shows that while no conclusion about the CPU usage can be made, there is a difference in the memory usage between the baseline scenario and the man-in-the-middle scenario.

	Small Message	Large Message
CPU Usage ($\mu \pm 3\sigma$) [%]	0.373 ± 0.3	0.407 ± 0.23
Memory Usage ($\mu \pm 3\sigma$) $[KiB]$	1360 ± 7.23	1360 ± 9

Table 8.9: MitM resource usage (N = 1)

Chapter 9

Future Work and Conclusion

This chapter outlines future work to be done to the system, followed by remarks on the current state of the work.

9.1 Future Work

Several factors limit the consistency and flexibility of the system. In this section, we discuss each issue and propose possible improvements.

9.1.1 Dedicated Car Hardware

As mentioned in Section 4.4.1, even the latest iteration of cars utilize a hobbyist-grade commodity product. Tolerances involved in components of the vehicle seem high, inducing play. The issue has a significant impact on the steering assembly, with the front wheel having enough mechanical slack to cause wobbling. The wobbling results in a slow, but noticeable oscillation.

Some design choices made for the product also do not make sense for the system. The products are designed to maximize excitement, with the powertrain geared for a higher speed. Such configuration causes more motor heat and shorter runtime.

Building dedicated car hardware would allow for a tighter tolerance, and complete control over design choices, resulting in improved consistency.

9.1.2 Improved Car Control

As discussed in Section 4.4.3, a purely kinematic controller used by the system limits the stability of large maneuvers. The limitation is especially disadvantageous for emergency maneuvers. Mitigating the issue would improve the flexibility of the system. There are two approaches: Augmenting the existing kinematic controller or adapting a dynamic controller.

The augmentation follows what the original implementors of Stanley controller did, providing a mapping to account for the dynamic behavior of cars [24]. However, this requires the cars' consistency to be improved to be effective.

Adapting a dynamic controller can be done in two ways. The existing kinematic controller can be entirely replaced, as with the system built by Liniger et al. [16]. On the other hand, the kinematic controller can is still suitable for normal driving scenarios, the dynamic controller for emergency maneuvers. Autonomoose [8] implements this strategy. The prerequisite to either approach is that the dynamic states of the car must be collected, by for example adding an Inertial Measurement Unit (IMU) to each car to measure the acceleration values.

9.1.3 Obstacle State Estimation

The currently implemented obstacle avoidance scenario, unfortunately, is not useful, as discussed in Section 4.3. The behavior of cars running the scenario is unrealistic, as vehicles would try to move away from obstacles not heading towards them. Such behavior is because the threat model discussed in Section 4.2.3 does not uniquely identify each obstacle. Lack of unique identification prevents the estimation of obstacles' trajectories. Unfortunately, uniquely identifying obstacles is a non-trivial problem, especially when obstacles can collide with each other and become a single blob from the perspective of the threat model. It is possible that the definition of obstacles in the system may need to change.

Enabling obstacle state estimation would allow for a more realistic behavior from cars. Also, it introduces a new possible venue for anomaly injection: Incorrect estimation of obstacle trajectories.

9.2 Conclusion

In this thesis, we discussed the implementation of the system dedicated to generating traces that can be used to verify anomaly detection algorithms. The implemented system not only provides a non-trivial simulation for cooperative driving scenarios, but also facilities for collecting traces, and creating and injecting anomalies. The system's purpose is also verified through a case study with a known anomaly detection algorithm, and a collection of latency and resource usage of trace tools and anomaly injection tools.

The system is still at its early stages of implementation, with many potential improvements. However, we hope that, over time, the system proves useful to researchers working on anomaly detection algorithms.

References

[1] Introduction to redis.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[3] 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*, pages 356–363. ACM, 2006.

[4] Bloomberg. Bloomberg aspen initiative on cities and autonomous vehicles, Oct 2017.

[5] Géry Casiez, Nicolas Roussel, and Daniel Vogel. 1 filter: a simple speed-based low-pass filter for noisy input in interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2012.

[6] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, 2009.

[7] E.M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model checking and the state explosion problem. In *Tools for Practical Software Verification*. Springer, 2012.

[8] A Dakibay. Autonomous driving: Baseline autonomy. Master's thesis, 2017.

[9] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo. A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. In *Applications of Data Mining in Computer Security*. Springer, 2002.

[10] FDA. Fda permits marketing of artificial intelligence-based device to detect certain diabetes-related eye problems. *FDA Press Release*, 2018.

[11] S. Furuhashi and S. Tagomori. msgpack/spec.md, Aug 2013.

[12] M Guiggiani. *Handling of Road Cars*. 2014.

[13] S Hangal and M Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 2002.

[14] M Hoeck and C Gasch. The influence of various 4wd driveline configurations on handling and traction on low friction surfaces. Technical report, SAE Technical Paper, 1999.

[15] E Keogh, J Lin, and A Fu. Hot sax: efficiently finding the most unusual time series subsequence. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, 2005.

[16] A Liniger, A Domahidi, and M Morari. Optimizationbased autonomous racing of 1:43 scale rc cars. *Optimal Control Applications and Methods*, 36(5):628–647, 2014.

[17] C Miller and C Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[18] E. Olson. Apriltag: A robust and flexible visual fiducial system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2011.

[19] M. Quigley, J. Faust, T. Foote, and J. Leibs. Ros: an open-source robot operating system. 2009.

[20] M. Salem, M. Crowley, and S. Fischmeister. Anomaly detection using inter-arrival curves for real-time systems. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 97–106, 2016.

[21] C. Simpson. Characteristics of rechargeable batteries, 2007.

[22] J Snider. Automatic steering methods for autonomous automobile path tracking. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08*, 2009.

