

UNIVERSITY OF CALGARY

Black-box Behavioral Model Inference for Autopilot Software Systems

by

Mohammad Jafar Mashhadi Ebrahim

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA

SEPTEMBER, 2020

© Mohammad Jafar Mashhadi Ebrahim 2020

Abstract

Inferring behavior model of a running software system is quite useful for several automated software engineering tasks, such as program comprehension, anomaly detection, and testing. Most existing dynamic model inference techniques are white-box, i.e., they require source code to be instrumented to get run-time traces. However, in many systems, instrumenting the entire source code is not possible (e.g., when using black-box third-party libraries) or might be very costly. Unfortunately, most black-box techniques that detect states over time are either univariate, or make assumptions on the data distribution, or have limited power for learning over a long period of past behavior. To overcome the above issues, in this thesis, I proposed a hybrid deep neural network that accepts as input a set of time series, one per input/output signal of the system, and applies a set of convolutional and recurrent layers to learn the non-linear correlations between signals and the patterns, over time. I have applied this approach on two real UAV autopilot case studies: one from our industry partner, MicroPilot (MP in short), with half a million lines of C code, and one widely used open source solution: Paparazzi. I ran more than 1200 system-level tests in total (to generate the input data) and inferred the system's internal state, over time. In case of Paparazzi, as it did not include system tests like MP, I created a tool that generates and executes meaningful test scenarios. Comparison with several traditional time series change point detection techniques showed that this approach improves their performance by up to 102% in MP's case and 94% in Paparazzi's, in terms of finding state change points, measured by F1 score. I also showed that this state classification algorithm provides on average 90.45% F1 score for MP and 82.23% for Paparazzi, which improves traditional classification algorithms by up to 17%

in MP's case and 20% in Paparazzi's.

In addition, by creating a hyper-parameter tuning pipeline using grid search technique, despite having a way smaller training set in the second case study (7 times smaller compared to the first one), I managed to get a better performance, up to 48% better, out of the neural network model as measured by 8 metrics. The tuning performance is compared to using the same hyper-parameters that worked for MP's case, for Paparazzi.

Acknowledgments

I would like to appreciate the support from my graduate supervisor, Dr. Hemmati, and thank him for his support and smart and helpful advice over the course of my studies. I cannot forget to thank Afrouz, for being there for me and keeping my morales up at all times; Also for helping me proofread this thesis. Dr. Walkinshaw and Dr. Westwick for their input in parts of my research. The Moses, for helping me with make the charts and visualizations more appealing. Hamed, for sharing his knowledge of Machine Learning, recommending me useful talks, books, papers, etc. Adam, for his thorough answers to my questions about MP's autopilot. MicroPilot Inc. and NSERC for providing financial support for this research. The anonymous reviewers of the papers I published for their constructive comments. And finally, to my family and all the friends who supported me and made it a more pleasant experience.

To the ones who made this a smoother journey.

Table of Contents

Abstract	ii
Acknowledgments	iv
Dedication	v
Table of Contents	vi
List of Tables	viii
List of Figures	x
Contributions	xiii
1 Introduction	1
2 Motivation	5
3 Background and Related Work	8
3.1 State Model Inference	8
3.2 Automated Test Generation	10
3.3 Change Point Detection	11
3.4 Convolutional and Recurrent Neural Networks	14
3.5 Using Deep Learning on Time Series Data	15
3.6 Definitions	16
4 Hybrid Neural Network for State Inference	17
4.1 The Model Architecture	17
4.2 Data Encoding	19
4.2.1 Data Preprocessing	20
4.3 The Model Implementation	21
5 Empirical Evaluation	23
5.1 The Study Objectives	23
5.1.1 RQ 1) How does the proposed technique perform in detecting the state changes?	23
5.1.2 RQ 2) How well does the proposed technique predict the internal state of the system?	24

5.1.3	RQ 3) Can the results be replicated with regards to state change point detection (RQ1) on another case study?	24
5.1.4	RQ 4) Can the results of internal state prediction task (RQ2) be replicated on Paparazzi autopilot?	24
5.1.5	RQ 5) How will hyper-parameter tuning affect the results?	24
5.2	Evaluation Metrics	25
5.2.1	CPD Performance Metrics (RQs 1, 3, and 5)	25
5.2.2	State detection metrics (RQs 2, 4, and 5)	25
5.3	Comparison Baselines	26
5.3.1	CPD baselines (RQs 1 and 3)	26
5.3.2	Multi-class classification baselines (RQs 2 and 4)	26
5.4	Experiment Design	26
5.4.1	CPD (RQs 1 and 3)	27
5.4.2	Multi-class classification (RQs 2 and 4)	27
5.4.3	Hyper-parameter optimization (RQ 5)	28
5.4.4	Choice of Paparazzi for the case study	29
5.4.5	Experiment Execution Environment	29
5.5	Data Collection	29
5.5.1	MicroPilot Autopilot	30
5.5.1.1	About MicroPilot's Design	30
5.5.1.2	Instrumentation	30
5.5.1.3	Test Scenarios	31
5.5.2	Paparazzi	33
5.5.2.1	Instrumentation	33
5.5.2.2	Test Scenarios	33
5.5.3	Labeling The Data	35
5.5.4	The Testing and Data Processing Tool Set	36
5.5.4.1	Flight Data Recorder	37
5.5.4.2	Automated Test Executor	38
5.5.4.3	Automated Test Generator	41
5.6	Results	42
5.6.1	RQ1 Results: CPD Performance	44
5.6.2	RQ2 Results: Multi-class Classification Performance	45

- 5.6.3 RQ3 Results: CPD Performance on Paparazzi (replication) 47
- 5.6.4 RQ4 Results: Multi-class Classification Performance on
Paparazzi (replication) 49
- 5.6.5 RQ5 Results: Hyper-parameter Tuning 51
- 5.7 Limitations and Threats to Validity 55
- 6 Conclusion and Future Work 57**
- Bibliography 59**
- Appendix A: PprzTester documentations 72**
- Appendix B: Copyright 79**

List of Tables

5.1	The $n = 10$ collected I/Os of autopilot. The inputs are sensor readings and the outputs are the servo position update commands. All these I/Os over time are used as the inputs of the state prediction model. . .	31
5.2	Change point detection precision, recall, and F1-score calculated for the baseline methods using three values of tolerance (τ) for multiple configurations.	43
5.3	Change point detection precision, recall, and F1-score calculated, on the test data, for the proposed model, using three values of tolerance (τ) compared with the respective τ 's best F1 score among baseline methods	44
5.4	Precision, recall, and F1 score of ridge classifiers (linear classifiers with L2 regularization) and decision tree classifiers (DT) with different sliding window widths (w). For each algorithm on each w several hyper-parameters were applied producing 152 different models. In this table, I only show the results of the best performing model in each group.	46
5.5	Change point detection methods performance on Paparazzi dataset. Since the 100% recalls are actually outliers, the next largest recall values are in bold face as well.	50
5.6	Precision, recall, and F1 score of ridge classifiers (linear classifiers with L2 regularization) and decision tree classifiers with different sliding window widths (w). In this table, I only show the results of the best performing model in each group.	51
5.7	The performance results of the model trained on paparazzi before hyper parameter tuning (i.e. using the hyper-parameters that worked best for MP on Paparazzi as well) vs. after fine-tuning hyper-params on Paparazzi's dataset.	54

2	Subset of all available messages in Paparazzi that are used in PprzTester tool	73
4	Test runner command line arguments	74
6	Test runner command line arguments	75

List of Figures

4.1	The input and output signals of the black-box system are captured as a multivariate time series; they are processed in a deep neural network that consists of 3 sections: convolutional, recurrent, and dense (fully connected) to predict the system's internal state and its changes over time.	17
4.2	Model architecture in a nutshell. Tandem convolutional layers with increasing kernel size fed into two sequence-to-sequence recurrent layers with 128 GRU cells each, which is then fed into dense layers to output the predicted system state, as a list of one-hot encoded states. \hat{O} will be the result of applying argmax operation on the last layer's output. $L = 18000, N_s = 25$	22
5.1	Distribution of flight log lengths for the $N = 888$ (out of the original 948 available logs) logs that were kept in the dataset ($200 \leq l_k \leq 20,000$), from MicroPilot.	32
5.2	High level overview of communication links architecture in Paparazzi, including the components I added for this study's purpose. Diagram re-illustrated based on a diagram in Hattenberger et al. (2014) with some modifications.	34
5.3	The histogram of number of tests of different test sizes. The smallest samples contains 140 samples worth of recorded flight data and the largest one has 2,580 samples, equivalent of a 516-second flight (Sampling rate is 5Hz).	36
5.4	A snapshot of ground control station software (GCS), visualizing the map, waypoints (blue diamonds), the aircraft, its flight history (blue), the intended flight path (green), as well as its state (bottom in green) and some telemetry data (top left). The plan that it is following is one of the automatically generated test scenarios.	40

5.5	Evaluation of the model on 30 random test data. Each graph shows the states in one run of the system. The colors show the states. The top-half of each plot depicts model's prediction of the system states (\hat{O}) and the bottom-half shows the true labels(O). Since the output is one-hot encoded, the item with the most probability is used as the predicted label at each point in time. X-axis is the time axis. Only the first 600 samples (2 minute of simulation) are shown to improve legibility.	48
5.6	The states are color coded and each rectangle shows one test, only the first 1000 samples (200 seconds) are shown for more clarity. The same as figure 5.5 in, the top half of each rectangle shows the model's output and the bottom half shows the true labels.	52
5.7	Matrix of scatter plots comparing precision and recall of the deep learning model on test data.	53
5.8	Parallel coordinates view showing the effects of selected hyper-parameters on the performance measures.	53
1	Class diagram for flight data recorder	76
2	Flight plan items	77
3	Flight executor classes	77
4	Class diagram for helpers and wrappers for PprzLink library	78

Contributions

During this research, I have made contributions in form of publishing papers, datasets, contributing to the open source, etc. Since not all of them ended up in this thesis, I would like to quickly enumerate them so they will not go unnoticed:

1. ICPC '19 conference paper: An empirical study on practicality of specification mining algorithms on a real-world application [Mashhadi and Hemmati \(2019\)](#)
2. IST journal paper: Interactive semi-automated specification mining for debugging: An experience report [Mashhadi et al. \(2019\)](#)
3. ASE '19: presenting the above journal paper in journal-first-conference-second track
4. Salvaged a Java instrumentation tool that is used for white-box analysis: <https://github.com/sea-lab/JInstrumenter>
5. Developed a fuzz test generator/executor/flight data recorder tool specialized for Paparazzi: https://github.com/MJafarMashhadi/pprz_tester
6. ASE '20 conference paper: Hybrid Deep Neural Networks to Infer State Models of Black-Box Systems
7. Contributed 10 pull requests to Paparazzi during the development of the fuzz testing tool. They included adding new features, patches for fixing bugs, and adding to the documentations. My changes are included in Paparazzi's latest release.

Chapter 1

Introduction

Automated specification mining or model inference [Lo et al. \(2011\)](#) is the process of automatically reverse engineering a model of an existing software system. Behavioral models (e.g., state machines) are typically inferred from a running system by abstracting the execution traces. The inferred models are useful artifacts in many use cases where the actual behavior (abstracted as the inferred model) of the system is needed for analysis, such as debugging [Al-Sharif \(2009\)](#); [Shang et al. \(2013\)](#); [Mashhadi et al. \(2019\)](#), testing [Walkinshaw \(2018\)](#); [Schieferdecker \(2012\)](#); [Papadopoulos and Walkinshaw \(2015\)](#); [Dallmeier et al. \(2011\)](#), anomalous behavior detection [Valdes and Skinner \(2000\)](#), and requirements engineering [Damas et al. \(2005\)](#).

Inferring a behavior model of a system in a black-box manner is particularly interesting. In many real-world applications, the large-scale system is built by integrating many off-the-shelf libraries that are only available as binaries (no source code access). Thus, from a system's point of view, knowing the exact behavior of the system including all the interactions between black-box units are needed for most run-time analysis.

Model inference techniques are generally grouped into static and dynamic analysis categories. Static approaches use the code as their input data. They are able to gather all the required information such as function call graphs for example from the source code itself without a need to run the software. A typical dynamic model inference pipeline, on the other hand, starts with running existing tests in

the software to collect required data [Papadopoulos and Walkinshaw \(2015\)](#).

Most current behavioral model inference techniques are dynamic analysis methods (usually are more accurate than static analysis for run-time behavior inference) that require source code instrumentation to collect execution traces [Lo et al. \(2011\)](#). A common theme in them is that they use the data collected as the system functions in the wild. They either run existing test cases or instrument and inspect the system being used in production. This is to have a diverse collection of data that is also meaningful and representative of the actual system behaviour in common use cases. These methods are usually helpful in unit-level analysis where the instrumentation is not expensive and access to the code is allowed for the unit under study. However, in the system-level, thorough instrumentation is more expensive (even prohibitively expensive [Mashhadi and Hemmati \(2019\)](#)) and might not be even possible for some units (black-box libraries). Therefore, for use cases such as system-level anomaly detection, testing, and debugging a black-box behavior model inference that works on readily available input/outputs of the system is crucial.

In this thesis, I propose a dynamic analysis method to detect the internal state and the state changes in a black-box software system using deep learning. I collected the numerical values of the inputs and outputs of the system, in regular time intervals to create a multivariate time-series. A hybrid deep learning model (including convolution and recurrent layers) was then trained on these time-series to predict the state of the system at each point in time. The deep learning model automatically performs feature extraction making it way more effective and flexible compared to traditional methods. In addition, I do not make any assumption about statistical properties of the data which makes it applicable to a wide range of subjects.

I applied and evaluated this method on an autopilot software used in an Unmanned Aerial Vehicle (UAV) system developed by our industry partner, Winnipeg-based Micropilot Inc. In addition to that, I further replicated the results on another highly capable and widely used autopilot, Paparazzi [Hattenberger et al. \(2014\)](#), as well. I evaluated the method from two perspectives: how well the model can detect the point in time when a state change happens? (RQs 1 and 3: Change Point Detection (CPD)), and how accurately it can predict which state the system is in,

during the execution? (RQs 2 and 4: State Classification). I also explored how and to what extent does hyper-parameter tuning affect this method's performance in RQ 5.

Comparing the proposed approach with state-of-the-art alternatives, shows that it performs better in both change point and state detection. In MP's case study I observed 88.00% to 102.20% improvement in the F1 score of this CPD, compared to traditional CPD techniques. In addition, I saw a 7.35% to 16.83% improvement in the F1 score of state detection, compared to traditional classification algorithms on a sliding window, over the data. The same numbers for the other case study were 13% to 43% improvement in CPD and 77.20% to 87.97% improvement in state detection.

The contributions of this thesis can be summarised as:

- Introducing the first (to the best of my knowledge) deep learning architecture to infer behavior models from black-box software systems.
- Empirically evaluating the model and achieving very high accuracy compared to baselines using two real-world and large-scale case studies on a UAV autopilot system developed by our industry partner as well as an open source UAV autopilot.
- Developed an automated fuzz testing tool capable of generating and executing test cases for Paparazzi autopilot.
- Created a hyper-parameter tuning pipeline to optimize the performance of the deep learning model.

Note that I have made all the source code, models, execution scripts, and some of the data available online¹. Due to confidentiality, the Micropilot dataset cannot be shared publicly.

The rest of this thesis is organized as follows: In chapter 2 I explain further how and in which contexts this research can be beneficial. Then in chapter 3 I go through some background material and related papers this work is based on. In chapter 4, I present my proposed method and model. The way it was evaluated

¹<https://github.com/sea-lab/hybrid-net>

and the results are presented in chapter 5. I provide a summary of the paper briefly discuss some paths for future continuation of this work in chapter 6.

Chapter 2

Motivation

Black-box components are ubiquitous in software development. Reusing high-quality black-box units generally offers a better overall system quality and a higher productivity [Edwards \(2001\)](#). The black-box units can be as small as a reusable library, or as large as a framework (such as .NET before going open-source), or a complete piece of software such as a remotely hosted web service. There are also scenarios where the unit's source code is not black-box in general, but not accessible to a specific team that wishes to perform the dynamic analysis.

One particular interesting use case of system-level black-box analysis is inferring run-time state model of a control software systems, where inputs/outputs are signals to/from the system. These inputs/outputs are typically multivariate time series, which are already logged in such systems (no overhead for instrumentation). The goal is automatically detecting the high-level system state and its changes, over time.

As discussed in [section 1](#), we partnered with an autopilot manufacturer and performed this study on their autopilot software. The goal was to determine its internal state from its input/output signals, over time. In this scenario, the inputs are the sensor readings going into autopilot and the outputs are command signals sent to controller motors of the aircraft, showing autopilot's reaction to each input at each state. A state in this example is the high-level stage of a flight and a state change happens when the current input values in the current state trigger a constraint in the implementation that changes the way the output signals

are generated.

In this example, the training set will consist of input and output values recorded during one execution of the system, as a multivariate time series, along with state ids (as labels) per time stamp. One execution of the autopilot will be the whole flight process that may go through a “take-off” until a successful “landing”. Depending on the flight plan, autopilot goes through states such as “acceleration”, “take-off”, “climbing”, “turning”, “descending”, etc.

During a flight, the autopilot monitors changes in the input values and makes adjustments to its outputs in order to hold some invariants (predefined rules). For example, if it is in the “hold altitude” mode, it monitors the altimeter’s readings and when it goes out of the acceptable range, proportionate adjustments to the throttle or the nose pitch will be made to get it back to the desired altitude. This is basically how a typical feedback loop controller, such as PID or its variations work [Åström \(2008\)](#). When autopilot’s state changes from “hold altitude” to “descend to X ft” state, the set of invariants that autopilot is trying to hold are changed. It means its reactions to variations in inputs will be different. In this example, a decreasing altimeter reading will not trigger an increase in the throttle anymore.