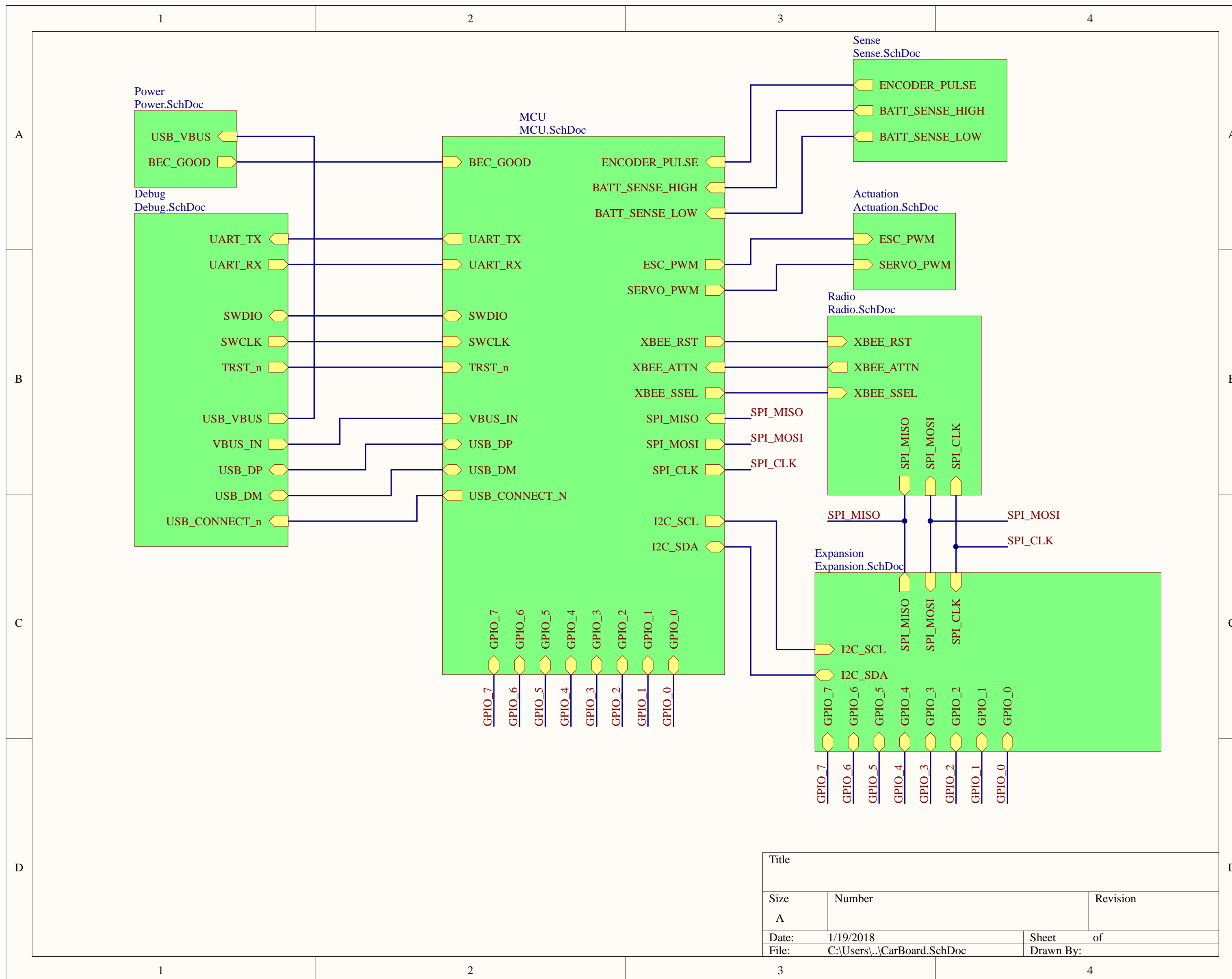
[23] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, 2009.

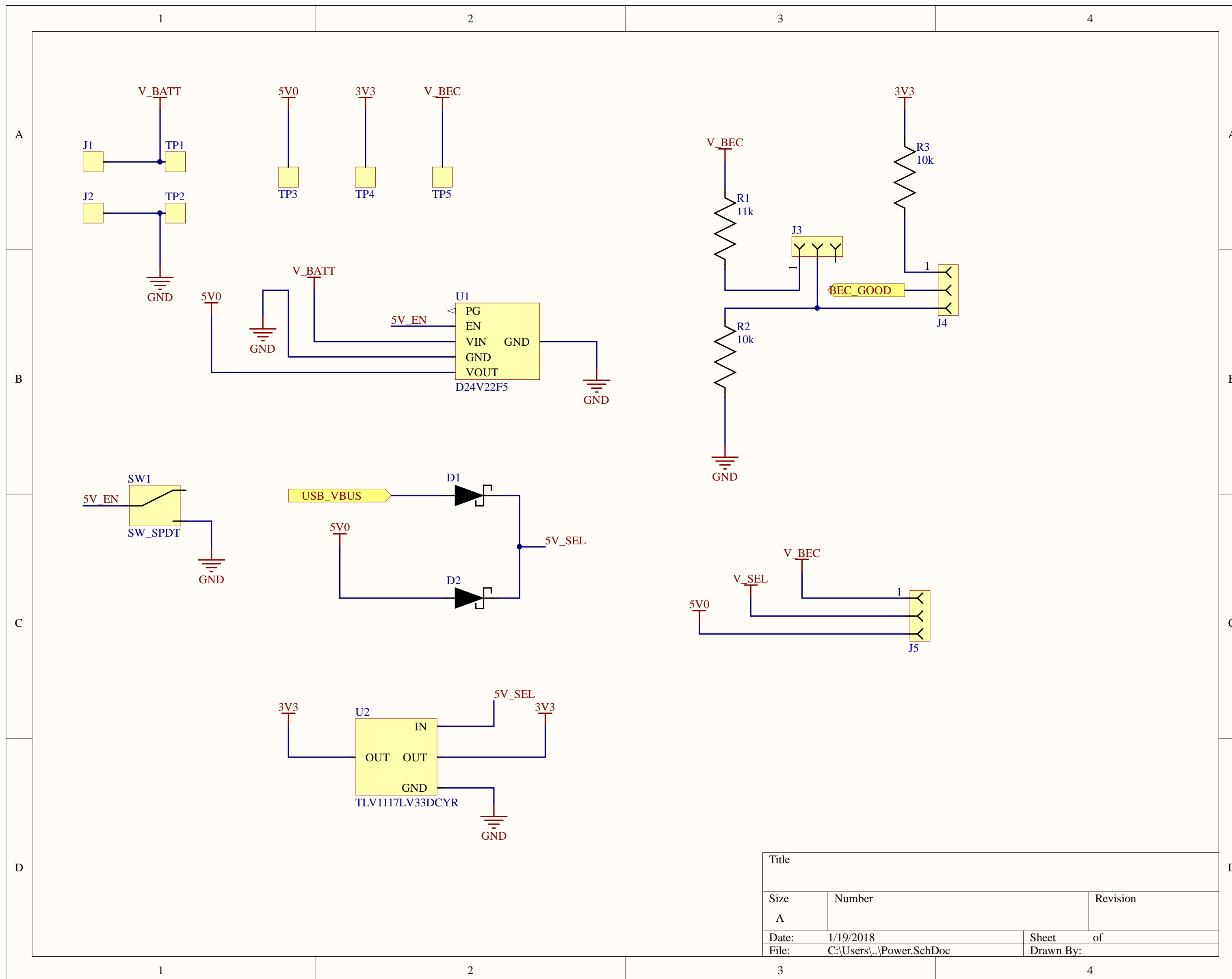
[24] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, pages 661–692, 2006.