Looking at the time series, a domain expert can identify what the state of autopilot is, at each point in time; the labeling process. Now the goal is to automate this task on a test set (in practice, future flights), assuming a training set is labeled by the experts (they only need to identify the state change time stamps, during a flight).

This problem can be tackled in two ways. The first solution is to identify the time stamp that the state change happens (i.e., Change Point Detection: RQ1); The more advanced solution is to predict the exact state per time stamp (i.e., State Classification: RQ2). The classic CPD techniques on time series [Truong et al. \(2018\)](#) are mainly applicable on univariate data or put assumptions on the input/output distributions, thus not applicable in this case with multivariate inputs and no assumptions or knowledge about the states’ distribution. The classic state classification techniques in time series are also weak in that they fail to balance between considering long-term relations or acting locally. The ones that use a sliding window, for example, do not have a long-term memory. The ones that act on the whole data on the other hand are too coarse-grained and

inaccurate for this task.

Therefore, the motivation for this study is to provide a black-box technique that can be applicable on both CPD and state classification problems, and overcome the limitations of the existing techniques, in terms of capturing the non-linear correlation between multivariate inputs and outputs as well as learning patterns over a long period of time. My proposal, which will be explained in detail in section 4, leverages the power of a deep neural network (DNN) with two types of layers that are particularly useful for this problem: a) convolutional layers which discover latent features from the data effectively through parameter sharing and b) recurrent layers that play a significant role in problems dealing with time series as they can learn long-term dependencies and seasonalities in the data.

This technique, after being trained, is useful for developers for debugging and also detecting anomalous behaviour and more. Labeling states in a input/output log is a common task in debugging process of these systems. However, it is a task that can benefit from automation. Currently, in debugging process the developers usually look at a subset of data and determine the states and state changes only in that part because it is not the quickest task but it is useful. An automated state detection technique can do this job accurately on the whole data thus making their workflow faster, more accurate, and more convenient.

Though the motivational example, as well as the case study, are from the UAV autopilot domain, the proposed method can be adapted to be applied to similar black-box control software systems in domains such as IoT, intelligent video surveillance, and self-driving cars.

Chapter 3

Background and Related Work

Unlike the numerous techniques in the literature for behavior model inference [Lang et al. \(1998\)](#); [Walkinshaw et al. \(2016\)](#); [Lo et al. \(2007\)](#); [Dallmeier et al. \(2006\)](#) which abstract a set of execution traces into states, my approach requires consuming a multivariate time-series and detect the state changes across time and predict the exact state labels. Thus, in this chapter, I briefly explain the existing techniques for “Change Point Detection” and “State Prediction” in time-series, “State Model Inference”, and “Automated Test Generation” that can serve as background for my approach.

3.1 State Model Inference

Dynamic model inference as a sub-field of specification mining is a vast field of research. There are model inference techniques for several different types of models that can be made for software systems, for example Damas et al. infer message sequence charts [Damas et al. \(2005\)](#), Lo et al. try inferring LSCs [Lo et al. \(2007\)](#), and Lemieux et al. explore mining LTLs [Lemieux et al. \(2015\)](#). However state models are a useful, popular, and widely researched type of model to infer. Two major categories of state models are finite state machines (FSM), and their extended version: EFSM. Extended Finite State Machines, are special kind of state machines that have conditional expressions called “transition guards” on their transitions [Lorenzoli et al. \(2008\)](#). A state transition can only happen if the

transition guard evaluates as true.

Roughly speaking, dynamic EFSM inference algorithms generally take a trace of “events” (along with perhaps some variable values) as their input [Walkinshaw et al. \(2016\)](#) to infer a generalized finite state machine. They use the events to find the state transitions and the values for detecting invariants and generating the guard conditions on the transitions. k-tails, gk-tail, EDSM, and MINT are examples of these algorithms, each improving upon the previous one [Biermann and Feldman \(1972\)](#); [Lorenzoli et al. \(2008\)](#); [Lang et al. \(1998\)](#); [Walkinshaw et al. \(2016\)](#). The data that is generated in software run time is quite varied; It can range from the logs, to function calls, to network packets, to syscalls, and so forth. There are several aspects to be considered for this matter: what kind of data is needed, how are they going to be collected, what kind of model we are looking to have, and how are they going to be used in model inference.

Walkinshaw et al., proposed an algorithm and developed a tool for state model inference [Walkinshaw et al. \(2016\)](#). Their work is based on previous endeavors on state merging algorithms such as gk-tail and k-tails [Lorenzoli et al. \(2008\)](#); [Biermann and Feldman \(1972\)](#). Their approach requires two collections of ordered events as input: a collection of positive examples which are events generated during a successful execution of the software as well as a set of negative examples. Each event there, is a function call; the log contains the list of functions that were called along with their parameters.

K-Tails algorithm [Biermann and Feldman \(1972\)](#) generates FSMs from execution traces which can be flexible in form. gk-tail takes it to the next level by adding transition guards to the state machine, making it an EFSM [Lorenzoli et al. \(2008\)](#). The input to gk-tail should include parameter values to be used for inferring the transition guards using Daikon [Ernst et al. \(2007\)](#).

As mentioned earlier, the input to dynamic model inference techniques can be of many different types. Other than the above examples that take function call logs as the execution trace, there are many other studies that used different types of input. For example, In Synoptic [Schneider et al. \(2010\)](#) the communications in a distributed system is being modeled as an automata and the events are the logs generated by its components. Howar et al. [Howar et al. \(2012\)](#) generate state models, with the events being high level actions such as ‘user registered’.

Krka et al., performed an empirical study on 4 different categories of model inference algorithms to figure out what makes each group of methods more effective [Krka et al. \(2014\)](#). Beschastnikh et al., proposed a method to mine invariants from partially ordered logs from concurrent/distributed systems [Beschastnikh et al. \(2012\)](#). Invariants can be used to augment state models [Beschastnikh et al. \(2014, 2011\)](#). Groz et al., use machine learning to heuristically infer state machine models of a un-resettable black-box system [Groz et al. \(2018\)](#), however a significant difference between my method and theirs is that their method still relies on discrete events (such as HTTP request and responses) while my method does not assume that the input and outputs contain any kind of “events” happening at certain times. My method aims to search for such events as change points in a continuous stream of data as time series. Papadopoulos et al., proposed an active learning method (with repeated interaction with a user) to model a black-box system and generate test data [Papadopoulos and Walkinshaw \(2015\)](#). In [Mashhadi et al. \(2019\)](#) a semi-interactive method for inferring state machines is proposed. It requires instrumenting the source code (white-box) and performs dynamic analysis of the system under study.

The important difference between these approaches and the problem I have in hand is in the input format. All these expect the input data to be in form of a list of ordered events; whereas in my approach I look at a continuous stream of data, and it is my job to detect the events (in form of state transitions) just by looking at the data.

3.2 Automated Test Generation

Automated testing is widely researched and ubiquitously used in the industry. Software test execution can be automated using the state of practice tools such as JUnit and Pytest. In such automated testing, a developer writes a test code that runs their source code, gives it inputs, and compares its outputs with the expected outputs to verify if the program is behaving correctly. However, in such practices the test generation and design is still manual and only the execution, evaluation and reporting is automated.

The next level in test automation is automated test generation, that is having

a tool that can automatically generate test code [Ibrahim et al. \(2007\)](#). There are several approaches in the literature for tackling this problem. Model-based testing approaches use specification models (inferred or hand crafted) to automate verification of requirements. For example, Papadopoulos et al.'s method [Papadopoulos and Walkinshaw \(2015\)](#) can generate test data using active learning. A similar paper is the work by Volpato and Tretmans [Volpato and Tretmans \(2014\)](#) that use state models for test generation.

Search-based testing approaches, use well-known search algorithms such as evolutionary algorithms to generate test cases and test data. Random testing approaches generate randomized input data for testing. For example, evo-suite is a widely used test generation tool that uses evolutionary algorithms under the hood [Fraser and Arcuri \(2011\)](#) to generate unit tests for Java applications. DART and Randoop [Godefroid et al. \(2005\)](#); [Pacheco and Ernst \(2007\)](#) generate random data to test the software. Feedback-directed random test generation is an step of improvement over naïve random test generation techniques [Pacheco et al. \(2007\)](#). Fuzz testing is a similar approach to random test generation. DeepFuzz uses fuzzing to generate syntactically valid C code to catch bugs in C compilers [Liu et al. \(2019\)](#). Fairfuzz [Lemieux and Sen \(2018\)](#) monitors branch coverage of generated inputs to guide fuzzing in a way that rare branches are also covered adequately. For this thesis, I created an automated test generation and execution tool to have more test data for one of the case studies. This tool is tailored specifically for that software, minding all the invariants, valid input ranges, the protocol and communication format, etc.

3.3 Change Point Detection

A fundamental tool in time-series data analysis is Change Point Detection (CPD). It refers to the task of finding points of abrupt change in the underlying statistical model or its parameters that could be a result of a state transition [Aminikhanghahi and Cook \(2017\)](#). It is a well-studied subject due to its wide range of applications [Basseville et al. \(1993\)](#). A multitude of statistical and algorithmic methods have been tried to tackle several variations of CPD problem [Chen and Gupta \(2011\)](#); [Hasan et al. \(2014\)](#); [Hsu \(1982\)](#); [Lee and Lee \(2017\)](#); [Oh and Kim \(2002\)](#); [Ramos](#)

et al. (2016); Chowdhury et al. (2012); Reeves et al. (2007); Rosenfield et al. (2010); Wang et al. (2011); Xie and Siegmund (2013); Yamanishi et al. (2004); Lavielle (1999); many of which perform effectively on a subset of CPD problems with some assumptions. The assumptions can be of various types. For example, one may assume the time series has only one input variable (univariate) Fryzlewicz et al. (2014), there is only one changing point Bai et al. (1998), or the number of change points is known beforehand Lavielle (2005), or they might assume some statistical properties on the data Chen and Gupta (2011); Takeuchi and Yamanishi (2006); Idé and Tsuda (2007). These are limiting factors, since many of these assumptions do not necessarily hold in this case. CPD techniques are categorized into two main groups: a) online methods that process the data in real-time and b) offline methods that start processing the data after receiving all the values Truong et al. (2018). Since my model inference use case of CPD can afford waiting to collect all historical training data, I considered only the offline techniques.

Ives and Dakos utilized locally linear models and used statistical significance test to determine at which point the changes in model parameters are large enough to signal a change in the state Ives and Dakos (2012). Blythe et al., used subspace analysis to reduce data dimensionality to keep the most non-stationary dimensions. This process helps detecting change points more effectively Blythe et al. (2012). Several techniques have used penalty functions to find models that best fit each segment of the signal Lavielle (1999, 2005); Keshavarz et al. (2018); Pein et al. (2017); Khan et al. (2019). Desobry et al., and Hido et al., proposed methods to indirectly use classifiers such as SVM to detect change points Desobry et al. (2005); Hido et al. (2008); Khan (2019). I applied their approach on the data in hand in early stages of the research but it could not perform as others. Lee et al., trained deep auto encoder networks that learns latent features in the data to detect change points Lee et al. (2018). Ebrahimzadeh et al., proposed what they call a pyramid recurrent neural network architecture, which is resilient to missing to detect patterns that are warped in time Ebrahimzadeh et al. (2019). There is also a family of methods based on Bayesian models that focus on finding changes in parameters of underlying distributions of the data Lee et al. (2018); Adams and MacKay (2007); Bai (1997); Barry and Hartigan (1993); Erdman and Emerson (2008); Ray and Tsay (2002).

Making assumptions about the data such as its distribution or the distribution of change points across the time and relying on basic statistical properties are the two major short comings of traditional CPD methods [Lee et al. \(2018\)](#), which my proposed approach has overcome.

In general, CPD algorithms consist of two major components: a) the search method and b) the cost function [Truong et al. \(2018\)](#). Search methods are either exact or approximate. For instance, Pelt is the most efficient exact search method in the CPD literature, which uses pruning [Killick et al. \(2012\)](#). Approximate methods include window-based [Basseville et al. \(1993\)](#), bottom-up [Keogh et al. \(2001\)](#), binary segmentation [Scott and Knott \(1974\)](#), and more. In the window-based segmentation a sliding window is rolled over the data and then sum of costs of left and right half-windows is subtracted from the cost of the whole window. When the difference gets significantly high it means that the discrepancy between left and right half of the window is high and therefore a change point probably lies right in the middle of the window. In the bottom-up method, the input signal is split into multiple smaller parts, then using a similarity measure adjacent segments are merged until no more merges are feasible. The binary segmentation method finds one change point and splits the input into two parts around that point and then recursively applies the same method on each part.

The cost functions are also quite various, from simply subtracting each point from the mean to much more complex metrics, such as auto-regressive cost functions [Angelosante and Giannakis \(2012\)](#), and kernel-based cost functions. Kernel-based costs can have a wide variety, since the kernel function can be almost arbitrary, however a handful of them such as linear and Gaussian kernels are among the most popular ones [Truong et al. \(2018\)](#).

In the context of this thesis, we need a CPD method with no assumption on data distribution, number of change points, etc. In addition, my CPD method should work on multivariate data, and be able to capture non-linear relations between signals. It also needs to be resilient to time lags between an input signal change and its effect on the output signal (and the systems state). There is no traditional CPD algorithms that covers all these requirements. Therefore, I propose a novel CPD techniques that is based on Hybrid DNNs and compare it with several existing CPD techniques as comparison baselines, which are explained in

details in section 5.

3.4 Convolutional and Recurrent Neural Networks

In both my problems (CPD and state classification), one can see that the changes in signals are more informative than their absolute values. Therefore, applying a derivation operation (or more generally a gradient) seems like necessary, at some point in the processing. Farid and Simoncelli listed some discrete derivation kernels in their study [Farid and Simoncelli \(2004\)](#), but to have a more generalized and more flexible notion of discrete derivatives, convolutions seems like a better choice to apply. Nowadays, applying convolutional filters on signals is pretty much a standard process in signal processing studies that leverage deep learning [Morales and Roggen \(2016\)](#); [Zeng et al. \(2014\)](#); [Yang et al. \(2015\)](#). Convolutional neural networks (CNNs) can learn to find features in a multidimensional input while being less sensitive to the exact location of the feature in the input [LeCun et al. \(2015\)](#). In the forward pass of a convolutional layer, multiple filters are applied to the input. It means that in a trained neural net, multiple features can be learned in one single convolutional layer.

Recurrent neural networks (RNN) have shown great performance in analysing sequential data such as machine translation, time-series prediction, and time-series classification [Cho et al. \(2014\)](#); [Zhang and Xiao \(2000\)](#); [Wang et al. \(2017\)](#); [Murad and Pyun \(2017\)](#); [Yang et al. \(2015\)](#); [Ordóñez and Roggen \(2016\)](#). RNNs can capture long-term temporal dependencies which is quite useful to solve this problem. [Che et al. \(2018\)](#) For example, they might learn that “climb” state in a UAV autopilot usually follows “take off”. Therefore, while it is outputting “take off” it anticipates what the next state will probably be and as soon as its input features start shifting, it detects the onset of a state change. It will help the model to better predict the system’s behavior and be quicker to detect state changes in a way that could hardly be achieved with classic methods. Therefore, in this thesis, I combine the CNNs and RNNs to create what is known as a hybrid deep neural network [Wang et al. \(2017\)](#) to use for both CPD and state classification problems, in this context.

3.5 Using Deep Learning on Time Series Data

Human activity recognition (HAR) is a well researched task which is quite relevant to the problem of black-box model inference. In HAR, just like in this context, a multivariate time series data is created from various sensors on a human body. The goal is to figure out what was the activity that human was performing in different time intervals. The sensors can be body worn accelerometers, or more generic sensors such as the ones found in a smart watch or a smartphone. Murad et al., [Murad and Pyun \(2017\)](#) have shown deep RNNs outperform fully convolutional networks and deep belief networks in HAR task. Hybrid models are the combination of some deep architectures [Wang et al. \(2019\)](#), such as a CNN + RNN or a CNN + a fully connected net. Morales et al., have shown the former performs better than the latter in HAR [Morales and Roggen \(2016\)](#). Yao et al., [Yao et al. \(2017\)](#) introduced a CNN + RNN architecture that outperforms the state of the art both in classification and in regression tasks. Similar results have been shown in other works such as [Ordóñez and Roggen \(2016\)](#); [Singh et al. \(2017\)](#); [Zheng et al. \(2016\)](#), as well.

Another related topic here is the time series classification. However, time series classification techniques often output only one label classifying entire data, thus not applicable in this context. What is more related to my problem is called “segmentation”, using the computer vision terminology (not be confused with time series segmentation, such as [Lemire \(2007\)](#)). U-net is one of the promising auto-encoder architectures for image segmentation [Ronneberger et al. \(2015\)](#). Perslev et al., developed a similar idea for time-series to capture long-term dependencies and called it U-time [Perslev et al. \(2019\)](#). It is fully convolutional and does not use memory cells (recurrent cells). A fully convolutional model can perform very well, since convolutions operate locally and image segments are large chunks of pixels in the 2D space and capturing local features using neighbouring pixels is quite useful. However, it cannot necessarily be as powerful on a more limited 1D data of time-series with different characteristics from an image. U-time’s design is optimized for the task of sleep phase detection, which does not have very clear boundaries between states and also the state changes are quite infrequent. Therefore, the same method does not necessarily generalize to tasks such as mine,

where I cannot make assumptions about frequency of state changes.

3.6 Definitions

Software Model: "A model is a representation in a certain medium of something in the same or another medium."..."The model has both semantics and notation and can take various forms that include both pictures and text. The model is intended to be easier to use for certain purposes than the final system." [Rumbaugh et al. \(2004\)](#)

Program State: "The finite state machine introduces a concept of a state as information about its past history. All states represent all possible situations in which the state machine may ever be. "...The history of input changes required for clear determination of the state machine behavior is stored in an internal variable State."[200 \(2006\)](#) **Software State Model:** An abstract model of a software that represents high-level behaviour of the software in relation to its inputs. Finite state machines, a graph of system states and the transitions between them, are a common example of behavioural state models.

Artificial Neural Network (ANN): A mathematical function composed of other smaller functions (neurons and layers) that map its input to the output. The function parameters are optimized during "training" phase to find the best fitting function to the (X, y) examples. [Goodfellow et al. \(2016\)](#)

Convolutional Neural Network (CNN): A ANN that has convolutional layers. Convolutional layers apply a convolution operation on their input. This operation is a weighted average of a walking window of input's elements with weights being the convolution kernel matrix.

Recurrent Neural Network (RNN): A ANN that uses recurrent cells, which are cells that their output depends both on the current input and its internal state matrix.

Hybrid Deep Neural Network: A neural network model that combines two other architectures such as CNN + RNN, CNN + dense, RNN + RBE, CNN + RBE, etc.

Chapter 4

Hybrid Neural Network for State Inference

In this section, I describe my proposed deep learning approach for the black-box state inference task, in details.

4.1 The Model Architecture

The goal of this study is to infer the states of a running software system, over time. Given that my assumption is we don't have access to the source code (or part of it), I only leverage the values of inputs and outputs of the system, over time. As can be seen in figure 4.1, I capture all the inputs and outputs of the system as a time

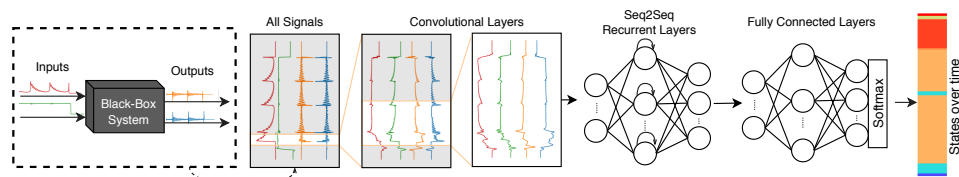


Figure 4.1: The input and output signals of the black-box system are captured as a multivariate time series; they are processed in a deep neural network that consists of 3 sections: convolutional, recurrent, and dense (fully connected) to predict the system's internal state and its changes over time.

series and then process it in a DNN. The architecture of my proposed model is a hybrid DNN which is inspired by models proposed in the field of Human Activity Recognition (HAR). This task is quite similar to the subject of this thesis in the sense that they both take in a multivariate time series-data (from sensor readings) and output the state of the system that generated those readings (see section 3.5 for more details on HAR papers). This DNN is made of three parts in sequence: 1) Convolutional, 2) Recurrent, and 3) Fully connected layers. This architecture addresses the aforementioned traditional methods' challenges; each part serves a different purpose in this process, as follows.

Convolutions, being more generalized than simple sliding windows, can discover patterns and features in the signals, both in temporal and in spatial (how signals affect each other) dimensions [Wang et al. \(2017\)](#). The convolutional layers' flexibility allows them to learn some typical preprocessing operations. For example a moving average or a discrete derivative can be learned as simple convolutional filters. They also help the model to be more resilient to varying time delays between noticing a deviation in input signals and the reaction that will appear in the output signals. Applying convolutional layers in sequence has been shown to result in each layer learning more complex features than the previous layers [Zeiler and Fergus \(2014\)](#). The number of layers, filters, and the kernel size are hyper-parameters that should be selected based on the size of data and the complexity of the system being modeled. Using a sequence of convolutions with a) increasing number of filters and the same kernel size, b) same number of filters and increasing kernel size, and c) decreasing filters with increasing kernel sizes are all different approaches that have been used in the literature by well-known architectures such as VGG and U-net [Simonyan and Zisserman \(2014\)](#); [Ronneberger et al. \(2015\)](#). I will discuss more details of CNN layers in my approach in Section 4.3.

Convolutions are quite powerful in discovering local features. To capture long-term features, recurrent layers which learn sequences of data are leveraged. For example, in my case, they can learn that "accelerate" and "take off" states only happen in the start of the states sequence, and each "take off" state is usually followed by a "climb" state. The type of recurrent cell to use (LSTM, GRU, etc.), how many cells to unravel in the layer, and the number of layers are also hyper-

parameters that need to be tuned depending on the size and complexity of the system under study.

Finally, one or more dense (fully connected) layers in the end are a common way of reducing the dimensions to match expected output dimensions. If there are only two states, the last layer can have a sigmoid activation function and be of shape L (the length of the input), otherwise, to match the one-hot encoding of labels, an output of shape $L \times N_s$ with softmax activation along the second axis (N_s) is required (N_s being the number of possible states).

In terms of loss function to optimize in the training process, a good choice is a dice overlap loss function, which is used in image semantic segmentation tasks, as well. An important property of this loss function is not getting negatively affected by class imbalances [Milletari et al. \(2016\)](#); [Sudre et al. \(2017\)](#).

4.2 Data Encoding

The input/output values of the black-box system create a multivariate time-series (T_k), which can be defined as a set of n univariate time series (V_i) of the same length l_k . Each V_i corresponds to the recorded values for one of the inputs or outputs of the system:

$$T_k = \{V_{1k}, V_{2k}, \dots, V_{nk}\} \quad (4.1)$$

$$|V_{1k}| = |V_{2k}| = \dots = |V_{nk}| = l_k \quad (4.2)$$

Note that, as figure 4.1 shows, we take both inputs and outputs as part of the time-series data to be fed as input into my deep learning models. This is to make sure that we can model state-based behavior of the system, where the current state depends not only on the inputs, but also on the last state(s) (captured as previous outputs) of the system. As an example, from the case study, if the outputs are not taken into account a mid-flight “descend” state and the “approach” state right before landing are indistinguishable, using the sensor readings (inputs) alone.

Having such a time-series, the only remaining pieces from a training set are the labels. Unlike the input/output values (the features in the dataset) the labels are not usually given. My method to infer the labels is a supervised approach. Thus, we need the domain expert to manually label each individual time stamp with

a state name/ID. In practice, what they would do is to identify the approximate time that a state change happens and assign the new state to one of the previous states labels or define a new label for this new state. Thus I encode the states information over time as a set of tuples in the form of (t_s, s) where t_s denotes the timestamp where the system entered state s . We show the set of all possible states with $S (s \in S)$ and define N_s as the cardinality of this set.

$$\begin{aligned} CP_k &= \{(t_{s_1}, s_1), (t_{s_2}, s_2), \dots, (t_{s_l}, s_l)\}, s_i \in S \\ N_s &= |S| \end{aligned} \quad (4.3)$$

So in summary, the dataset consists of N pairs of the I/O values as features and their state information as labels $\{(X = T_k, y = CP_k) | 1 \leq k \leq N\}$.

4.2.1 Data Preprocessing

Before being fed into the model \mathcal{F} (as defined below), the inputs and labels need some preprocessing.

$$\mathcal{F}(\delta(T), m): \mathbb{R}^{L \times n} \times \mathbb{R}^L \rightarrow S^L. \quad (4.4)$$

To run more efficiently, TensorFlow expects all the inputs to have the same length. To do that, the shorter T_k s should be zero-padded to length $L = \max\{l_k\}$. The padding function δ does that. Therefore, eventually, the input to the model will be T_k s that are rearranged to form a tensor of shape $n \times L$ along with a padding mask (denoted with m). The mask tells the model where the tail starts so the model can ignore all the zeros from there on.

$$\begin{aligned} \hat{O} &= \langle \hat{o}_i \in S \rangle_{i=1}^L = \mathcal{F}([\delta(V_1)^T \delta(V_2)^T \dots \delta(V_n)^T], m) \\ m &= \delta(\mathbb{1}_l) \quad \text{i.e.} \quad \langle m_j \rangle_{j=1}^l = 1, \langle m_j \rangle_{j=l+1}^L = 0 \end{aligned} \quad (4.5)$$

Here l denotes the length of the input before padding. It is equal to l_k for the k th training data (T_k).

As defined in (4.3), CP_k s are tuples of (t, s) which indicate the system have gone into state s at time t . To train the model, CP_k needs to be expanded into a vector of length L denoted by O where each element o_t holds the state at time t . To define it formally, the elements can be derived from CP_k using the following

formula:

$$\begin{aligned}
 O &= \langle \forall t \in \mathbb{N}_L : s_i \mid (t_{s_i}, s_i) \in CP_k \wedge \\
 &\quad t_{s_i} = \max\{t_{s_j} \mid (t_{s_j}, s_j) \in CP_k \wedge t_{s_j} \leq t\} \rangle
 \end{aligned}
 \tag{4.6}$$

For example: Suppose $L = 10$ and $CP = \{(0, a), (3, b), (5, c), (8, a)\}$ then $O = \langle a a a b b c c c a a \rangle$. If there are more than two possible states ($N_s > 2$), O needs to be one-hot encoded, at this stage.

4.3 The Model Implementation

The first few layers of the model are convolutional layers. I used 5 convolutional layers with 64 filters each and a growing kernel size. The intuition behind this design is that starting with a small kernel guides the training in a way that the first layers learn simpler more local features that fits in their window (kernel size). Kernel sizes started with 3 since it is a common number in the literature for kernel sizes, then I used multiples of 5 from 5 to 20. The rationale behind choosing 5 is because the sampling frequency is 5, so each layer with a kernel size of $5n$ processes a whole n seconds worth of simulation data, in each step. Stopping at kernel size of 20 was a compromise between generalizability and model size. Generally, a larger model has more learning capacity, but it is also more prone to over-fitting. The current models are the smallest I could make the models (to avoid over-fitting), without compromising the performance.

Same compromise was made in the second section of the model (Recurrent layers), the sweet spot for hyper-parameters here was to use two GRU layers with 128 cells each. Their output was fed into a fully connected layer with 128 neurons with a leaky ReLU ($\alpha = 0.3$) activation function [Maas et al. \(2013\)](#) and finally to a dense layer with $N_s = 25$ units with softmax activation. I used Adam optimizer [Kingma and Ba \(2014\)](#) that could converge in 60-80 epochs, i.e. validation accuracy plateaued. The full architecture can be seen in figure 4.2.

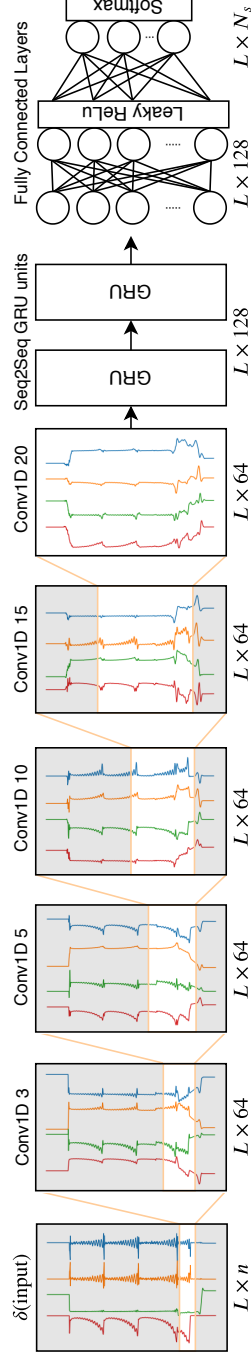


Figure 4.2: Model architecture in a nutshell. Tandem convolutional layers with increasing kernel size fed into two sequence-to-sequence recurrent layers with 128 GRU cells each, which is then fed into dense layers to output the predicted system state, as a list of one-hot encoded states. \hat{O} will be the result of applying argmax operation on the last layer's output. $L = 18000$, $N_s = 25$

Chapter 5

Empirical Evaluation

In this section, I explain my empirical evaluation process of the proposed approach through two case studies.

5.1 The Study Objectives

The goals of this study is to evaluate the proposed method in terms of change point detection and state inference, in comparison to traditional techniques in this domain. Therefore, the research questions are as follows:

5.1.1 RQ 1) How does the proposed technique perform in detecting the state changes?

The goal of this RQ is to see how close the predicted state-change times are to the real state-change times. In other words, in RQ1, we do not predict the exact state labels and are only interested in predicting the change. To answer this question, I compare the performance of my proposed approach with several traditional baselines (see 5.3.1), in terms of modified precision, recall, and F1 scores that are introduced in section 5.2.1, in the context of an industrial autopilot case study.

5.1.2 RQ 2) How well does the proposed technique predict the internal state of the system?

In RQ1, we are only interested in detecting the time a state-change happens (binary classification), but here in RQ2, I extend that and are also interested in predicting the label of the new state that the system is going into (multi-class classification). Therefore, to answer this RQ, I change the labels from a Boolean (changed/not changed) to the actual collected labels. The evaluation of this RQ is again in the context of the industrial autopilot case study.

5.1.3 RQ 3) Can the results be replicated with regards to state change point detection (RQ1) on another case study?

In the first two research question, I evaluate my proposed model inference technique on the data collected from our industry partner, MicroPilot. However, to assess the generalizability of the findings, a replication on a similar software is quite helpful. In this RQ and the next one I want to explore how my method performs on Paparazzi, an open source equivalent to MP¹'s autopilot.

5.1.4 RQ 4) Can the results of internal state prediction task (RQ2) be replicated on Paparazzi autopilot?

I fed the data to a number of classic machine learning algorithms as baselines. The problem setting is a multi-class classification, though with Paparazzi, the number of classes are smaller. There are 20 possible states in Paparazzi as opposed to MicroPilot's 25. This is due to their design differences in defining a mission and controlling an automated vehicle (the aircraft) to perform it.

5.1.5 RQ 5) How will hyper-parameter tuning affect the results?

This method is sensitive to the choice of hyper parameters. They need to be tuned for each case since the same values that worked well for one case study will not necessarily work well with another one. Hyper-parameters include the number of convolutional layers, number of convolutional filters in each layer, number

¹MicroPilot

of recurrent cells, and optimizer parameters such as learning rate. There are no gold standards for the values of these parameters, they need to be tuned for each problem.

5.2 Evaluation Metrics

5.2.1 CPD Performance Metrics (RQs 1, 3, and 5)

Given that in RQ1 there is an inherent class imbalance (there are far more points where a change has *not* happened compared to points with a state-change positive label), I avoid using accuracy and report both precision and recall. However, the original precision/recall metrics require some modifications due to the difficulty of predicting the exact time stamp that a state-change happened. To handle this, similar to related work [Truong et al. \(2018\)](#), I use a tolerance margin τ . If a detected state-change ($\in \hat{CP}_k$) is within $\pm\tau$ of a true change ($\in CP_k$), we call the prediction a True Positive, otherwise it is a False Positive. Similar adjustment to definition is applied for True Negative and False Negative. Formally speaking, I define predicted change points for k -th sample as:

$$\hat{CP}_k = \{(t, \hat{o}_t) \mid \hat{o}_t \neq \hat{o}_{t-1}\} \quad (5.1)$$

Please note that in (5.1), \hat{o}_t refers to t -th element of output vector \hat{O} , as previously defined in (4.5). Based on that the confusion matrix elements are calculated as:

$$\begin{aligned} TP &= \left| \{(\hat{t}, \hat{s}_t) \in \hat{CP}_k \mid \exists (t, s_t) \in CP_k \text{ s.t. } |t - \hat{t}| < \tau\} \right| \\ FP &= \left| \{(\hat{t}, \hat{s}_t) \in \hat{CP}_k \mid \nexists (t, s_t) \in CP_k \text{ s.t. } |t - \hat{t}| < \tau\} \right| \\ FN &= \left| \{(t, s_t) \in CP_k \mid \nexists (\hat{t}, \hat{s}_t) \in \hat{CP}_k \text{ s.t. } |t - \hat{t}| < \tau\} \right| \end{aligned} \quad (5.2)$$

With these in mind, I measure precision, recall, and their harmonic mean F1 Score with three values for τ : 1, 3, and 5 seconds. The smaller the tolerance is the stricter the definitions become and the lower the numbers are.

5.2.2 State detection metrics (RQs 2, 4, and 5)

In RQ2, we have a multi-class classification problem and thus multiple precision-s/recalls will be calculated, one per class (state label). I then report the mean

value across all classes.

$$\begin{aligned}
 P_s &= \{ \hat{s}_t \in \hat{O}_k \mid \hat{s}_t = s \} \\
 T_s &= \{ s_t \in O_k \mid s_t = s \} \\
 TP_s &= \{ \hat{s}_t \in P_s \mid \hat{s}_t = s_t \in O_k \}
 \end{aligned}
 \tag{5.3}$$

$$Precision = \frac{1}{N_s} \sum_{s=1}^{N_s} \frac{|TP_s|}{|P_s|} , \quad Recall = \frac{1}{N_s} \sum_{s=1}^{N_s} \frac{|TP_s|}{|T_s|}$$

5.3 Comparison Baselines

5.3.1 CPD baselines (RQs 1 and 3)

I used ‘ruptures’ library developed by authors of a recent CPD survey study [Truong et al. \(2018\)](#). It provides a modular framework for applying several CPD algorithms to univariate and multivariate data. As mentioned earlier two main elements of a CPD algorithm in their survey are the search method and the cost function. I tried every possible combination of these two.

5.3.2 Multi-class classification baselines (RQs 2 and 4)

I used Scikit-learn’s implementation of the classification algorithms: A ridge classifier (Logistic regression with L2 regularization) and three decision trees. The ridge classifier was configured to use the built-in cross validation to automatically chose the best regularization hyper-parameter α in the range of 10^{-6} to 10^6 . Each decision tree was regularized by setting “maximum number of features” and “maximum depth”.

5.4 Experiment Design

So far, we have a good understanding and a big picture about the design of experiments conducted in this study. In this section I provide the details on experiment configurations as well as my rationale on specific design choices that require some more explanation.