APPENDICES

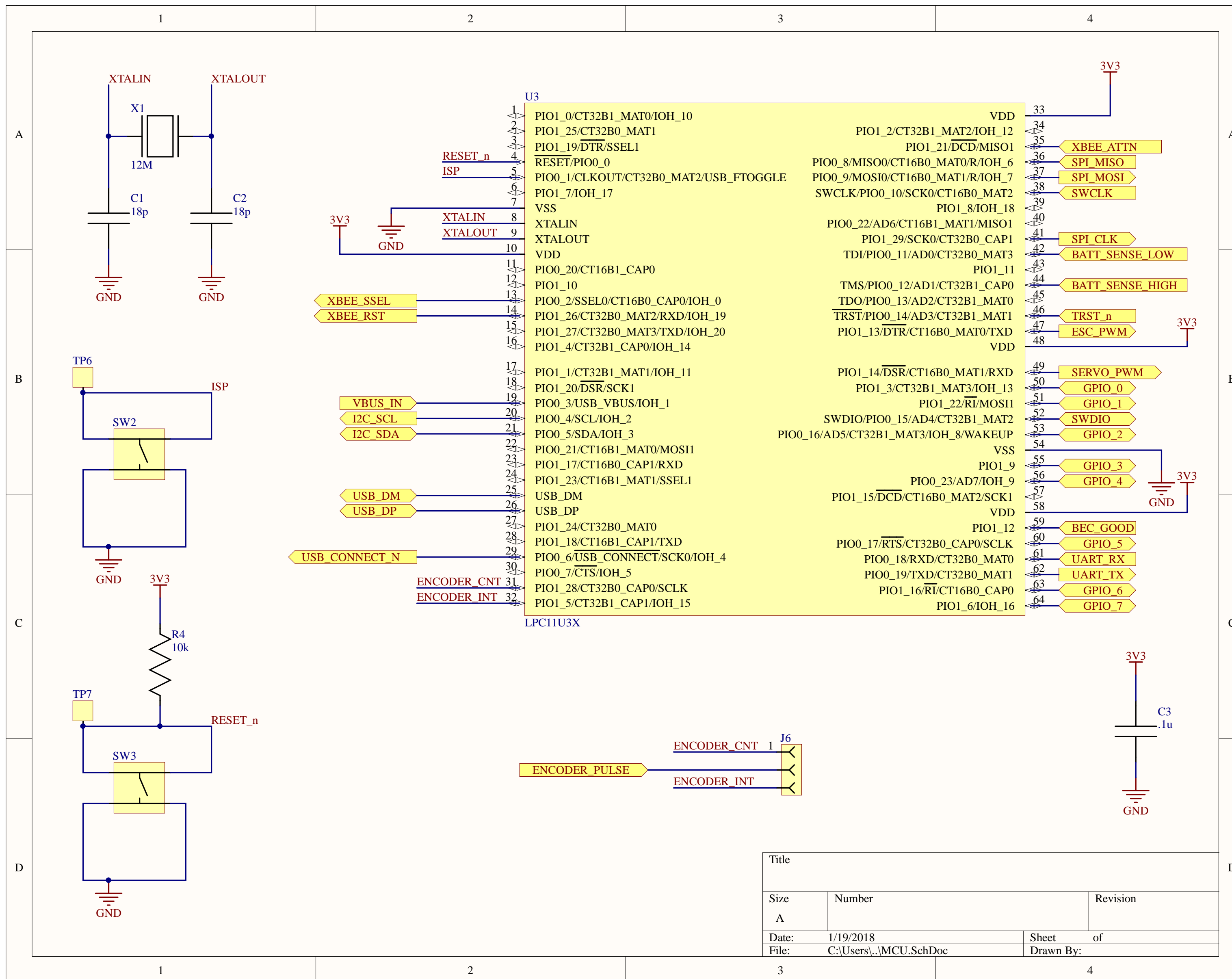
Appendix A

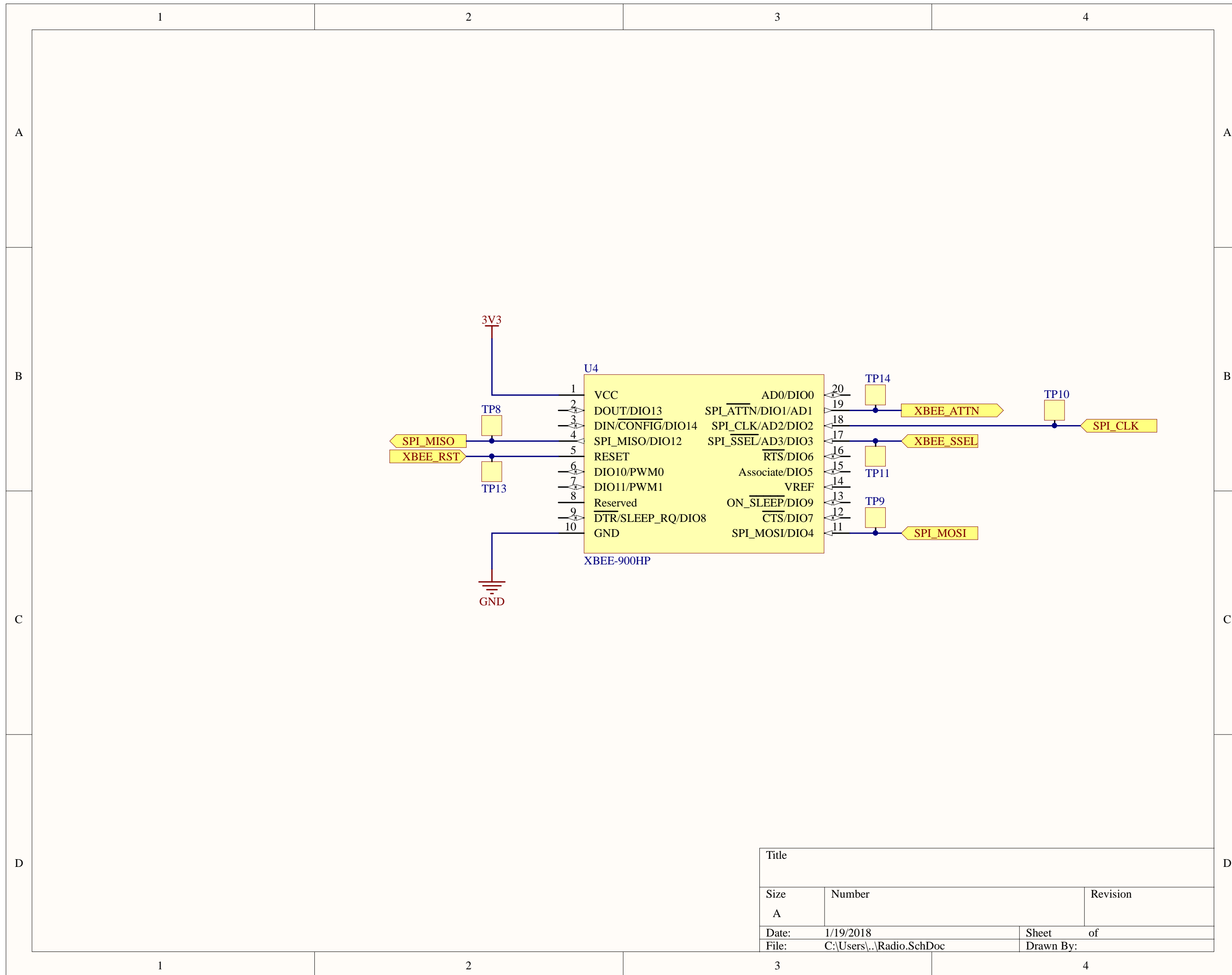
Car Circuit Board Schematic

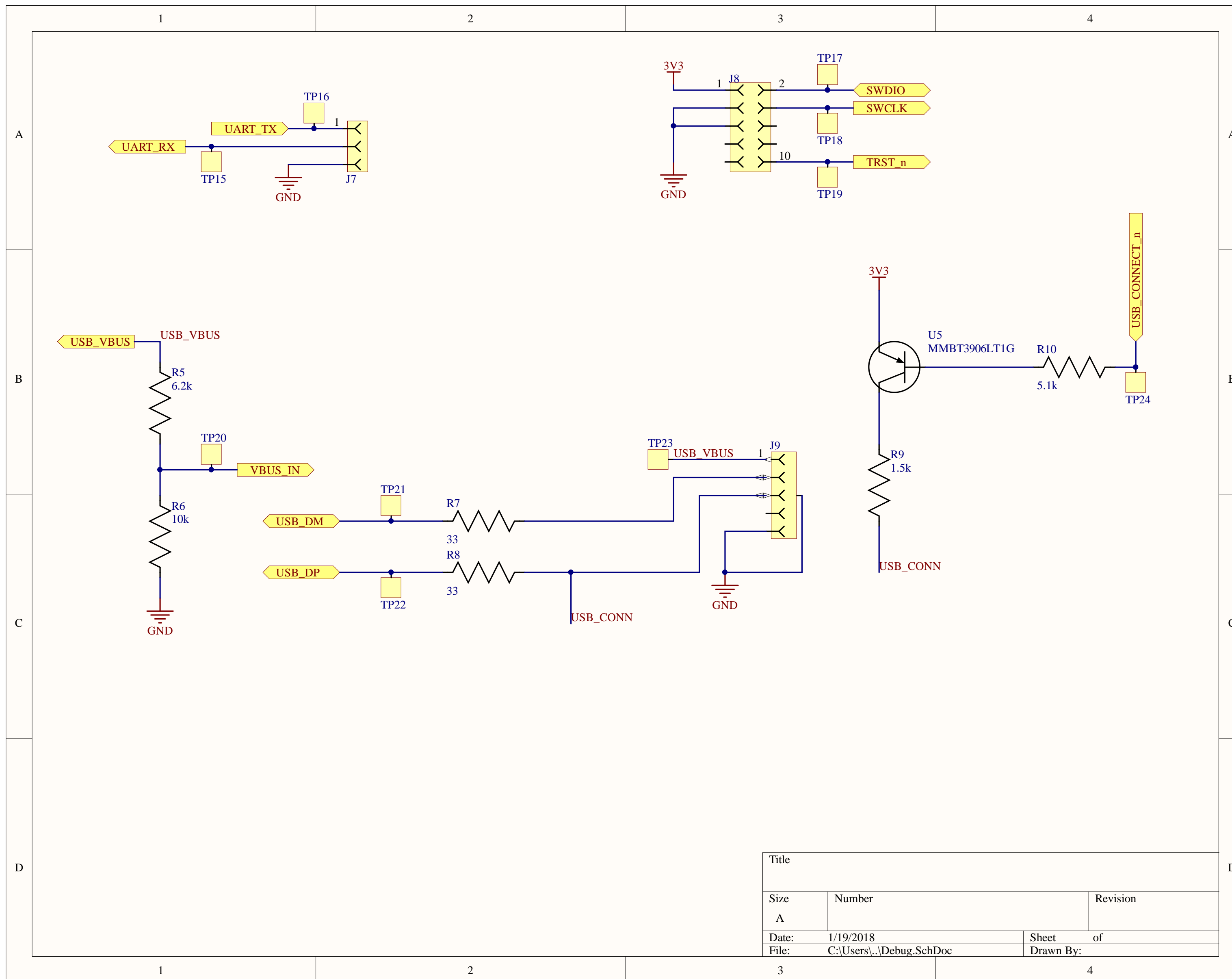

Power
Power.SchDoc
USB_VBUS
BEC_GOOD
MCU
MCU.SchDoc
BEC_GOOD
ENCODER_PULSE
BATT_SENSE_HIGH
BATT_SENSE_LOW
UART_TX
UART_RX
ESC_PWM
SERVO_PWM
SWDIO
SWCLK
TRST_n
XBEE_RST
XBEE_ATTN
XBEE_SSEL
VBUS_IN
USB_DP
USB_DM
USB_CONNECT_N
SPI_MISO
SPI_MOSI
SPI_CLK
I2C_SCL
I2C_SDA
GPIO_7
GPIO_6
GPIO_5
GPIO_4
GPIO_3
GPIO_2
GPIO_1
GPIO_0
Debug
Debug.SchDoc
UART_TX
UART_RX
SWDIO
SWCLK
TRST_n
USB_VBUS
VBUS_IN
USB_DP
USB_DM
USB_CONNECT_n
Sense
Sense.SchDoc
ENCODER_PULSE
BATT_SENSE_HIGH
BATT_SENSE_LOW
Actuation
Actuation.SchDoc
ESC_PWM
SERVO_PWM
Radio
Radio.SchDoc
XBEE_RST
XBEE_ATTN
XBEE_SSEL
SPI_MISO
SPI_MOSI
SPI_CLK
Expansion
Expansion.SchDoc
I2C_SCL
I2C_SDA
Title
Size
A
Number
Revision
Date: 1/19/2018
Sheet of
File: C:\Users\..\CarBoard.SchDoc
Drawn By:



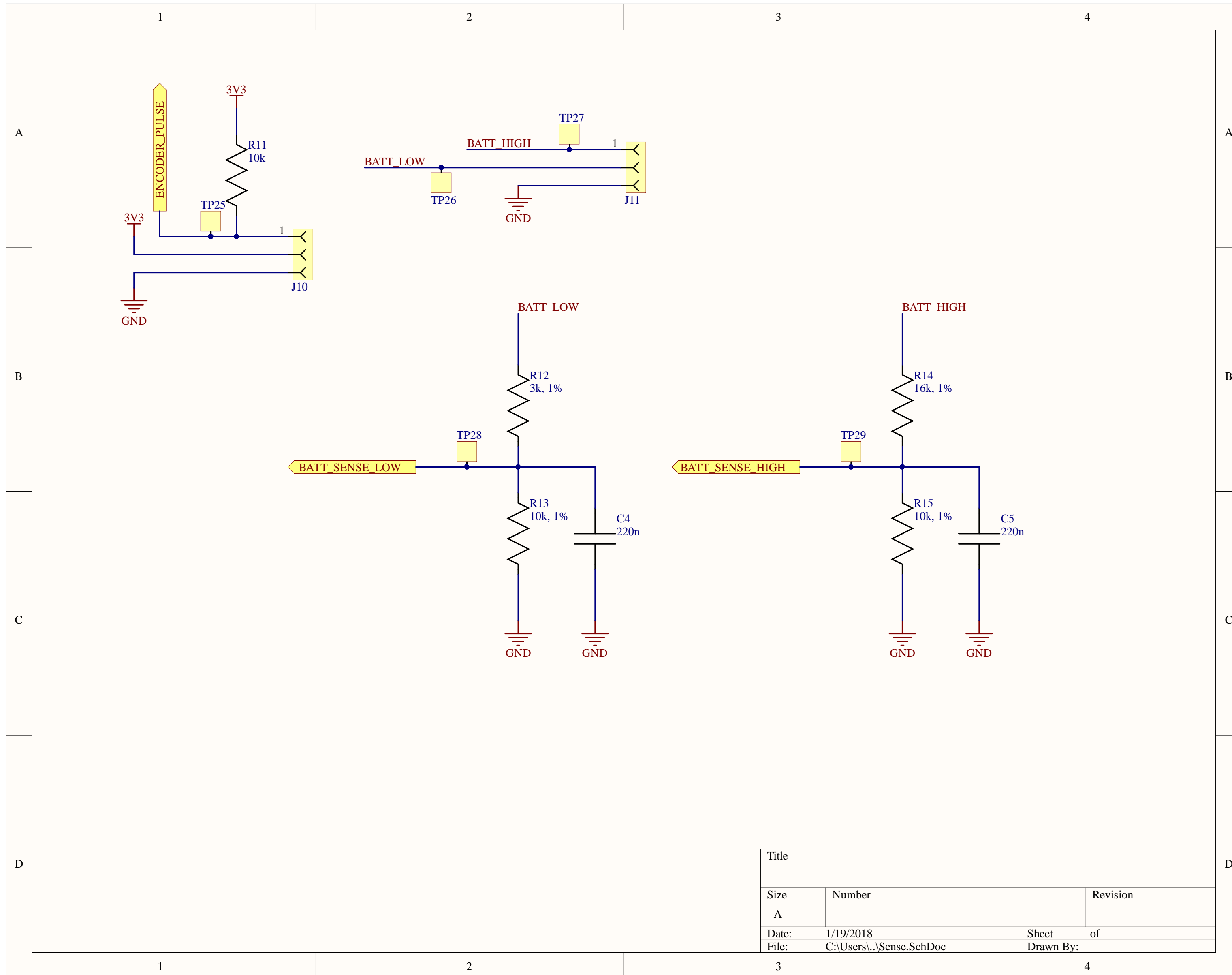
V_BATT
J1
TP1
J2
TP2
GND
5V0
3V3
V_BEC
TP3
TP4
TP5
V_BEC
R1
11k
3V3
R3
10k
J3
J4
BEC_GOOD
R2
10k
GND
V_BATT
5V0
U1
PG
EN
VIN
GND
GND
VOUT
5V_EN
D24V22F5
GND
GND
SW1
5V_EN
SW_SPDT
GND
USB_VBUS
D1
5V0
D2
5V_SEL
V_BEC
V_SEL
5V0
J5
3V3
U2
IN
OUT
OUT
GND
TLV1117LV33DCYR
5V_SEL
3V3
GND
Title
Size
A
Number
Revision
Date: 1/19/2018
File: C:\Users\..\Power.SchDoc
Sheet of
Drawn By:


XTALIN
XTALOUT
X1
12M
C1
18p
C2
18p
GND
U3
LPC11U3X
PIO1_0/CT32B1_MAT0/IOH_10
PIO1_25/CT32B0_MAT1
PIO1_19/DTR/SSEL1
RESET/PIO0_0
PIO0_1/CLKOUT/CT32B0_MAT2/USB_FTOGGLE
PIO1_7/IOH_17
VSS
XTALIN
XTALOUT
VDD
PIO0_20/CT16B1_CAP0
PIO1_10
PIO0_2/SSEL0/CT16B0_CAP0/IOH_0
PIO1_26/CT32B0_MAT2/RXD/IOH_19
PIO1_27/CT32B0_MAT3/TXD/IOH_20
PIO1_4/CT32B1_CAP0/IOH_14
PIO1_1/CT32B1_MAT1/IOH_11
PIO1_20/DSR/SCK1
PIO0_3/USB_VBUS/IOH_1
PIO0_4/SCL/IOH_2
PIO0_5/SDA/IOH_3
PIO0_21/CT16B1_MAT0/MOSI1
PIO1_17/CT16B0_CAP1/RXD
PIO1_23/CT16B1_MAT1/SSEL1
USB_DM
USB_DP
PIO1_24/CT32B0_MAT0
PIO1_18/CT16B1_CAP1/TXD
PIO0_6/USB_CONNECT/SCK0/IOH_4
PIO0_7/CTS/IOH_5
PIO1_28/CT32B0_CAP0/SCLK
PIO1_5/CT32B1_CAP1/IOH_15
VDD
PIO1_2/CT32B1_MAT2/IOH_12
PIO1_21/DCD/MISO1
PIO0_8/MISO0/CT16B0_MAT0/R/IOH_6
PIO0_9/MOSI0/CT16B0_MAT1/R/IOH_7
SWCLK/PIO0_10/SCK0/CT16B0_MAT2
PIO1_8/IOH_18
PIO0_22/AD6/CT16B1_MAT1/MISO1
PIO1_29/SCK0/CT32B0_CAP1
TDI/PIO0_11/AD0/CT32B0_MAT3
PIO1_11
TMS/PIO0_12/AD1/CT32B1_CAP0
TDO/PIO0_13/AD2/CT32B1_MAT0
TRST/PIO0_14/AD3/CT32B1_MAT1
PIO1_13/DTR/CT16B0_MAT0/TXD
VDD
PIO1_14/DSR/CT16B0_MAT1/RXD
PIO1_3/CT32B1_MAT3/IOH_13
PIO1_22/RI/MOSI1
SWDIO/PIO0_15/AD4/CT32B1_MAT2
PIO0_16/AD5/CT32B1_MAT3/IOH_8/WAKEUP
VSS
PIO1_9
PIO0_23/AD7/IOH_9
PIO1_15/DCD/CT16B0_MAT2/SCK1
VDD
PIO1_12
PIO0_17/RTS/CT32B0_CAP0/SCLK
PIO0_18/RXD/CT32B0_MAT0
PIO0_19/TXD/CT32B0_MAT1
PIO1_16/RI/CT16B0_CAP0
PIO1_6/IOH_16
3V3
RESET_n
ISP
XBEE_SSEL
XBEE_RST
VBUS_IN
I2C_SCL
I2C_SDA
USB_DM
USB_DP
USB_CONNECT_N
ENCODER_CNT
ENCODER_INT
XBEE_ATTN
SPI_MISO
SPI_MOSI
SWCLK
SPI_CLK
BATT_SENSE_LOW
BATT_SENSE_HIGH
TRST_n
ESC_PWM
SERVO_PWM
GPIO_0
GPIO_1
SWDIO
GPIO_2
GPIO_3
GPIO_4
BEC_GOOD
GPIO_5
UART_RX
UART_TX
GPIO_6
GPIO_7
TP6
SW2
TP7
SW3
R4
10k
J6
ENCODER_PULSE
C3
.1u
Title
Size
A
Number
Revision
Date: 1/19/2018
Sheet of
File: C:\Users\..\MCU.SchDoc
Drawn By:

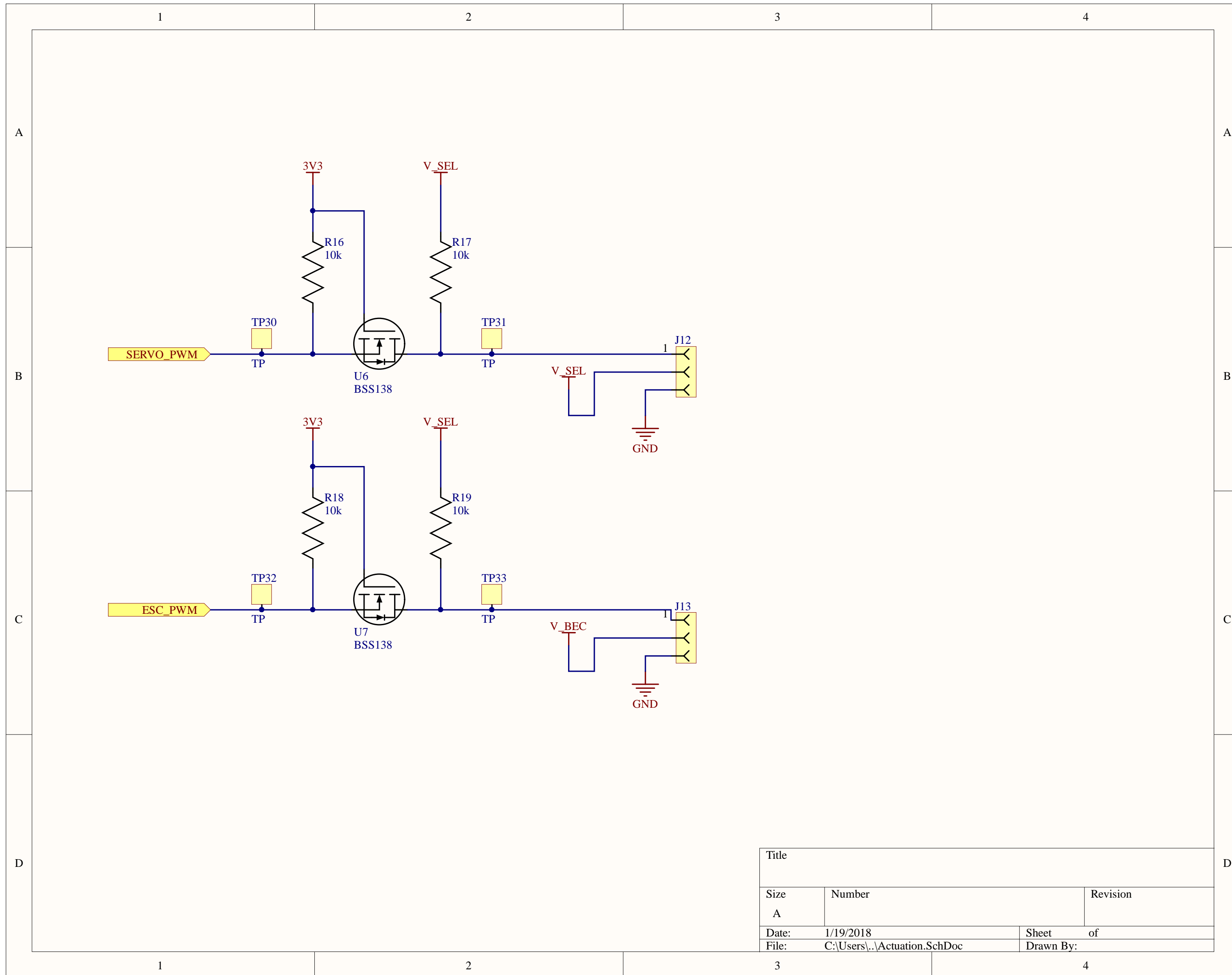

3V3
U4
VCC
DOUT/DIO13
DIN/CONFIG/DIO14
SPI_MISO/DIO12
RESET
DIO10/PWM0
DIO11/PWM1
Reserved
DTR/SLEEP_RQ/DIO8
GND
AD0/DIO0
SPI_ATTN/DIO1/AD1
SPI_CLK/AD2/DIO2
SPI_SSEL/AD3/DIO3
RTS/DIO6
Associate/DIO5
VREF
ON_SLEEP/DIO9
CTS/DIO7
SPI_MOSI/DIO4
XBEE-900HP
TP8
TP13
TP14
TP11
TP9
TP10
SPI_MISO
XBEE_RST
XBEE_ATTN
SPI_CLK
XBEE_SSEL
SPI_MOSI
GND
Title
Size
A
Number
Revision
Date: 1/19/2018
File: C:\Users\..\Radio.SchDoc
Sheet of
Drawn By:

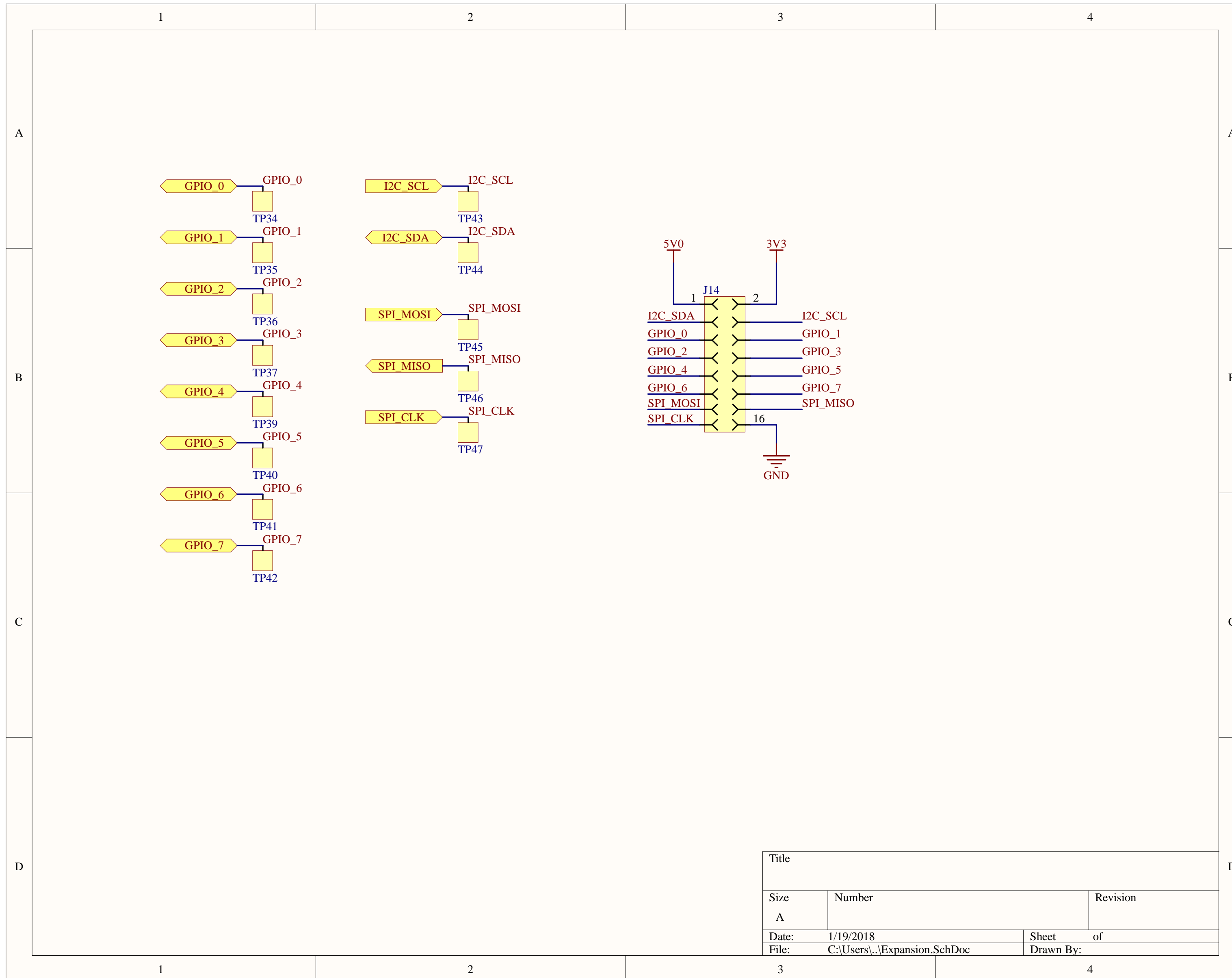


1
2
3
4
A
B
C
D
TP16
UART_TX
1
UART_RX
TP15
GND
J7
3V3
J8
1
2
TP17
SWDIO
SWCLK
TP18
10
TRST_n
TP19
GND
USB_CONNECT_n
3V3
U5
MMBT3906LT1G
R10
5.1k
TP24
USB_VBUS
USB_VBUS
R5
6.2k
TP20
VBUS_IN
R6
10k
GND
TP21
USB_DM
R7
33
R8
USB_DP
33
TP22
USB_CONN
TP23
USB_VBUS
J9
1
GND
R9
1.5k
USB_CONN
Title
Size
A
Number
Revision
Date:
1/19/2018
Sheet
of
File:
C:\Users\..\Debug.SchDoc
Drawn By:



ENCODER_PULSE
3V3
R11
10k
TP25
3V3
1
J10
GND
TP27
BATT_HIGH
1
BATT_LOW
TP26
GND
J11
BATT_LOW
R12
3k, 1%
TP28
BATT_SENSE_LOW
R13
10k, 1%
C4
220n
GND
GND
BATT_HIGH
R14
16k, 1%
TP29
BATT_SENSE_HIGH
R15
10k, 1%
C5
220n
GND
GND
Title
Size
A
Number
Revision
Date: 1/19/2018
Sheet of
File: C:\Users\..\Sense.SchDoc
Drawn By:


3V3
V_SEL
R16
10k
R17
10k
TP30
TP31
SERVO_PWM
TP
U6
BSS138
TP
1
J12
V_SEL
GND
3V3
V_SEL
R18
10k
R19
10k
TP32
TP33
ESC_PWM
TP
U7
BSS138
TP
1
J13
V_BEC
GND
Title
Size
A
Number
Revision
Date: 1/19/2018
Sheet of
File: C:\Users\..\Actuation.SchDoc
Drawn By:


1
2
3
4
A
B
C
D
GPIO_0
GPIO_1
GPIO_2
GPIO_3
GPIO_4
GPIO_5
GPIO_6
GPIO_7
TP34
TP35
TP36
TP37
TP39
TP40
TP41
TP42
I2C_SCL
I2C_SDA
SPI_MOSI
SPI_MISO
SPI_CLK
TP43
TP44
TP45
TP46
TP47
5V0
3V3
J14
1
2
16
I2C_SDA
GPIO_0
GPIO_2
GPIO_4
GPIO_6
SPI_MOSI
SPI_CLK
I2C_SCL
GPIO_1
GPIO_3
GPIO_5
GPIO_7
SPI_MISO
GND
Title
Size
A
Number
Revision
Date: 1/19/2018
File: C:\Users\..\Expansion.SchDoc
Sheet of
Drawn By:

Appendix B

Failure Mode Effects Analysis (FMEA)

#	Component	Potential Failure Mode	(1. Local 2. Next Level Up 3. System Level)	SEV MIN	SEV	Potential Cause(s) of Failure	OCC	DET	RPN
1	Car Tracker Camera	Hardware Failure	1. No frame available 2. Loss of tag (car) positions 3. Loss of car control		10	1. Broken camera 2. USB cable disconnected 3. Misconfigured focal length	5	6	21
2	Obstacle Tracker Camera	Hardware Failure	1. No frame available 2. Loss of obstacle positions 3. No obstacle-based behavior, such as avoidance		7	1. Broken camera 2. USB cable disconnected	3	6	16
3	IR Emitter	Hardware Failure	1. Scene not illuminator 2. Camera unable to detect tags 3. Loss of car control		10	1. Broken IR Emitter 2. Disconnected power cable	1	7	18
4	Base Radio	Hardware Failure	1. No radio link to cars 2. Car commands ignored 3. Loss of car control		10	1. XBee unplugged 2. Broken XBee board 3. USB cable disconnected	5	7	22
5	Encoder	Hardware Failure	1. No encoder pulses available 2. Loss of conveyor velocity 3. Degraded car control - steady state error		7	1. Broken encoder 2. Broken arduino board 3. Disconnected encoder-Arduino cable 4. Disconnected USB cable	1	8	16
6	Car Pose (/car/N/pose)	Intermittent Loss	1. Missing pose values 2. Missing command values 3. Degraded car control - jolt	5	9	Driving scenario: GPS signal jamming, poor signal, degraded GPS	1	1	11
7	Car Pose (/car/N/pose)	Nonsensical Data - Noise	1. Degraded estimation of car pose 2. Command values fluctuate even at steady state 3. Degraded car control - jolt and steady state error	5	7	Driving scenario: Poor GPS signal, degraded GPS System: Poor camera, filter calibration	1	8	16
8	Car Pose (/car/N/pose)	Transformation - Offset	1. Controller believes offset pose is actual pose 2. Command values compensate for perceived perturbation 3. Degraded car control - steady state error	5	10	Driving scenario: GPS spoofing attack System: Camera mount position offset	1	8	19
9	Car Velocity (/car/N/twist)	Intermittent Loss	1. Missing velocity values 2. Missing command values 3. Degraded car control - drift then jolt	5	9	Driving scenario: Degraded velocity sensor System: Poor camera, filter calibration	1	8	18
10	Car Velocity (/car/N/twist)	Nonsensical Data - Noise	1. Degraded estimation of car velocity 2. Nonsensical command values 3. Degraded car control - jolt and steady state error	5	7	Driving scenario: Degraded velocity sensor System: Poor camera or filter calibration, resolution too low	1	8	16
11	Car Velocity (/car/N/twist)	Transformation - Offset	1. Controller believes offset velocity is actual velocity 2. Command values compensate for perceived perturbation 3. Degraded car control - unbounded steady state error	5	10	Driving scenario: Attack on state estimator System: Poor camera or filter calibration	1	8	19
12	Car Control Urgency (/car/N/urgency)	Nonsensical Data - Injection	1. Controller believes emergency maneuvers need to be made 2. Aggressive command values 3. Degraded car control - overshoot or oscillation	7	8	Driving scenario: Fool system into evasion mode	1	1	10
13	Car Command (/car/N/command, /car/N/command_value)	Intermittent Loss	1. Missing command values 2. No command sent to car 3. Degraded car control - drift then jolt	5	9	Driving scenario: Faulty ECU	1	1	11
14	Car Command (/car/N/command, /car/N/command_value)	Transformation - Throttle Spike	1. Sudden increase in command values 2. Car surges forward or stops 3. Degraded car control - jolt	5	10	Driving scenario: CAN bus injection attack	1	1	12
15	Car Timeout Reset (/car/N/reset)	Nonsensical Data - Injection	1. Car controller gets reset 2. Car controller I term cannot accumulate 3. Degraded car control - steady state error	5	6	System: Misconfigured timeout interval	1	8	15
16	Car Threat Map (/car/N/cars/blobs)	Intermittent Loss	1. Missing car threat assessment 2. Car cannot react to fast moving surrounding cars 3. Potential collision with other cars		7	Driving scenario: Degraded sensor, poor conditions for sensor	1	1	9
17	Car Threat Map (/car/N/cars/blobs)	Transformation - Angle Offset	1. Car believes other cars are in wrong directions 2. Car reacts suboptimally to surrounding cars 3. Potential collision with other cars		7	Driving scenario: Mismounted or degraded sensor	1	1	9

18	Car Threat Map (/car/N/cars/blobs)	Transformation - Distance Offset	1. Car believes other cars are further away than they are 2. Car reacts suboptimally to surrounding cars 3. Potential collision with other cars		7	Driving scenario: Mismounted or degraded sensor	1	1	9
19	Obstacle Threat Map (/car/N/obstacles/blobs)	Intermittent Loss	1. Missing obstacle threat assessment 2. Car cannot react to fast obstacles 3. Potential collision with other obstacles		7	Driving scenario: Degraded sensor, poor conditions for sensor System: Poorly calibrated obstacle camera	1	8	16
20	Obstacle Threat Map (/car/N/obstacles/blobs)	Transformation - Angle Offset	1. Car believes obstacles are in wrong directions 2. Car reacts suboptimally to obstacles 3. Potential collision with other obstacles		7	Driving scenario: Mismounted or degraded sensor System: Poorly mounted obstacle camera	1	8	16
21	Obstacle Threat Map (/car/N/obstacles/blobs)	Transformation - Distance Offset	1. Car believes obstacles are further away than they are 2. Car reacts suboptimally to obstacles 3. Potential collision with other obstacles		7	Driving scenario: Mismounted or degraded sensor System: Poorly mounted obstacle camera	1	8	16
22	Conveyor Speed (/treadmill/velocity)	Intermittent Loss	1. Car controller acts on previous known conveyor speed 2. Car cannot react to changing conveyor speed 4. Degraded car control - Steady state error	5	6	System: Degraded encoder	1	8	15
23	Conveyor Speed (/treadmill/velocity)	Nonsensical Data - Noise	1. Car controller acts on degraded conveyor speed 2. Car controller applies unsuitable feedforward 4. Degraded car control - jolt, steady state error	5	6	System: Degraded or misconfigured encoder	1	8	15