5.4.1 CPD (RQs 1 and 3)

There are three search methods implemented in “ruptures” library which were suitable to use in my experiments. Pelt [Killick et al. \(2012\)](#) is the most efficient exact search method. Two other ones are approximate search methods: bottom-up segmentation and window-based search method. After trying to run Pelt algorithm on MicroPilot’s data, I realized that it takes prohibitively longer to run compared to the approximate methods without providing much better results, so I only use the bottom-up and the window-based segmentation methods, as the CPD baselines in RQ1. Fortunately, for RQ 3 with smaller dataset size of Paparazzi, it was feasible (though still really time-consuming) to try “Pelt” as well, making the replication question richer in that sense. When using window-based search method, I left the window size parameter with the default size of 100 [Keogh et al. \(2001\)](#).

For the cost function, I tried “Least Absolute Deviation”, “Least Squared Deviation”, “Gaussian Process Change”, “Kernelized Mean Change”, “Linear Model Change”, “Rank-based Cost Function”, and “Auto-regressive model change” as defined in the library. Their parameters were left as default. To optimize the number of change points a penalty value (linearly proportionate to the number of detected change points) is added to the cost function, which limits the number of detected change points, the higher the penalty the fewer reported change points. We tried three different ratios (100, 500, and 1000) for the penalty.

5.4.2 Multi-class classification (RQs 2 and 4)

To prepare the data to be fed to the classification algorithms, I used a sliding window of width w over the 10 time-series values and then flattened it to make a vector of size $10w$ as the features. For the labels, I used one-hot encoded state of the system. The window sizes were chosen as same as the sizes of convolutional layers’ kernel sizes (3, 5, 10, 15, 20), to make the baselines better comparable with my method.

As stated already, the ridge classifier was configured to use the built-in cross validation to automatically chose the best regularization hyper-parameter α . To regularize the decision trees I tried: no limits, $\sqrt{10w}$, and $\log_2 10w$ for “maximum

number of features” regularization parameter. To find best “maximum depth” I first tried having no upper bound and observed how deep the tree grows; then I tried multiple numbers less than the maximum, until a drop in performance was observed.

In RQ4 I used the same configurations and procedures as the MP’s case (RQ2), with the only difference being on removing the depth limit from the decision trees. Tuning the depth limit was an arduous and inaccurate task that resulted in minimal improvements (if any, as will be seen in the RQ2’s answer later on), so it was not worth the time. Overall, I ran $(1 + 3) \times 6 = 24$ different settings for classic learning algorithms to answer RQ4.

5.4.3 Hyper-parameter optimization (RQ 5)

I designed a model creation and evaluation pipeline that takes hyper-parameters as the input and outputs the model performance scores on test data (a.k.a. tuning data) as its output. The hyper-parameters that I searched over are:

- Number of GRU layers: 1 or 2
- Number of GRU cells in the recurrent section: 5 values between 64 and 512
- Number of convolutional filters in each layer: 5 values between 16 to 72
- The size of convolution kernels and the number of convolutional layers: between 3 to 6 layers with increasing kernel size, starting from kernel sizes 3 or 5
- The learning rate of Adam optimizer: 3 values from 3×10^{-4} to 3×10^{-3}

Please refer to figure 4.2 for a recap on these hyper-parameters. I performed a grid search over these parameters, using Tensor Board for keeping track of the metrics and finding the right balance. Tensor Board is a monitoring tool made for TensorFlow that provides great insight for better training TensorFlow models. In total, there were 520 configurations that were used to train models on training dataset and evaluated on test (tuning) dataset.

5.4.4 Choice of Paparazzi for the case study

Paparazzi [Hattenberger et al. \(2014\)](#) project started in 2003 as an academic autopilot and continues to be developed with the state of the art in the autonomous flying vehicle's field. Another major player in open source autopilot software scene is ArduPlane; however I chose to do this study only on Paparazzi for the following reasons: Paparazzi is a more capable and well designed solution in general. A detailed comparison about how Paparazzi is superior to ArduPlane can be read at https://wiki.paparazziuav.org/wiki/Paparazzi_vs_X. In addition to that, after doing a preliminary study, I found out that Paparazzi has a more straightforward and robust protocol for remote controlling and data collection, as will be explained in detail in section 5.5.2. Furthermore, Paparazzi supports multiple flight dynamic model (FDM) simulators. One of them is JBSim² which provides an advanced physical model of complex dynamics in air-frames and sensors for an accurate and close to the reality simulation.

5.4.5 Experiment Execution Environment

Training and evaluation of the deep learning model was done on a single node running Ubuntu 18.04 LTS (Linux 5.3.0) equipped with Intel Core i7-9700 CPU, 32 gigabytes of main memory, and 8 gigabytes of GPU memory on a NVIDIA GeForce RTX 2080 graphics card. The code was implemented using Keras on TensorFlow 2.0 [Abadi et al. \(2015\)](#).

The baseline models could not fit on that machine, so two nodes on Compute Canada's Beluga cluster, one with 6 CPUs and 75GiB of memory and one with 16 CPUs and 64GiB of memory, were used to train and evaluate them.

5.5 Data Collection

In this section more details on how data was collected from each system will be presented.

²<http://jsbsim.sourceforge.net/>

5.5.1 MicroPilot Autopilot

5.5.1.1 About MicroPilot's Design

MicroPilot's autopilot is a commercial autopilot with a codebase of 500k lines of C code. MicroPilot is the world-leader in professional UAV autopilot which develops both hardware and software for 1000+ clients (including NASA, Raytheon, and Northrop Grumman) in 85+ countries during the past 20+ years.

The primary control mechanism in the autopilot is a hierarchy of PID loops, as explained both in chapter 2 and 4. High level commands are loaded on the autopilot as a flight plan. The flight plan looks like “takeoff, climb to 300 feet, go to waypoint A, go to waypoint B, land”. These commands determine which PID loops must be activated and what should their ‘desired values’ be. For example, a “go to waypoint A” command activates a PID loop that tries to minimize the distance between current location of the aircraft and point A. This is a high level loop that activates other lower-level PID loops to achieve its goal. Those lower level loops can be one to maintain the altitude and another loop that keeps the aircraft heading on the straight line from current location to point A. This hierarchical chain of ‘higher-level goals controlling lower level ones’ goes on, down to the level that directly controls aerodynamic surfaces of the aircraft.

5.5.1.2 Instrumentation

Control decisions in this software are made in a 5Hz loop, it means that every 200ms all the sensor inputs are read and based on the current state of the aircraft and the system's goal at the moment (e.g. maintaining a constant speed) decisions will be made and output is generated. Considering this, the best way to capture those data is in the end of each iteration of this loop. I inserted instrumentation code there, to log input and output values (listed in Table 5.1) at the exact spot where they are updated. Please note that although it is more convenient to capture the values in this way, it does not give us any special advantage or insight that breaks the black-box condition. In other words, the exact same data could be collected from the compiled binaries without any access to the internals, just with extra steps. Inputs and outputs, after all, are the very least thing available in both

Table 5.1: The $n = 10$ collected I/Os of autopilot. The inputs are sensor readings and the outputs are the servo position update commands. All these I/Os over time are used as the inputs of the state prediction model.

Inputs	
Pitch	The angle that aircraft's nose makes with the horizon around lateral axis
Roll	The angle of aircraft's wings make with the horizon around longitudinal axis
Yaw	The rotation angle of aircraft around the vertical axis
Altitude	AGL ⁴ Altitude
Air speed	Speed of the aircraft relative to the air
Outputs	
Elevator	Control surfaces that control the Pitch
Aileron	Control surfaces that control the Roll
Rudder	Control surface that controls the Yaw
Throttle	Controller of engine's power, ranges from 0 to 1
Flaps	Surfaces of back of the wings that provide extra lift at low speeds, usually used during the landing

black-box and white-box settings.

5.5.1.3 Test Scenarios

MicroPilot has a repository of 948 system tests, I ran them in a software simulator³ and collected the logged flight data, over time. The test cases are system-level tests. Each test case includes a flight scenario for various supported aircraft. A flight scenario goes through different phases in a flight such as “take off”, “climb”, “cruise”, “hitting way points”, and “landing”. Out of the 948 flight logs, I omitted 60 that were either too short or too long (shorter than 200 samples or longer than 20k samples). Figure 5.1 shows the distribution of the remaining log lengths. The maximum length (L) was 18,000 samples.

³It is developed by MicroPilot and provides an accurate simulation of the aerodynamic forces on the aircraft, the physical environment irregularities (e.g. unexpected wind gusts), and noises in sensor readings

⁴Above Ground Level

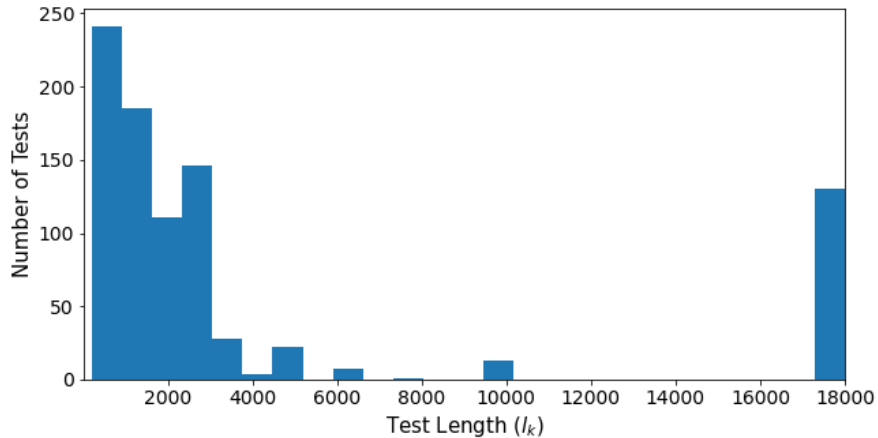


Figure 5.1: Distribution of flight log lengths for the $N = 888$ (out of the original 948 available logs) logs that were kept in the dataset ($200 \leq l_k \leq 20,000$), from MicroPilot.

The data was randomly split into three chunks of 90%, 5%, and 5% for training, validation, and testing, where each sample corresponds to one test execution. Note that separate test and validation sets are needed to facilitate proper hyperparameters tuning, without leaking information.

Please note that these test scenarios are simulated on multiple different airframes with very different aerodynamics and even different types of control surfaces (some do not have a rudder for example); I did not group the data based on the airframe. I did this to reduce the chance of overfitting and increase generalizability of the trained model; however it seems obvious (although it did not try it) that if one does that the resulting models would work better for the specialized airframes they will be trained on.

The autopilot can be used in SWIL and HWIL modes [Melmoth et al. \(2019\)](#), which stand for software in the loop and hardware in the loop respectively. I used SWIL mode as it provides what was needed without any of the costs and hassles that come from HWIL mode.

5.5.2 Paparazzi

5.5.2.1 Instrumentation

Paparazzi provides a rich and flexible API that can be configured to record several different parameters in flight. The aircraft periodically sends data back to the ground station over a wireless link using a protocol called Paparazzi link. Paparazzi link is built over Ivy, a message bus protocol that uses UDP. In Paparazzi's architecture a process called 'link' interfaces the wireless link to the aircraft to the computer's network; on one side are the Paparazzi link messages that come and go as UDP datagrams and on its other side is the (often wireless⁵) connection to the aircraft. In simulations, the modem and wireless communications are no longer needed, instead the autopilot runs as a separate process and mimics a wireless channel over the local network. (See Figure 5.2)

Paparazzi comes with a multitude of small tools that could do most of what I needed in terms of instrumentation. There is a remote logger and a log player which are quite close to the instrumentation tool I need, however upon trying them in action, I figured that they cannot record some of the information that I need. Therefore, I developed a custom flight data recorder tool.

5.5.2.2 Test Scenarios

Unlike MicroPilot that had a large a number of system tests (in addition to other types of tests such as unit tests which I did not use), Paparazzi comes with only unit tests. As it is an open source software under GPL licence it comes with no warranty and also does not need certain certifications and approvals that commercial systems require, therefore incentives to have such tests are lower. Although it is a reliable and widely used autopilot, it owes that reliability more to its widespread use in action (by many researchers and enthusiasts) rather than automated tests that verify its behaviour. In this situation, where many eyes are watching over the code, bugs are discovered and patched quickly. However, to the best of my knowledge they are not recorded as system tests that verify the bugs are properly patched and detect regressions, in the future.

⁵A wired connection is used in HWIL test mode as well as some scenarios where the autopilot equipment is used in a autonomous submarine rather than an autonomous unmanned aircraft.

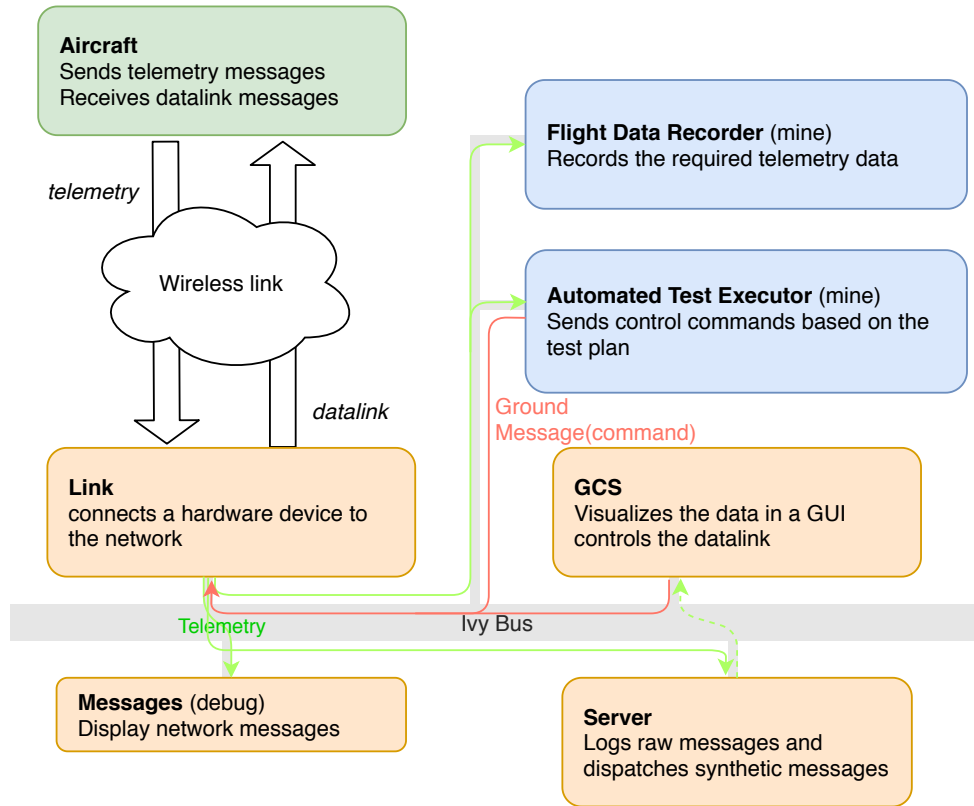


Figure 5.2: High level overview of communication links architecture in Paparazzi, including the components I added for this study’s purpose. Diagram re-illustrated based on a diagram in [Hattenberger et al. \(2014\)](#) with some modifications.

To fill the void, I created a fuzz testing tool that can automatically generate valid, diverse, and meaningful automated system tests for Paparazzi based on the example flight plan that is included with it. My tool can automatically generate system tests, run them in a simulator (or on hardware⁶), and also collect required telemetry data from the aircraft. It is called PprzTester and the source code, version history, project planning data (bugs, enhancements, tasks and issues, etc), and the documentations are available on GitHub at https://github.com/MJafarMashhadi/pprz_tester. The targeted randomizations in test in-

⁶Although I have not tested running tests on a hardware (HWIL) to confirm, but having implemented the protocol it potentially is capable of doing so

puts are augmented with the stochastic wind model in the simulation to further diversify the observed behaviours.

In addition to that tool, I needed to patch some parts of Paparazzi to make the logging and testing more similar to MicroPilot, for example increase telemetry reporting rate from 2Hz to 5Hz. A list of these patches including the reason why that change was necessary or beneficial and the exact lines of code that need to be changed is available in the project wiki at https://github.com/MJafarMashhadi/pprz_tester/wiki/Paparazzi-Patches. Aside from the patches, I found some bugs, missing features, missing documentations, and bad smells in the code that needed to be fixed. I contributed new code and documentation to the Paparazzi project to address these issues. The contributions were useful, up to the standards, and welcome in the project; they are included in the latest release⁷ of Paparazzi autopilot: version 5.16.

More details about the tool is left for next section, section 5.5.4. The result of generating and running tests was 378 runs worth of different flight scenarios. After collecting the data I performed some pre-processing steps on them to make them more similar to what the previous model was trained on. These pre-processing steps include normalizing some values as well as metric to imperial unit conversions.

Figure 5.3 shows the distribution of the test lengths. The test lengths range from 140 samples (70 seconds) to 2,580 samples, with a median of 2,170 and a mean of 1896.

The data was split into 3 chunks after shuffling: 70% of the data was used for training, 20% was used as the test set for tuning the hyper-parameters, and the remaining 10% was set aside as the validation data to measure the trained model's performance.

5.5.3 Labeling The Data

In the empirical study to evaluate this approach, I use the source code to collect the exact time a state-change happens and the actual state labels (ground truth). However, in practice, labeling the training set is supposed to be done by the

⁷as of August 3rd, 2020

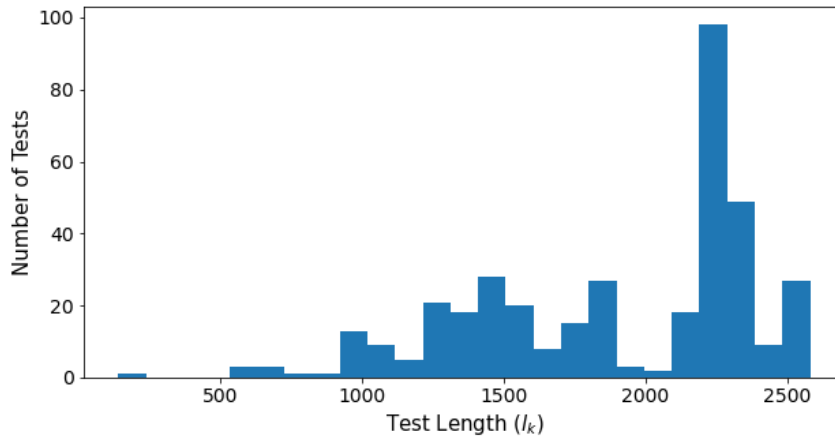


Figure 5.3: The histogram of number of tests of different test sizes. The smallest samples contains 140 samples worth of recorded flight data and the largest one has 2,580 samples, equivalent of a 516-second flight (Sampling rate is 5Hz).

domain expert in a black-box manner. This is not an infeasible task or extra overhead. Monitoring the logs and identifying the current system state is in fact part of the developers/testers regular practice during inspection and debugging. All that is provided here is a tool that given a partial labeling (only on the training set), automatically predict the state labels and the state-change times, for future flights. Also note that even though I use the source code to label the training set, I still look at the test set as a black-box and do not leak any information.

A few examples of what the states might be are: Accelerate Take-off rotation, Climb, Cruise, Fixed-altitude Turn to right/left, Approach, Flare, Decelerate, and more.

5.5.4 The Testing and Data Processing Tool Set

I implemented the pipeline of generating tests, running them, and aggregating flight logs in an integrated tool with three components, one for each stage. In the following sections I will explain these components and their role in the system. I also very briefly explain how event-driven programming paradigm was

implemented here both to decrease coupling and to make it more resilient to unexpected or buggy behaviour. It is necessary for two reasons: 1. as a testing tool, it is *expected* to encounter bugs in software under test and should be resilient to them 2. the message passing design of Paparazzi architecture, in addition to its ability to support multiple aircraft in flight at the same time left me with no choice but to use an event-driven design.

5.5.4.1 Flight Data Recorder

Although Paparazzi comes with a logging feature in its ‘server’ component (Figure 5.2), it only logs a subset of required data. Furthermore, the data comes in separate messages with different frequencies; for example speed updates are sent in AIRSPEED message once a second, orientation updates and engine rpm are reported in ATTITUDE and ENGINE_STATUS messages respectively which are dispatched every 200 milliseconds, while servo outputs are only sent once every 5 seconds. All these data need to be aggregated and aligned. Please refer to the Table 2 in [Appendix A](#) for a full list of messages used.

The most flexible option with the least overhead is to have an independent module that understands Paparazzi link, captures these messages and does data aggregation and logging in real-time. So I created a flight data recorder to collect all the required data from telemetry messages (See Figure 5.2).

In the beginning the instance of “AircraftManager” singleton class starts with listening for NEW_AIRCRAFT messages. This message type notifies this class when new aircraft come online in the simulation. Then it sends a AIRCRAFTS_REQ request message to server to get a list of currently online aircraft. The response to both of these messages are processed in the same way: The aircraft unique ID will be checked against the hash table of known aircraft, if it is a new one an “Aircraft” instance will be created and added to the hash table. (See the class diagram in Figure 1 in [Appendix A](#))

Each “Aircraft” object in the makes a number of requests back and forth with other components in the system (server, data link, and the autopilot process) to gather required information about that aircraft, including its flight plan. This class along with “Aircraft Parameters” and “Aircraft Commands” provide a unified

programmatic API for monitoring and controlling the aircraft.

I implemented observer pattern [Gamma \(1995\)](#) in “Aircraft Parameters” to enable other components (such as flight data recorder and automated test executor) to listen for changes in the aircraft state and respond accordingly in an event-driven manner. After creation of an “Aircraft” object, as a part of its initialization, several observers are created and attached to it. A “Record Flight” instance is one of them. It observes new FLIGHT_PARAM messages as well as changes in ‘throttle’, ‘flight time’, and ‘commands.values’ parameters. This class stores and aligns these parameters in a pandas data frame which is stored on disk periodically. The data can be stored in a human-readable CSV file or as a compressed HD5 binary. Some data normalization and unit conversions (such as meters to feet) also happen before saving in order to make the generated data similar to MicroPilot’s.

Supplementary UML diagrams are provided in [Appendix A](#).

5.5.4.2 Automated Test Executor

Here I first explain how the test executor work, since knowing this is prerequisite of understanding the test generator component. The test executor takes the control of the aircraft by running prefabricated test scenarios. While the low level control of the aircraft is done by the autopilot software (the system under test), it needs high level commands such as “climb to 200ft”. Automated test executor does that, using hand crafted test scenarios or the ones generated by the test generator component.

Tests scenarios are defined as subclasses of PlanBase class. Each plan needs to override a method that returns an iterable of plan items which should be executed one by one.

```
class ExamplePlan(PlanBase):  
    def get_items(self, kwargs):  
        block_name = kwargs.pop('block_name')  
        circles = int(kwargs.pop('circles'))  
        return [items.JumpToBlock(block_name),  
                items.WaitForCircles(n_circles=circles)]
```

The above plan for example, takes a block name and a number of circles as its parameter and executes the two items in succession. An example of running this plan can be like the command below:

```
$ run_test.py ExampleAircraft ExamplePlan\  
    -Dblock_name='loiter' -Dcircles=2
```

Using this parameters is analogous to test parameterization in mature unit testing frameworks (such as `pytest`). It allows similar test plans that are only different in some parameters to be consolidated in one test case. Using parameters also improves the reproducibility of test scenarios while improving its flexibility. Test scenarios (plans) do not need to include commands for waiting for aircraft to take off and land, the testing tool automatically wraps it with appropriate initialization code.

The core test plan runner is implemented as an observer class that listens for multiple messages and parameters to get notified about changes in the aircraft's state (See Figure 3 in [Appendix A](#), also Figure 5.2). It iterates over the flight plan items and calls their `match` method to decide whether that item should be executed. Whenever a match is found the item will be executed and the iterator will move to the next test plan item. Consult Figure 2 in [Appendix A](#) for a class diagram of all flight plan items.

Test plans should be importable from `pprz_tester.generated_plans` module. The structure is simple and clearly defined so that they can easily be hand-crafted (like the example above) or generated using the test generator component.

The test runner is tailored specifically for Paparazzi in several ways:

1. As mentioned before, it takes care of aircraft initialization, take off, and landing
2. It uses standard Paparazzi environment variables such as `$PAPARAZZI_HOME`.⁸
3. It can build the autopilot if `--build` argument is set so the user does not have to build it manually in Paparazzi center.

⁸If it is not set, the user can use `-p /path/to/paparazzi` command line argument to set this value, and if that is not set too, a default value will be used.

5. Empirical Evaluation

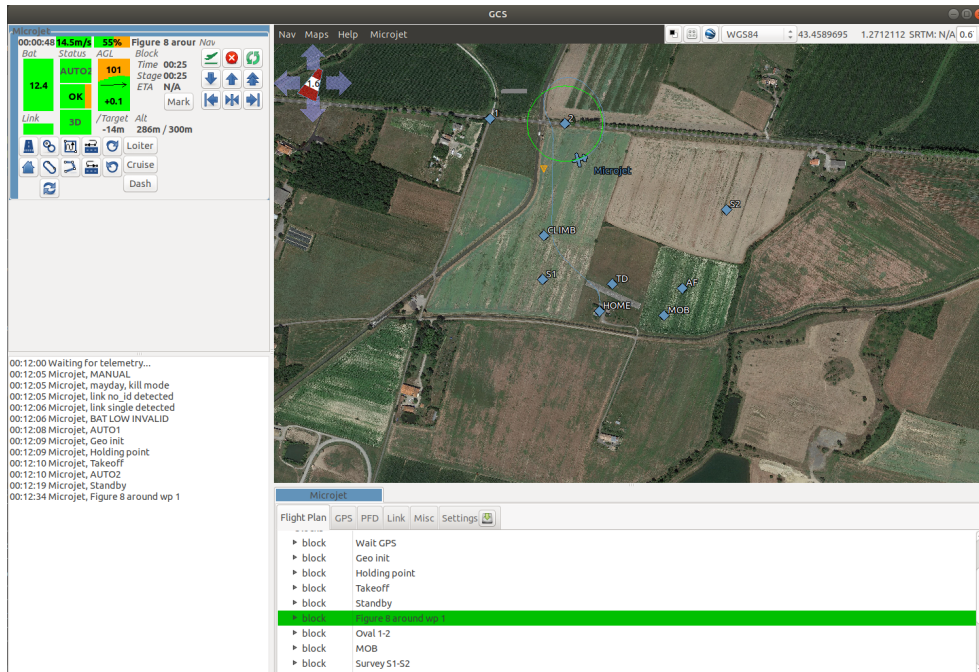


Figure 5.4: A snapshot of ground control station software (GCS), visualizing the map, waypoints (blue diamonds), the aircraft, its flight history (blue), the intended flight path (green), as well as its state (bottom in green) and some telemetry data (top left). The plan that it is following is one of the automatically generated test scenarios.

4. If `--gcs` argument is set, the ground control station window will be opened (See Figure 5.4). It provides a real-time map that visualizes the aircraft path, waypoint locations, wind direction and velocity, and several other parameters of the aircraft such as their airspeed and altitude.
5. `--no-sim` argument tells tester to not launch the simulator. Its use case can be when a physical autopilot is being used (similar to MicroPilot's HWIL mode), or when for any reason the user chooses to run the simulator manually.

Test runner can move waypoints as well. The user can fix any number of waypoints at specified locations, providing their latitude longitude and altitudes.

It can also randomize their location inside a cube. Boundaries of that cube (i.e. east-west, north-south, and floor-ceiling) are customizable through provided command line arguments.

The comprehensive list of command line arguments is included in Table 4 in [Appendix A](#).

5.5.4.3 Automated Test Generator

The test generator component is a fuzz test generator specifically designed for Paparazzi. An autopilot has plenty of parameters to change. Many of them need to remain unchanged. For example, changing aerodynamic and physical parameters of an air frame will make it behave incorrectly or even crash. A naïve algorithm might be tempted to only change these parameters to optimize a metric of “bug”s per test. To comply with the constraints in input format and range and optimize test scenario diversity without having to have thousands of tests, I opted for developing a specialized automated test generator.

This test generator is completely compatible with Paparazzi and designed with specific needs of an autopilot in mind. It begins with loading and parsing the one fixed wing flight plan that comes with Paparazzi. It defines multiple actions (control blocks) available to a fixed wing aircraft. Two important components in the flight plan that are used are the waypoint names and locations, and available blocks. A test scenario is generated by taking the initial flight plan and fuzzing three parameters about it:

1. Waypoint locations
2. Sequence of actions
3. Action timing

The test generator can be configured to fuzz all or some of them. I generated 31 tests with fuzzing the last two. Please note that the sequence of actions can have any length between 1 and the number of possible actions (5, in my case). Also for each sequence, $n!$ different permutations can be made that will result in very different outcomes. That makes the number of tests, $5! \times 1 + 4! \times 5 + \dots + 1! \times 5 = 325$.

Also note that these generated tests are made of unique blocks, in other words, a test scenario does not contain plans like “Do A then B then A again” while these plans are perfectly valid. In case one wants to have such plans they can write test plans manually.

The output is a python module that contains a subclass of `PlanBase`, as expected by the automated test executor. These two work together seamlessly and the user can just use them in a plug-and-play manner without touching the code. Though, the generated code is designed to be effortlessly understandable and easily editable it even incorporates some auto-generated comments in the code.

The full table of command line arguments that test generator accepts is presented in Table 6 in [Appendix A](#). To provide an example, the following command generates tests of length (number of actions) 2 for ‘Microjet’ aircraft and writes it to the file `l2.py` in the specified directory. It fuzzes the location (latitude, longitude, and altitude) of waypoint ‘S2’ in the default cube, just overriding the altitude bounds in 200 to 220 meters, while the coordinates of ‘S1’ are fixed. The initialization and landing stages which must be excluded from the test scenario are also specified.

```
gen_test.py --exclude "Wait_GPS" "Geo_init" "Holding
point" "Takeoff" "Standby" "Land_Right_AF-TD" "Land_Left
AF-TD" "land" "final" "flare" --fuzz-wps S2 \
--wp-fuzz-bounds-alt 200 220 -w S1 43.4659053 1.27 300 \
--length 2 Microjet pprz_tester/generated_plans/l2.py
```

Some of the available actions and all the waypoints in this flight plan can be seen in [Figure 5.4](#).

5.6 Results

In this section, I present the results of the experiments and answer the two research questions.

Table 5.2: Change point detection precision, recall, and F1-score calculated for the baseline methods using three values of tolerance (τ) for multiple configurations.

Cost Function	Search Method	Penalty	$\tau = 1s$			$\tau = 3s$			$\tau = 5s$		
			Prec.	Recall	F1	Prec.	Recall	F1	Prec.	Recall	F1
Autoregressive Model	Bottom Up	1000	10.43%	75.44%	18.33%	21.21%	80.32%	33.55%	28.94%	81.22%	42.68%
	Window Based	100	2.94%	3.98%	3.38%	8.53%	11.41%	9.76%	12.89%	17.54%	14.86%
Least Absolute Deviation	Bottom Up	500	7.32%	52.54%	12.85%	17.52%	87.73%	29.20%	25.02%	88.95%	39.05%
	Window Based	500	5.24%	8.31%	6.42%	15.20%	24.03%	18.62%	21.79%	38.25%	27.76%
Least Squared Deviation	Bottom Up	1000	7.44%	85.09%	13.68%	16.40%	89.81%	27.74%	24.16%	90.47%	38.14%
	Window Based	500	3.59%	6.79%	4.70%	10.27%	16.51%	12.66%	16.18%	26.84%	20.19%
Linear Model Change	Bottom Up	100	37.59%	28.98%	32.73%	45.20%	38.39%	41.52%	48.07%	41.36%	44.46%
	Window Based	500	6.70%	4.14%	5.12%	20.50%	13.05%	15.95%	38.78%	26.77%	31.67%
Gaussian Process Change	Bottom Up	100	3.77%	92.23%	7.25%	8.99%	92.23%	16.39%	13.53%	92.23%	23.60%
	Window Based	100	2.94%	3.95%	3.37%	8.69%	11.50%	9.90%	13.64%	18.30%	15.63%
Rank-based Cost Function	Bottom Up	100	13.45%	60.19%	21.98%	19.49%	80.10%	31.35%	22.98%	87.23%	36.38%
	Window Based	100	8.10%	13.70%	10.18%	15.72%	30.73%	20.80%	21.38%	46.64%	29.32%
Kernelized Mean Change	Bottom Up	100	4.13%	3.24%	3.63%	12.22%	8.14%	9.77%	15.38%	10.58%	12.54%
	Window Based	100	2.82%	3.00%	2.91%	10.14%	8.40%	9.19%	13.64%	12.61%	13.10%

Table 5.3: Change point detection precision, recall, and F1-score calculated, on the test data, for the proposed model, using three values of tolerance (τ) compared with the respective τ 's best F1 score among baseline methods

τ	Prec.	Recall	F1 score	Baseline F1
1s	56.77%	79.32%	66.18%	32.73%
3s	69.58%	88.88%	78.06%	41.52%
5s	79.82%	91.87%	85.42%	44.46%

5.6.1 RQ1 Results: CPD Performance

Table 5.2 shows the results of running CPD algorithms for various configurations (as described in 5.3.1). For each search method and cost function pair only one of the penalty values which resulted in the highest F1 scores for all τ values is reported.

The first observation from the results is that as values of τ increases the scores get better. This was expected, since larger values relax the constraints on which detected change points are considered as a true positive. Another observation is that the bottom-up segmentation consistently outperforms the window-based segmentation method. I can also see that the linear cost function beats all the other ones, in terms of precision. The Gaussian cost function achieves much higher recall values costing it a huge loss in precision. It means this cost function results in detecting numerous change points spread across the time axis, so there is a good chance of having at least one change point predicted close to each true change point (hence the high recall), but also there are a lot of false positives, which leads to a low precision.

Measuring the same metrics on how the proposed model performs on the test data shows better scores, almost twice the F1 score of the best performing baseline (see Table 5.3). Please note that unlike machine learning algorithms (such as mine), CPD algorithms do not have a separate training and testing phases. This fact works in their favor (by using the entire dataset for prediction and not just the training set), but still my model outperforms them.

In terms of execution cost, running all 42 different settings of CPD algorithms on the whole dataset took a bit over 12 hours in the cloud using 16 CPUs and

64GB of main memory. The deep learning model on the other hand takes about an hour to train (which only needs to be done once), on a smaller machine (see section 5.4.5). It made predictions on the whole dataset in less than a minute. So to answer RQ1, my method has shown $(66.18/32.73)-1 = 102.20\%$ improvement in F1 score with $\tau = 1s$, $(78.06/41.52)-1 = 88.00\%$ with $\tau = 3s$, and $(85.42/44.46)-1 = 92.13\%$ with $\tau = 5s$; almost doubling the score compared to the baselines.

The proposed model, which requires less memory compared to traditional CPD algorithms, improved their best performance by up to 102%, measured by F1 score, in less execution time.

5.6.2 RQ2 Results: Multi-class Classification Performance

To answer RQ2, I first compare different configurations of the baseline methods using the F1 score (harmonic mean of precision and recall) on the test data. The results are presented in Table 5.4.

Comparing the baseline methods with my approach (the last row) in Table 5.4 shows that my model outperforms all baselines. Comparing it with the model with the best F1-score shows a $(86.29/73.21)-1 = 17.87\%$ improvement in precision as well as a $(95.04/82.16)-1 = 15.68\%$ improvement in recall that means $(90.45/77.42)-1 = 16.83\%$ overall improvement in F1-score.

To have a feeling of how good my predictions are in practice, Figure 5.5 shows the output of my model side by side with the ground truth. The horizontal axis shows sample ID (time) and the states are color coded. As it is seen, this algorithm performs better when the state changes are farther apart. Also there are some state changes that happen quite briefly which are not detected. That is not to a great surprise since it takes some time for state changes to be reflected in the outputs and those might not have got any chance.

The classical models only see one window of the data at a time, convolutional layers on the other hand are more generalized and flexible since each filter in each layer is comparable to a sliding window. As we saw in Table 5.4, a larger window size means a higher performance. However, it gets significantly more difficult to train a model with large window sizes. In addition, convolutions can

Table 5.4: Precision, recall, and F1 score of ridge classifiers (linear classifiers with L2 regularization) and decision tree classifiers (DT) with different sliding window widths (w). For each algorithm on each w several hyper-parameters were applied producing 152 different models. In this table, I only show the results of the best performing model in each group.

w	Classifier	Max Depth	Max Features	Prec.	Recall	F1
3	Ridge	-	-	71.39%	20.73%	32.13%
3	DT	-	-	69.21%	82.36%	75.21%
5	Ridge	-	-	69.15%	21.89%	33.26%
5	DT	100	-	68.37%	83.16%	75.04%
10	Ridge	-	-	71.97%	24.02%	36.02%
10	DT	260	-	67.94%	79.14%	73.12%
15	Ridge	-	-	76.87	25.90%	38.75%
15	DT	-	$\sqrt{10w}$	69.06%	80.76%	74.45%
20	Ridge	-	-	80.38%	26.50%	39.86%
20	DT	175	$\sqrt{10w}$	73.21%	82.16%	77.42%
My Approach				86.29%	95.04%	90.45%

automatically learn preprocessing steps that could be beneficial such as a moving average. Each convolutional filter can learn a linear combination of its inputs. So when the convolutional layers are stacked on each other, with non-linear activation functions in between, the hypothesis space they can learn becomes quite large, probably much larger than most of the classical ML algorithms here. Also, they are still quite efficient (more efficient than baselines) due to parameter sharing and their high parallelizability.

The fact that the performance improves as the window size increases indicates the positive effect of being able to see longer-term relations in detecting the system's state. Recurrent cells (such as GRU) can capture long-term dependencies (that do not necessarily fall into one window) and learn sequences. This is one of the major differences between an RNN model and others, such as decision trees,

which do not have such a notion of a “long-term memory” as LSTM/GRU neural networks do. All a decision tree could see is the values in a sliding window.

In terms of the training complexity (time and memory), this method is superior as well. That can largely be attributed to the use of deep learning. In baseline models, as the window size w grows the training and evaluation complexity grows, up to a point that they ran out of memory – consuming all the 47GB of main memory and swap area. This forced us to train them in the cloud. Meanwhile, as mentioned earlier, the deep learning model could be trained on a 8GB GPU in roughly an hour. (see section 5.4.5 for the machines’ specs). Also, the decision tree training was not parallelized using only one core of the CPU, while virtually all deep learning models can be heavily parallelized on a GPU/TPU.

The proposed model, which requires less than half as many CPUs and 70% as much memory compared to the best performing classical ML model, improved their best performance by up to 17%, measured by F1 score, in less execution time.

5.6.3 RQ3 Results: CPD Performance on Paparazzi (replication)

The results of applying baseline algorithms on Paparazzi dataset can be seen in table 5.5. The range of CPD performance is quite varied across the techniques, for example there are scores less than 1% (e.g. 0.39% recall score of linear models) as well as some 100% scores (e.g. recall scores for Gaussian models with bottom-up search method). Please note that the high variation in performance scores is not limited to recall metric. The 100% recalls are accompanied with a low precision; achieving that is not difficult. A method that outputs every point as a change-point will get similar results. The setting that got most of the best numbers is Pelt algorithm using an L1 cost function and a high penalty coefficient of 1,000. However, please note that Pelt is a quite slow algorithm that could easily become infeasible to run on larger dataset, as it was the case for the first two RQs. As a matter of fact, running Pelt took more than 14 hours on the same machine that performed all other CPD algorithms in less than 1 hour. What you see in table 5.5 is the summary of the results of more than 23,800 experiments.

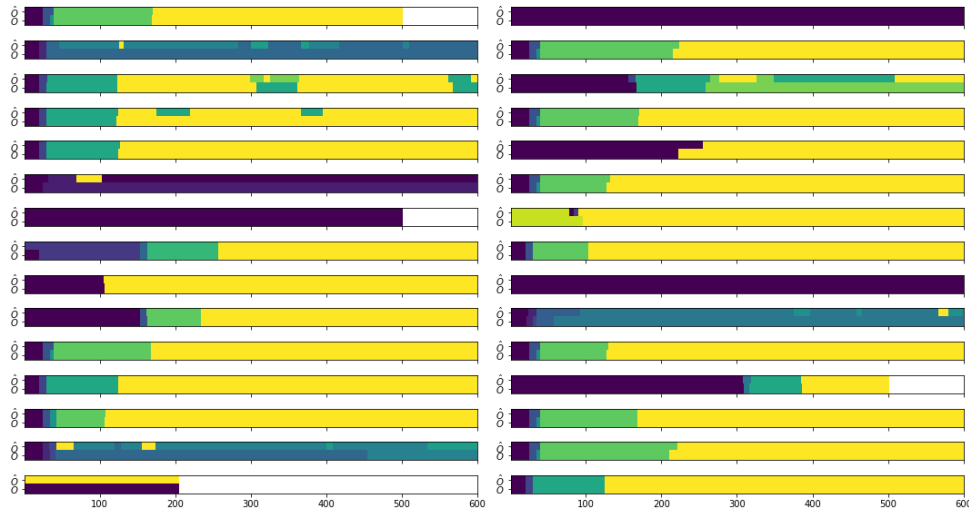


Figure 5.5: Evaluation of the model on 30 random test data. Each graph shows the states in one run of the system. The colors show the states. The top-half of each plot depicts model’s prediction of the system states (\hat{O}) and the bottom-half shows the true labels (O). Since the output is one-hot encoded, the item with the most probability is used as the predicted label at each point in time. X-axis is the time axis. Only the first 600 samples (2 minute of simulation) are shown to improve legibility.

Comparing the baseline results with the neural network model’s results in the last three rows of the table, you can see improvements of 47.88%, 34.81%, and 18.30% in F1 scores compared to the best numbers in the baselines (for $\tau = 1, 3, 5$ seconds respectively). Although this model still performed better than the baselines, the improvement margin was higher in RQ1. (see the RQ1 results summary box and the paragraph above that). However, you can see that the scores that this approach achieved are not low, but the baselines could do a better job on Paparazzi’s data compared to MicroPilot’s, therefore shrinking the margin of improvement. It can be due to the fact that the new dataset is tiny compared to MP, classic machine learning algorithms can do better so a deep learning model cannot shine here as it could for MP. As a reminder: Paparazzi dataset consists of 300 tests, each containing up to 2,500 samples vs. 900 tests in MP with lengths up to 18,000 samples. It is worth mentioning that both in the first two RQs and here,

the variation in baselines performances was way higher while my deep learning approach shows a more consistent performance; check the second column (recall with $\tau = 1s$) for example.

The proposed approach was applied on another similar software system with a smaller dataset size and showed a near 48% improvement over the baselines (with a 94% F1 score), confirming that it is a feasible approach even with smaller dataset size. The lower margin of improvement compared to RQ1 is attributed to the baselines performing better on this data, rather than my approach not performing good enough.

5.6.4 RQ4 Results: Multi-class Classification Performance on Paparazzi (replication)

In table 5.6 the comparative results of the aforementioned algorithms with my method is presented. To compare with the MP's case, we can see that in all but two settings limiting the maximum number of features did not help improve the model. Another similar observation is that the scores do not vary much with changing the window size, and decision trees almost universally outperform linear classifiers. The scores themselves are just around the same values as well: Ridge classifier F1 score here is in 21-29% range, which was 32-39% in RQ2. For decision trees it is in 67-68% range here while it is in 73-77% range in RQ2.

The deep learning model's performance measures (on validation data) can be found in the last row of the table. You can see an $(82.23/68.96) - 1 = 19.24\%$ improvement in F1 score, over the baselines. It confirms that this method can beat the best performing baselines in classifying input samples into the system's internal states. The improvement in RQ2 is 16.83%, not very different from this. This, again provides some evidence on the generalizability of the results in the first two RQs.

Figure 5.6 summarizes the prediction results of the deep learning model compared to the ground truth. You can get a sense of how good does an '82.23%' F1 score look by examining that chart. To measure this quality it in a more scientific

Cost Function	Search Method	Penalty	$\tau = 1s$			$\tau = 3s$			$\tau = 5s$		
			Prec.	Recall	F1	Prec.	Recall	F1	Prec.	Recall	F1
Autoregressive Model	Bottom Up	500	13.65%	14.13%	18.89%	39.73%	36.40%	36.14%	57.11%	53.20%	50.04%
	Pelt	500	13.56%	13.85%	19.03%	39.67%	36.17%	36.03%	56.97%	52.89%	49.83%
	Window Based	100	15.97%	7.59%	13.54%	45.47%	20.40%	29.56%	64.16%	30.76%	41.20%
Least Absolute Deviation	Window Based	100	24.50%	15.92%	19.85%	56.39%	37.24%	44.53%	68.94%	48.20%	56.23%
	Pelt	1000	26.74%	34.08%	29.92%	60.77%	77.32%	67.74%	74.73%	93.16%	82.61%
	Bottom Up	1000	26.66%	34.99%	30.35%	60.11%	78.14%	67.61%	72.85%	92.94%	81.30%
Least Squared Deviation	Window Based	1000	25.01%	16.14%	20.26%	54.58%	35.77%	42.96%	68.73%	48.11%	56.18%
	Pelt	1000	21.28%	80.04%	33.49%	51.05%	99.58%	67.20%	65.36%	99.97%	78.76%
	Bottom Up	1000	21.30%	81.24%	33.62%	50.98%	99.50%	67.12%	65.24%	99.97%	78.66%
Linear Model Change	Bottom Up	100	7.39%	0.39%	9.98%	27.44%	1.49%	10.27%	52.51%	2.75%	9.90%
	Window Based	100	7.39%	0.39%	9.98%	27.44%	1.49%	10.27%	52.51%	2.75%	9.90%
	Pelt	100	7.39%	0.39%	9.98%	27.44%	1.49%	10.27%	52.51%	2.75%	9.90%
Gaussian Process Change	Window Based	100	7.39%	0.39%	9.98%	27.44%	1.49%	10.27%	52.51%	2.75%	9.90%
	Pelt	100	7.39%	0.39%	9.98%	27.44%	1.49%	10.27%	52.51%	2.75%	9.90%
	Bottom Up	100	12.42%	100.00%	22.03%	33.15%	100.00%	49.55%	49.04%	100.00%	65.47%
Rank-based Cost Function	Pelt	100	21.11%	31.85%	25.51%	52.29%	74.63%	61.13%	65.88%	89.30%	75.46%
	Bottom Up	100	20.55%	31.84%	25.17%	49.97%	73.22%	59.04%	64.57%	89.35%	74.61%
Kernelized Mean Change	Bottom Up	100	15.66%	1.90%	9.45%	44.23%	5.36%	12.95%	62.37%	7.48%	15.66%
	Pelt	100	14.07%	1.64%	9.77%	42.06%	4.74%	12.59%	60.92%	6.67%	14.56%
	Window Based	100	8.31%	0.61%	9.97%	29.38%	2.02%	10.60%	53.34%	3.39%	10.79%
My Approach	Dataset:	Validation	44.70%	52.38%	48.24%	84.56%	90.54%	87.45%	90.53%	96.86%	93.59%
		Test (Tuning)	42.56%	48.26%	45.23%	87.11%	89.68%	88.37%	92.54%	96.48%	94.47%
		Training	56.43%	62.53%	59.32%	89.64%	92.79%	91.19%	93.97%	97.93%	95.91%

Table 5.5: Change point detection methods performance on Paparazzi dataset. Since the 100% recalls are actually outliers, the next largest recall values are in bold face as well.

Table 5.6: Precision, recall, and F1 score of ridge classifiers (linear classifiers with L2 regularization) and decision tree classifiers with different sliding window widths (w). In this table, I only show the results of the best performing model in each group.

w	Classifier	Max Features	Precision	Recall	F1
3	Ridge	-	44.82%	14.45%	21.85%
3	Decision Tree	$\sqrt{10w}$	61.88%	73.56%	67.22%
5	Ridge	-	46.62%	15.10%	22.81%
5	Decision Tree	-	64.28%	73.00%	68.36%
10	Ridge	-	47.59%	16.29%	24.27%
10	Decision Tree	-	65.06%	73.36%	68.96%
15	Ridge	-	44.41%	17.33%	24.93%
15	Decision Tree	-	63.53%	74.68%	68.65%
20	Ridge	-	57.97%	19.10%	28.74%
20	Decision Tree	-	64.72%	73.36%	68.77%
25	Ridge	-	62.13%	19.66%	29.87%
25	Decision Tree	$\sqrt{10w}$	61.17%	75.18%	67.45%
Proposed Method			77.20%	87.97%	82.23%

way, it is possible to use this state model in a downstream task and see how that task is performing.

Compared to the baselines, the proposed approach could detect the internal state of system with a 77% precision and 88% recall rate, showing a 19% improvement. Similar results were seen in RQ 2 (which is closely related to this question) confirming generalizability of the findings.

5.6.5 RQ5 Results: Hyper-parameter Tuning

I used Tensor Board to visualize and compare the effects of different hyper-parameters on the model’s performance. First, for a sanity check, I visualized 6

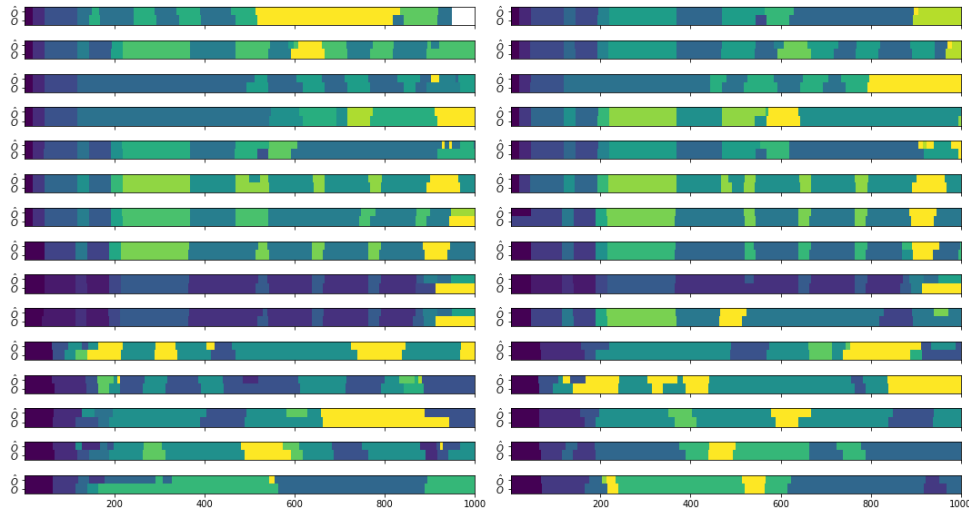


Figure 5.6: The states are color coded and each rectangle shows one test, only the first 1000 samples (200 seconds) are shown for more clarity. The same as figure 5.5 in, the top half of each rectangle shows the model's output and the bottom half shows the true labels.

metrics, precision and recall in detecting change points with a the smallest and largest tolerances of $\tau = 1s$ and $\tau = 5s$, and precision and recall in estimating the internal state in Figure 5.7. A healthy linear and positive correlation is visible between virtually all pairs of these metrics (all except classification precision), it means that trying to improve one will not come at the expense of others.

Filtering the data to find the commonalities of better hyper-parameters shows that the number of GRU cells has a high correlation with both CPD precision and classification precision. First and foremost the learning rate is has the largest correlation (absolute) with the performance metrics. The best configurations all share the lowest learning rate. Filtering the other configs out, the next most important factor was the convolutional layer counts. The configurations that did not include the last and largest convolutional layer with kernel size of 20, performed much better than the others. It makes sense because training larger kernels gets harder and harder and requires more data. Also, based on the previous studies, especially in the field of computer vision, we can say that each filter

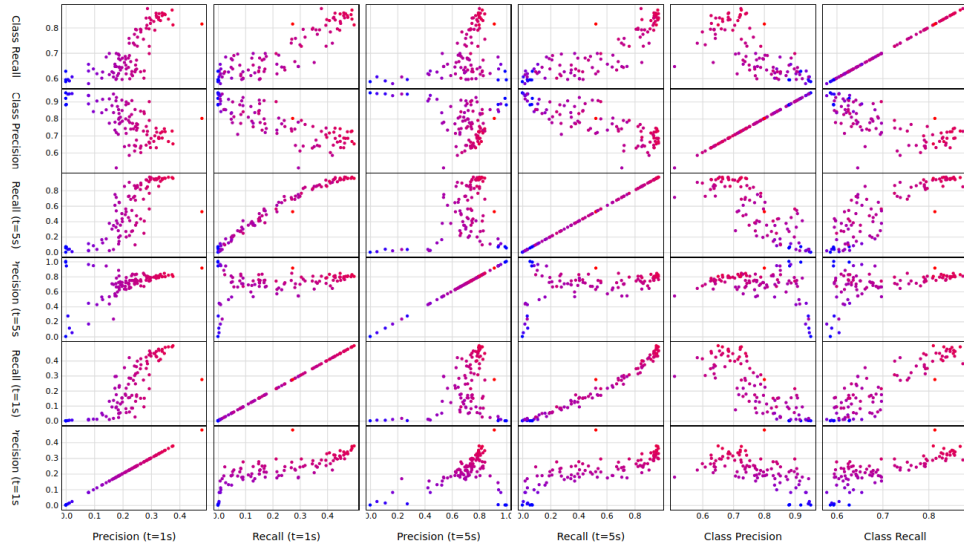


Figure 5.7: Matrix of scatter plots comparing precision and recall of the deep learning model on test data.

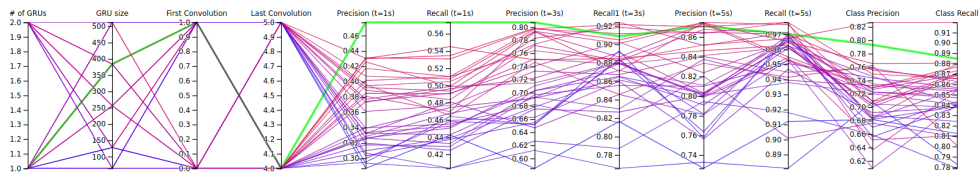


Figure 5.8: Parallel coordinates view showing the effects of selected hyper-parameters on the performance measures.

learns one feature so it is more effective to have small filters that are easier to train and can learn multiple features of the data. Therefore I can reduce the size of convolutional section of the model and use more recurrent cells to boost the performance without adding too many parameters.

After filtering out some of the worst configurations (such as the ones with lower than 40% Recall (t=1s)), the better ones are shown in figure 5.8. This chart shows 3 + 8 parallel axes, 3 hyper-parameters and 8 metrics. The green highlighted line is the best configuration tried, with the highest area under the curve. That configuration has the best or close to the best performance in all metrics. The

Evaluation Metric	Default Hyper-params	Tuned Hyper-params
Precision ($\tau = 1s$)	28.90%	44.70%
Recall ($\tau = 1s$)	37.44%	52.38%
F1 ($\tau = 1s$)	32.62%	48.24%
Precision ($\tau = 3s$)	59.34%	84.56%
Recall ($\tau = 3s$)	71.52%	90.54%
F1 ($\tau = 3s$)	64.87%	87.45%
Precision ($\tau = 5s$)	73.85%	90.53%
Recall ($\tau = 5s$)	85.18%	96.86%
F1 ($\tau = 5s$)	79.11%	93.59%
State Detection Precision	57.35%	77.20%
State Detection Recall	78.90%	87.97%
State Detection F1	66.41%	82.23%

Table 5.7: The performance results of the model trained on paparazzi before hyper parameter tuning (i.e. using the hyper-parameters that worked best for MP on Paparazzi as well) vs. after fine-tuning hyper-params on Paparazzi’s dataset.

configuration is to use 384 GRU cells in 1 layer, 16 filter per convolutional layer sizes from 5 to 15, and a slower learning rate of 3×10^{-4} . I used this set of hyper-parameters to train the best model. The stopping criteria on training the model was for the validation loss to plateau, with a ‘patience’ of 10 epochs. The optimizer converged in 54 epochs, after less than 3 minutes of training. You can see the performance scores of the tuned model in tables 5.6 and 5.5.

The sensitivity of this approach to the choice of hyper parameters can be seen clearly here. I first started with the hyper-parameters that worked best for MP and the values close to them. With that set of hyper-parameters the performance was even worse than the baselines in some cases; see table 5.7. In the second iteration, after removing the worst performing configurations and defining a search space, that set of parameters was discarded. Even though the search space was not very large, the range of This is to emphasize the effect and importance of fine-tuning hyper-params.

The result of this tuning was a model with fewer convolutional layers and smaller kernel sizes but with more convolutional filters and recurrent layers that could perform better than the baselines. This shows having more and smaller convolutional filters is more effective in capturing local features in the data compared to having a small number of large filters (with kernels of size 20 for example which has 200 parameters). It also shows the important role that recurrent layers play in discovering long-term relations between inputs and outputs.

5.7 Limitations and Threats to Validity

One of the limitations of this approach is that it might miss an input-output invariant correlation. It can happen when the input remains constant or it changes too little to reveal its relation with certain outputs. We assume that during the data collection, sampling happens in regular intervals; this approach probably will have a hard time achieving high performances, working on unevenly spaced time-series data.

Although labeling the input/output series for debugging, comprehension, ... purposes is a common task that developers do in the industry and in open source settings, performing the labeling on a large dataset can become arduous; thus making this approach less practical in some settings. However, as will be stated later, there are potential remedies to reduce the amount of required data to train a model. Also, it is worth mentioning that training this model needs to happen once. The labeling task can be looked at as an investment; after the model is trained, it can be used in the future to automatically label the data without any need to more user input.

In terms of construct validity, I am using standard metrics to evaluate the results. However, the use of tolerance margin should be taken with caution since it is a domain-dependant variable and can change the final results. To alleviate this threats, I have used multiple margins and reported all results. In terms of internal validity threats, I reduced the threat by not implementing the CPD baselines

by myself and rather reusing existing libraries. In terms of conclusion validity threats, I have used many (888) real test cases from MicroPilot's test repository and provided a proper train-validation-test split for training, tuning, and evaluation. Finally, in terms of external validity threats, this study suffers from being limited to only two case studies. However, (a) the studies are large-scale real-world systems with many test cases, and (b) they are both from industry and open source to be more representative. In the future one potential extension is to extend this work with more case studies from other domains, to increase its generalizability. Another external validity threat could be in the random split of training and validation tests. There is no straightforward way to make sure the validation dataset is 100% different from the training data; however the split was performed randomly as it is common in Machine Learning and there are no repetitive test cases in the dataset, so the chances are high that the validation dataset is different enough from the training data.

Chapter 6

Conclusion and Future Work

In this thesis, I developed a novel method for inferring black box models for autopilot software systems. My method at its core is a deep neural network that combines convolutions and recurrent cells: hybrid CNN-RNN model. This design is inspired by deep neural network architectures that showed good performance in other fields (such as speech recognition, sleep phase detection, and human activity recognition). It can be used for both CPD and state classification problems in multivariate time series.

This method be used as a black-box state model inference for variety of use cases such as testing, debugging, and anomaly detection in control software systems, where there are several input signals that control output states.

I have trained and evaluated this neural network on two case studies of a UAV autopilot softwares, one from the FOSS¹ community and one from our industry partner. It showed promising results in inferring a behavioural model from the autopilot execution data; showing significant improvement in both change point detection and state classification as compared with several baselines on 10 comparison metrics.

Some potential extensions to this work include: (a) Examining if and how transfer learning can be employed to make this method work with smaller datasets and also reduce the labeling overhead. (b) Adapting this method to work with other data types such as images in video surveillance systems, or sensor data +

¹Free (as in freedom) and Open-Source software.

6. Conclusion and Future Work

stream of images in self-driving cars, could be another interesting line of work for continuing this work (c) Using the inferred model to perform a downstream task such as test generation or validation. (d) This approach at its current form is an offline method which is more useful for a postmortem analysis. To make it beneficial in more settings, it could be expanded into being an online (or even real-time) method.

Bibliography

Modeling software with finite state machines : a practical approach. Auerbach, Boca Raton, FL, 2006. ISBN 9780849380860.

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <https://www.tensorflow.org/>. Software available from tensorflow.org.

Ryan Prescott Adams and David JC MacKay. Bayesian online changepoint detection. *stat*, 1050:19, 2007.

Ziad A Al-Sharif. *An Extensible Debugging Architecture Based on a Hybrid Debugging Framework*. PhD thesis, University of Idaho, 2009.

Samaneh Aminikhanghahi and Diane J Cook. A survey of methods for time series change point detection. *Knowledge and information systems*, 51(2):339–367, 2017.

Daniele Angelosante and Georgios B Giannakis. Group lassoing change-points in piecewise-constant ar processes. *EURASIP Journal on Advances in Signal Processing*, 2012(1):70, 2012.

- Jushan Bai. Estimation of a change point in multiple regression models. *Review of Economics and Statistics*, 79(4):551–563, 1997.
- Jushan Bai, Robin L Lumsdaine, and James H Stock. Testing for and dating common breaks in multivariate time series. *The Review of Economic Studies*, 65(3): 395–432, 1998.
- Daniel Barry and John A Hartigan. A bayesian analysis for change point problems. *Journal of the American Statistical Association*, 88(421):309–319, 1993.
- Michèle Basseville, Igor V Nikiforov, et al. *Detection of abrupt changes: theory and application*, volume 104. prentice Hall Englewood Cliffs, 1993.
- Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277, 2011.
- Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, Arvind Krishnamurthy, and Thomas E Anderson. Mining temporal invariants from partially ordered logs. *Operating systems review*, 45(3):39–46, 2012. ISSN 0163-5980.
- Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 468–479, 2014.
- Alan W Biermann and Jerome A Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE transactions on Computers*, 100(6):592–597, 1972.
- Duncan A.J. Blythe, Paul Von Bunau, Frank C. Meinecke, and Klaus Robert Muller. Feature extraction for change-point detection using stationary subspace analysis. *IEEE Transactions on Neural Networks and Learning Systems*, 23(4):631–643, apr 2012. ISSN 2162237X. URL <http://ieeexplore.ieee.org/document/6151166/>.

- Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent Neural Networks for Multivariate Time Series with Missing Values. *Scientific Reports*, 8(1):1–12, dec 2018. ISSN 20452322.
- Jie Chen and Arjun K Gupta. *Parametric statistical change point analysis: with applications to genetics, medicine, and finance*. Springer Science & Business Media, 2011.
- Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- Md Foezur Rahman Chowdhury, S-A Selouani, and D O’Shaughnessy. Bayesian on-line spectral change point detection: a soft computing approach for on-line asr. *International Journal of Speech Technology*, 15(1):5–23, 2012.
- Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24, 2006.
- Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering*, 38(2):243–257, 2011.
- Christophe Damas, Bernard Lambeau, Pierre Dupont, and Axel Van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.
- Frédéric Desobry, Manuel Davy, and Christian Doncarli. An online kernel change detection algorithm. *IEEE Transactions on Signal Processing*, 53(8):2961–2974, 2005.
- Zahra Ebrahimzadeh, Min Zheng, Selcuk Karakas, and Samantha Kleinberg. Deep Learning for Multi-Scale Changepoint Detection in Multivariate Time Series. Technical report, 2019.

- Stephen H Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97–111, 2001.
- Chandra Erdman and John W Emerson. A fast bayesian change point analysis for the segmentation of microarray data. *Bioinformatics*, 24(19):2143–2148, 2008.
- Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.
- Hany Farid and Eero P Simoncelli. Differentiation of Discrete Multidimensional Signals. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 13(4):496–508, 2004.
- Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.
- Piotr Fryzlewicz et al. Wild binary segmentation for multiple change-point detection. *The Annals of Statistics*, 42(6):2243–2281, 2014.
- Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
- Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05*, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930566. doi: 10.1145/1065010.1065036. URL <https://doi.org/10.1145/1065010.1065036>.
- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- Roland Groz, Adenilso Simao, Nicolas Bremond, and Catherine Oriat. Revisiting ai and testing methods to infer fsm models of black-box systems. In *2018*

-
- IEEE/ACM 13th International Workshop on Automation of Software Test (AST)*, pages 16–19. IEEE, 2018.
- Abeer Hasan, Wei Ning, and Arjun K Gupta. An information-based approach to the change-point problem of the noncentral skew t distribution with applications to stock market data. *Sequential Analysis*, 33(4):458–474, 2014.
- Gautier Hattenberger, Murat Bronz, and Michel Gorraz. Using the paparazzi uav system for scientific research. 2014.
- Shohei Hido, Tsuyoshi Idé, Hisashi Kashima, Harunobu Kubo, and Hirofumi Matsuzawa. Unsupervised change analysis using supervised learning. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 148–159. Springer, 2008.
- Falk Howar, Bernhard Steffen, Bengt Jonsson, and Sofia Cassel. Inferring canonical register automata. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 251–266. Springer, 2012.
- DA Hsu. A bayesian robust detection of shift in the risk structure of stock market returns. *Journal of the American Statistical Association*, 77(377):29–39, 1982.
- Rosziati Ibrahim, Mohd Zainura Saringat, Noraini Ibrahim, and Noraida Ismail. An automatic tool for generating test cases from the system’s requirements. In *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*, pages 861–866. IEEE, 2007.
- Tsuyoshi Idé and Koji Tsuda. Change-point detection using krylov subspace learning. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 515–520. SIAM, 2007.
- Anthony R. Ives and Vasilis Dakos. Detecting dynamical changes in nonlinear time series using locally linear state-space models. *Ecosphere*, 3(6):art58, jun 2012. ISSN 2150-8925. URL <http://doi.wiley.com/10.1890/ES11-00347.1>.
- Eamonn Keogh, Selina Chu, David Hart, and Michael Pazzani. An online algorithm for segmenting time series. In *Proceedings 2001 IEEE international conference on data mining*, pages 289–296. IEEE, 2001.

- Hossein Keshavarz, Clayton Scott, and XuanLong Nguyen. Optimal change point detection in gaussian processes. *Journal of Statistical Planning and Inference*, 193:151–178, 2018.
- Haidar Khan. *Predicting Change Points in Multivariate Time Series Data*. PhD thesis, Rensselaer Polytechnic Institute, 2019.
- Haidar Khan, Lara Marcuse, and Bülent Yener. Deep density ratio estimation for change point detection. 2019.
- Rebecca Killick, Paul Fearnhead, and Idris A Eckley. Optimal detection of change-points with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.
- Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 178–189, 2014.
- Kevin J Lang, Barak A Pearlmutter, and Rodney A Price. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference*, pages 1–12. Springer, 1998.
- Marc Lavielle. Detection of multiple changes in a sequence of dependent variables. *Stochastic Processes and their Applications*, 83(1):79–102, sep 1999. ISSN 0304-4149. URL <https://www.sciencedirect.com/science/article/pii/S030441499900023X>.
- Marc Lavielle. Using penalized contrasts for the change-point problem. *Signal processing*, 85(8):1501–1510, 2005.
- Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

- Wei-Han Lee and Ruby B Lee. Implicit smartphone user authentication with sensors and contextual machine learning. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 297–308. IEEE, 2017.
- Wei-Han Lee, Jorge Ortiz, Bongjun Ko, and Ruby Lee. Time Series Segmentation through Automatic Feature Learning. Technical report, 2018.
- Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.
- Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ltl specification mining (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–92. IEEE, 2015.
- Daniel Lemire. A better alternative to piecewise linear time series segmentation. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, pages 545–550. SIAM, 2007.
- Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1044–1051, 2019.
- David Lo, Shahar Maoz, and Siau-Cheng Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, page 465–468, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938824. doi: 10.1145/1321631.1321710. URL <https://doi.org/10.1145/1321631.1321710>.
- David Lo, Siau-Cheng Khoo, Jiawei Han, and Chao Liu. *Mining software specifications: methodologies and applications*. CRC Press, 2011.

- Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering*, pages 501–510, 2008.
- Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.
- Mohammad Jafar Mashhadi and Hadi Hemmati. An empirical study on practicality of specification mining algorithms on a real-world application. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 65–69. IEEE, 2019.
- Mohammad Jafar Mashhadi, Taha R Siddiqui, Hadi Hemmati, and Howard Loewen. Interactive semi-automated specification mining for debugging: An experience report. *arXiv preprint arXiv:1905.02245*, 2019.
- Tyler Desmond Melmoth, Adam Jacob Toews, Daniel Serge Brouillette, and Nicholas Andrew Playle. True hardware in the loop spi emulation, September 17 2019. US Patent 10,417,360.
- Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. V-net: Fully convolutional neural networks for volumetric medical image segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)*, pages 565–571. IEEE, 2016.
- Francisco Javier Ordóñez Morales and Daniel Roggen. Deep convolutional feature transfer across mobile activity recognition domains, sensor modalities and locations. In *Proceedings of the 2016 ACM International Symposium on Wearable Computers*, pages 92–99, 2016.
- Abdulmajid Murad and Jae-Young Pyun. Deep recurrent neural networks for human activity recognition. *Sensors*, 17(11):2556, 2017.
- Kyong Joo Oh and Kyoung-jae Kim. Analyzing stock market tick data using piecewise nonlinear model. *Expert Systems with Applications*, 22(3):249–255, 2002.

- Francisco Ordóñez and Daniel Roggen. Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition. *Sensors*, 16(1):115, jan 2016. ISSN 1424-8220. URL <http://www.mdpi.com/1424-8220/16/1/115>.
- Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.
- Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE, 2007.
- Petros Papadopoulos and Neil Walkinshaw. Black-box test generation from inferred models. In *Proceedings - 4th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, RAISE 2015*, pages 19–24. IEEE, may 2015. ISBN 9781479919345. URL <http://ieeexplore.ieee.org/document/7168327/>.
- Florian Pein, Hannes Sieling, and Axel Munk. Heterogeneous change point inference. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 79(4):1207–1227, 2017.
- Mathias Perslev, Michael Jensen, Sune Darkner, Poul Jørgen Jennum, and Christian Igel. U-time: A fully convolutional network for time series segmentation applied to sleep staging. In *Advances in Neural Information Processing Systems*, pages 4417–4428, 2019.
- Rychelly Glenneson da S Ramos, Paulo Ribeiro, and José Vinícius de M Cardoso. Anomalies detection in wireless sensor networks using bayesian changepoints. In *2016 IEEE 13th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 384–385. IEEE, 2016.
- Karl J (Karl Johan) Åström. *Feedback systems : an introduction for scientists and engineers*. Princeton University Press, Princeton, 2008. ISBN 9780691135762.

- Bonnie K Ray and Ruey S Tsay. Bayesian methods for change-point detection in long-range dependent processes. *Journal of Time Series Analysis*, 23(6):687–705, 2002.
- Jaxk Reeves, Jien Chen, Xiaolan L Wang, Robert Lund, and Qi Qi Lu. A review and comparison of changepoint detection techniques for climate data. *Journal of applied meteorology and climatology*, 46(6):900–915, 2007.
- Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
- David Rosenfield, Enlu Zhou, Frank H Wilhelm, Ansgar Conrad, Walton T Roth, and Alicia E Meuret. Change point analysis for longitudinal physiological data: detection of cardio-respiratory changes preceding panic attacks. *Biological psychology*, 84(1):112–120, 2010.
- James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.
- I. Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, Jan 2012. ISSN 1937-4194. doi: 10.1109/MS.2012.13.
- Sigurd Schneider, Ivan Beschastnikh, Slava Chernyak, Michael D Ernst, and Yuriy Brun. Synoptic: Summarizing system logs with refinement. In *SLAML*, 2010.
- Andrew Jhon Scott and M Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, pages 507–512, 1974.
- Weiyi Shang, Zhen Ming Jiang, Hadi Hemmati, Bram Adams, Ahmed E Hassan, and Patrick Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 402–411. IEEE Press, 2013.
- Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

- Monit Shah Singh, Vinaychandran Pondenkandath, Bo Zhou, Paul Lukowicz, and Marcus Liwicki. Transforming sensor data to the image domain for deep learning—an application to footprint detection. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2665–2672. IEEE, 2017.
- Carole H Sudre, Wenqi Li, Tom Vercauteren, Sebastien Ourselin, and M Jorge Cardoso. Generalised dice overlap as a deep learning loss function for highly unbalanced segmentations. In *Deep learning in medical image analysis and multimodal learning for clinical decision support*, pages 240–248. Springer, 2017.
- Jun-ichi Takeuchi and Kenji Yamanishi. A unifying framework for detecting outliers and change points from time series. *IEEE transactions on Knowledge and Data Engineering*, 18(4):482–492, 2006.
- Charles Truong, Laurent Oudre, and Nicolas Vayatis. Selective review of offline change point detection methods. jan 2018. URL <http://arxiv.org/abs/1801.00718>.
- Alfonso Valdes and Keith Skinner. Adaptive, model-based monitoring for cyber attack detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 80–93. Springer, 2000.
- Michele Volpato and Jan Tretmans. Active learning of nondeterministic systems from an ioco perspective. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 220–235. Springer, 2014.
- Neil Walkinshaw. *Testing Functional Black-Box Programs Without a Specification*, pages 101–120. Springer International Publishing, Cham, 2018. ISBN 978-3-319-96562-8. doi: 10.1007/978-3-319-96562-8_4. URL https://doi.org/10.1007/978-3-319-96562-8_4.
- Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *Empirical Software Engineering*, 21(3):811–853, 2016.

- Jindong Wang, Yiqiang Chen, Shuji Hao, Xiaohui Peng, and Lisha Hu. Deep learning for sensor-based activity recognition: A survey. *Pattern Recognition Letters*, 119:3–11, 2019.
- Yao Wang, Chunguo Wu, Zhaohua Ji, Binghong Wang, and Yanchun Liang. Non-parametric change-point method for differential gene expression detection. *PloS one*, 6(5), 2011.
- Zhiguang Wang, Weizhong Yan, and Tim Oates. Time series classification from scratch with deep neural networks: A strong baseline. In *2017 International joint conference on neural networks (IJCNN)*, pages 1578–1585. IEEE, 2017.
- Yao Xie and David Siegmund. Sequential multi-sensor change-point detection. In *2013 Information Theory and Applications Workshop (ITA)*, pages 1–20. IEEE, 2013.
- Kenji Yamanishi, Jun-Ichi Takeuchi, Graham Williams, and Peter Milne. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. *Data Mining and Knowledge Discovery*, 8(3):275–300, 2004.
- Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. Deep convolutional neural networks on multichannel time series for human activity recognition. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.
- Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, and Tarek Abdelzaher. Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web, WWW '17*, page 351–360, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee. ISBN 9781450349130. doi: 10.1145/3038912.3052577. URL <https://doi.org/10.1145/3038912.3052577>.
- Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

- Ming Zeng, Le T Nguyen, Bo Yu, Ole J Mengshoel, Jiang Zhu, Pang Wu, and Joy Zhang. Convolutional neural networks for human activity recognition using mobile sensors. In *6th International Conference on Mobile Computing, Applications and Services*, pages 197–205. IEEE, 2014.
- Jia-Shu Zhang and Xian-Ci Xiao. Predicting chaotic time series using recurrent neural network. *Chinese Physics Letters*, 17(2):88, 2000.
- Yi Zheng, Qi Liu, Enhong Chen, Yong Ge, and J Leon Zhao. Exploiting multi-channels deep convolutional neural networks for multivariate time series classification. *Frontiers of Computer Science*, 10(1):96–112, 2016.

Appendix A: PprzTester documentations

UML Diagrams and tables for the developed tools: flight data recorder, automated test generator, and automated test executor.

Message Name	Sent By	Use
PPRZ_MODE	Aircraft	PprzTester listens to this message to find out when the autopilot software is booted up and is ready for flight. It also contains the state of active PID loops that is used to determine the internal state of the autopilot.
NAVIGATION	Aircraft	Contains the current active flight plan block as well as number of full circles the aircraft has completed among other data
COMMANDS	Aircraft	Contains commands from the autopilot to the servos (the outputs)
FLIGHT_PARAMS	Aircraft	Contains most of the sensor readings and required aircraft parameters
ENGINE_STATUS	Aircraft	Contains engine RPM and throttle among other parameters
CIRCLE_STATUS	Aircraft	Contains detailed information about aircraft status while it is in circling mode. Overlaps with NAVIGATION to some extent
WAYPOINT_MOVED	Server	Sent in case any waypoints are relocated
MOVE_WAYPOINT	PprzTester	To update waypoint locations
NEW_AIRCRAFT	Server	Event of a new aircraft coming online in the network. It does not necessarily mean that it is ready for flight, should listen for PPRZ_MODE as well
AIRCRAFTS_REQ	PprzTester	Request to get the list of active aircraft
AIRCRAFTS	Server	Response to the above request
CONFIG_REQ	PprzTester	Request for an aircraft's configs
CONFIG	Server	Response to the above request containing its name, flight plan, DL settings, air frame parameters, etc.
DL_SETTING	PprzTester	Update a DL setting value. It is used to launch the aircraft in the air
JUMP_TO_BLOCK	PprzTester	Commands the aircraft to change active block to another block defined in the flight plan

Table 2: Subset of all available messages in Paparazzi that are used in PprzTester tool

Argument	Description
airframe	The aircraft to simulate
plan	Plan name to run as a python module in <code>pprz_tester.generated_plans</code>
-agent-name	Specify unique agent name on ivy bus
-l, -log	Log file directory
-log-format	The format to store and compress logs in, can be <code>hd5</code> or <code>csv</code>
-p, -paparazzi-home	Directory in which Paparazzi source code is cloned in
-b, -build	Build the aircraft before launching the simulation
-gcs	Open GCS window
-no-sim	Does not launch the simulator
-prep-mode	The required conditions before starting the flight scenario. It can be waiting for altitude to stabilize (climb) or waiting for a complete circle around stand by waypoint (circle) or both.
-fuzz-wps	Waypoints to fuzz locations of. Use <code>*</code> to fuzz all
-wp-fuzz-bounds-lat	Minimum and maximum latitude (south-north) to fuzz waypoint locations in
-wp-fuzz-bounds-lon	Minimum and maximum longitude (east-west) to fuzz waypoint locations in
-wp-fuzz-bounds-alt	The boundaries inside which waypoint altitudes are fuzzed
-w, -wp-location	Fix one or more waypoints' locations (overrides fuzzing)
-Dname=value	Optional plan arguments. They will be passed to <code>get_items</code> as keyword arguments.

Table 4: Test runner command line arguments

Argument	Description
airframe	The aircraft to generate tests for
output	Output file name or the directory to output the generated plans
-i, --include	Flight plan blocks to include in the generated plans
-x, --exclude	Flight plan blocks to exclude from the generated plans
-l, --length	Number of flight plan blocks to include.
-p, --paparazzi-home	Shared with test runner in Table 4
--fuzz-wps	
--wp-fuzz-bounds-lat	
--wp-fuzz-bounds-lon	
--wp-fuzz-bounds-alt	
-w, --wp-location	

Table 6: Test runner command line arguments

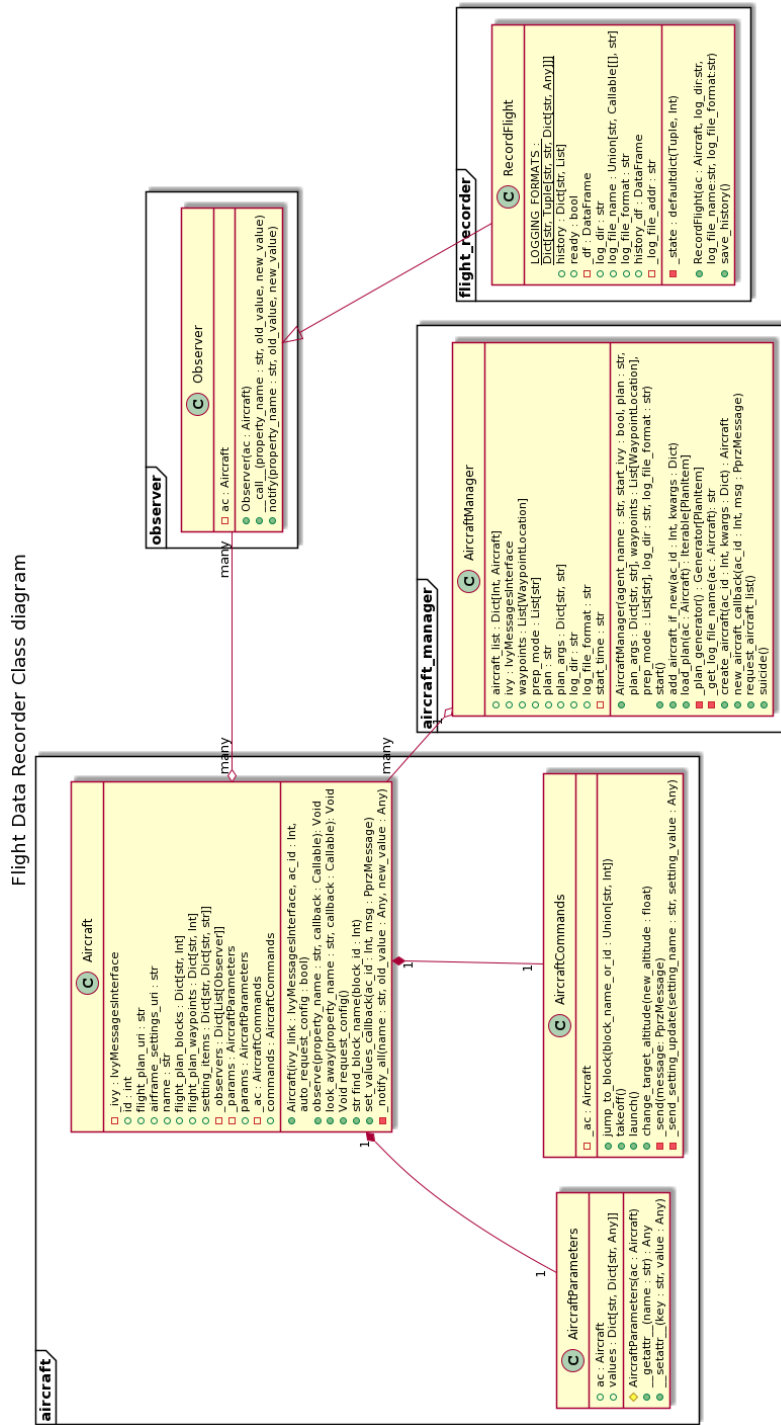


Figure 1: Class diagram for flight data recorder

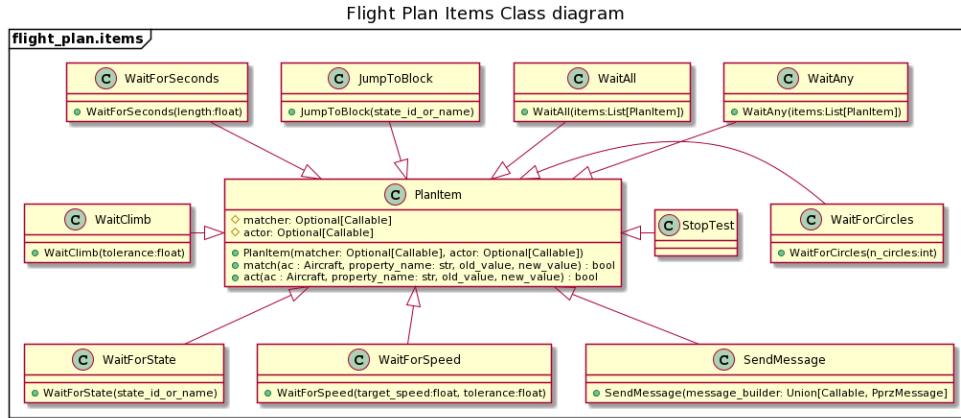


Figure 2: Flight plan items

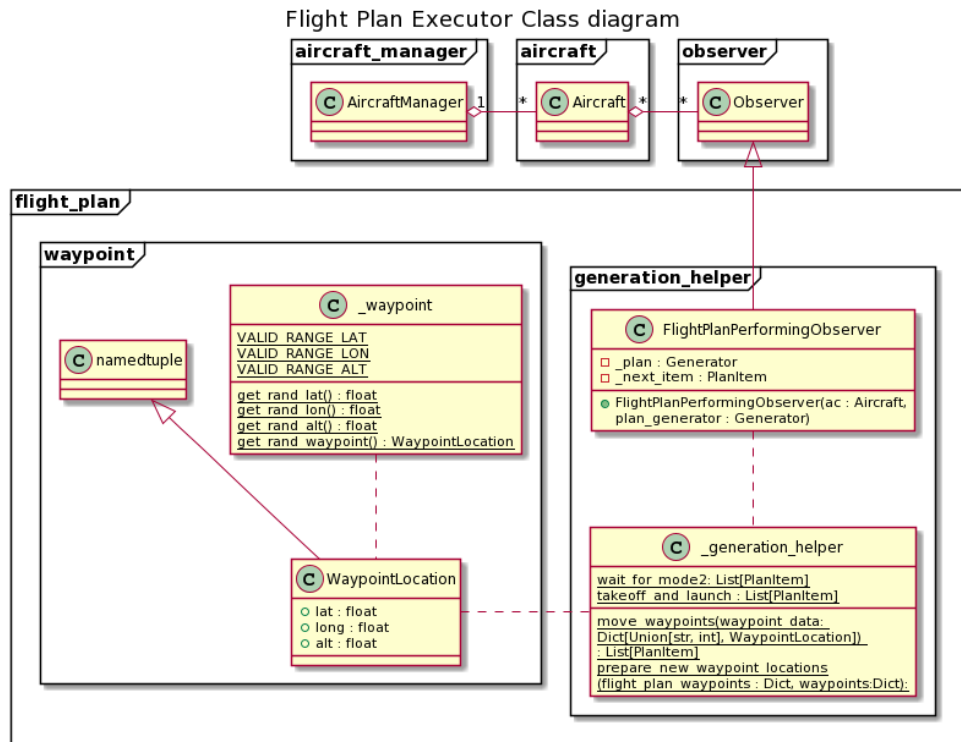


Figure 3: Flight executor classes

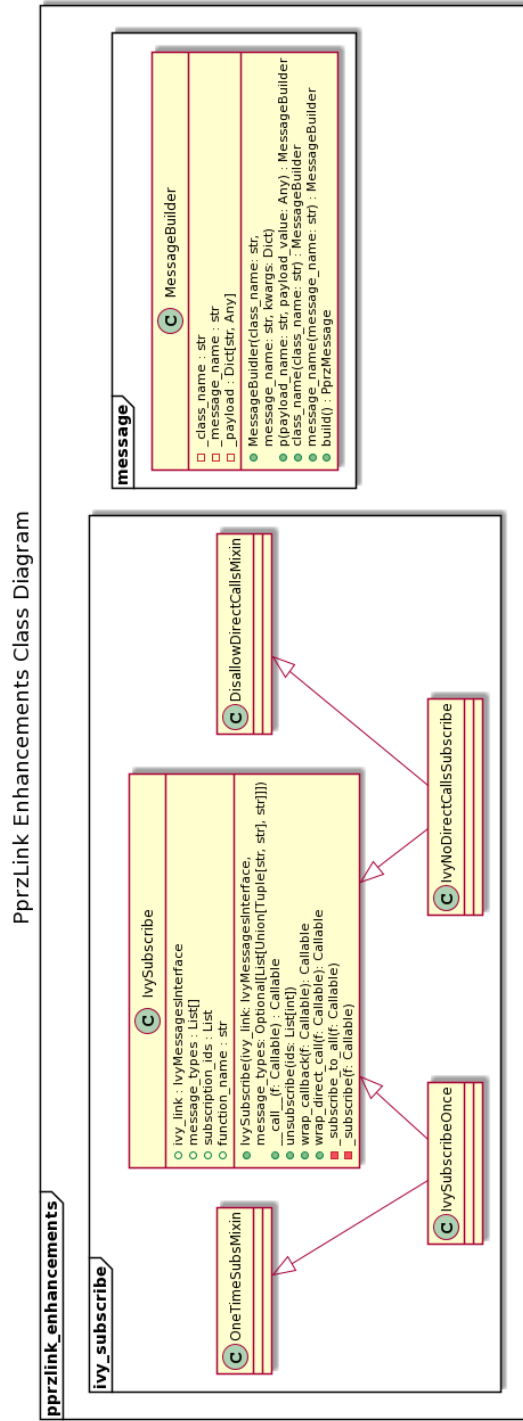


Figure 4: Class diagram for helpers and wrappers for PprzLink library

Appendix B: Copyright

ACM Reference Format:

Mohammad Jafar Mashhadi and Hadi Hemmati. 2020. Hybrid Deep Neural Networks to Infer State Models of Black-Box Systems. In *35th IEEE/ACM*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE '20, September 21–25, 2020, Virtual Event, Australia

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6768-4/20/09...\$15.00

<https://doi.org/10.1145/3324884.3416559>

International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3324884.3416559>

I, as the author, retained the copyright to the above manuscript while licensing ACM for publication.