RWTH Aachen University
Software Engineering Group

Assessing Safety Aspects for Continuous Experimentation on the Example of Automated Driving

Master Thesis

presented by
Crispin Kirchner

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr. S. Kowalewski

Advisor: Evgeny Kusmenko

The present work was submitted to the Chair of Software Engineering

Aachen, February 22, 2017

Deutscher Titel: *Einschätzung von Sicherheitsaspekten für kontinuierliches Experimentieren am Beispiel von automatischem Fahren*

Acknowledgements

Reserved for the non-draft version

This work was supported by a fellowship within the FITweltweit programme of the German Academic Exchange Service (DAAD).

Eidesstattliche Versicherung

Kirchner, Crispin

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit~~/~~Bachelorarbeit~~/ Masterarbeit* mit dem Titel

Einschätzung von Sicherheitsaspekten für kontinuierliches Experimentieren am Beispiel von automatischem Fahren

Assessing Safety Aspects for Continuous Experimentation on the Example of Automated Driving

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Abstract

Continuous Experimentation is a current trend in the development of software-only systems, used to make design decisions based on credible data, instead of opinions and guesswork. Applying this approach to the domain of autonomous vehicles promises large-scale testing, while decreasing time to deployment and increasing software quality. With automobiles transforming from hardware-focused systems to mere software platforms, development speed and short feedback loops enter the requirements list for automotive software. While continuous experimentation might help reaching these goals, it also imposes some major challenges, such as constrained hardware and safety issues. This thesis contributes a framework for Continuous Experimentation in the automotive domain, accounting for these two challenges. Selected ideas of the suggested concepts have been evaluated experimentally, using a self-driving prototype vehicle.

Zusammenfassung

Continuous Experimentation ist eine Technik bei der Entwicklung von Software-Systemen. Ihr Ziel ist es, Design-Entscheidungen auf der Grundlage von belastbaren Daten zu treffen, anstelle von Meinungen und Mutmaßungen. Die Übertragung dieses Ansatzes auf die Domäne autonomer Fahrzeuge verspricht hoch-skaliertes Testen, bei gleichzeitig reduzierter Bereitstellungszeit und erhöhter Software-Qualität. Da sich bei PKW derzeit einen Wandel vom Hardware-fokussierten System zur Software-Plattform vollzieht, stehen Entwicklungsgeschwindigkeit und kurze Rückmeldungszyklen neuerdings auch bei der Entwicklung von Fahrzeugsoftware auf der Anforderungsliste. Continuous Experimentation könnte beim Erreichen dieser Ziele helfen, allerdings stehen dem Herausforderungen entgegen, wie etwa leistungsschwache Hardware und Sicherheitserwägungen. Die vorliegende Arbeit schlägt ein Framework für Continuous Experimentation in der automotive-Domäne vor, welches diese Herausforderungen angeht. Ausgewählte Ideen der vorgeschlagenen Konzepte wurden mithilfe eines autonomen Versuchsträgers experimentell evaluiert.

Contents

List of Figures

List of Tables

Acronyms

AGV autonomous ground vehicle
CE continuous experimentation
CAN controller area network
CPS cyber-physical system
CPU central processing unit
GPS global positioning system
MVP minimum viable product
OEC overall evaluation criterion
OEM original equipment manufacturer
QoS quality of service
RoI region of interest
SaaS software as a service
SUV sports utility vehicle
TTM time-triggered module
UI user interface
VP vanishing point

1 Introduction

With carmakers teaming up with chip manufacturers and mobile communication companies [28], it can be observed that the car is making a shift from being a hardware-focused platform to a software-focused one. At the same time, rather software-centric firms enter the automotive market[1]. The list of companies researching autonomous driving is quite diverse[2], including non-automotive Intel, nVidia and Apple, software-centric Waymo, Uber and Baidu, as well as startup companies like comma.ai or nuTonomy. Automotive newcomer Tesla showcases agile methods by changing the hardware of their cars in mid-production or by upgrading software functionality over-the-air, and offers SAE Level 2 [27] assisted driving[3].

In software-only systems, continuous experimentation (CE) is used in order to make decisions based on credible data gained via systematic experimentation instead of expert knowledge, guesswork [7] or the "highest paid person's opinion" [15, 16]. Using systematic experiments, a variant of the product is presented to a part of the users, in order to decide based on metrics which variant is better (see Figure 1.1). For example if a webshop is modified in order to increase sales, an obvious metric could be the number of sales per visit. A part of the users would then get to see the modifed version of the shop (the *treatment*), while the rest get to see the original (the *control*). During the experiment, the desired metrics are collected (in this example sales per visitor). An experiment has three possible outcomes: Either there is no significant difference in the key metrics, or there is a significant difference either favoring the one or the other variant.

Transferring this idea to the domain of autonomous ground vehicles (AGVs) promises an increase in innovation speed. It would enable evaluating experimental software in the customer's vehicles. This promises testing at very large scale and with high coverage. However, running experimental software along with production software requires additional computation resources, while the hardware in cars is normally dimensioned with small computation power reserves [22, 30]. Furthermore, most of the software in cars is safety-critical [4]. This leads us to the research goal of this thesis:

[1] techcrunch.com/2016/09/20/chinas-leeco-raises-1-08b-to-build-its-electric-sports-car Accessed on December 21, 2016

[2] cbinsights.com/blog/autonomous-driverless-vehicles-corporations-list Accessed on January 12, 2017

[3] tesla.com/presskit/autopilot#autopilot Accessed on February 2, 2017

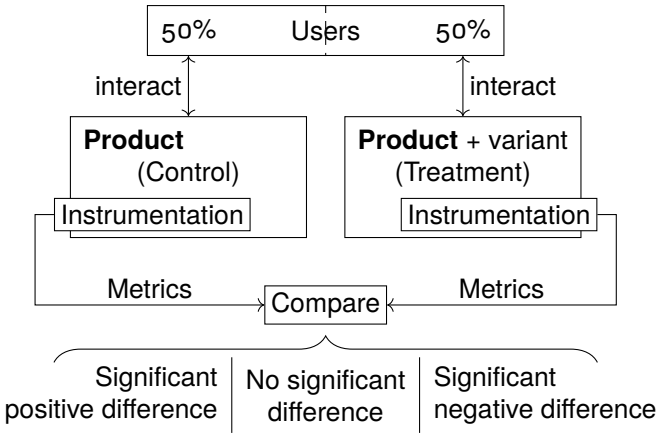


Figure 1.1: Continuous experimentation in software-only systems

RG: Developing the foundations for transferring the idea of continuous experimentation into the automotive domain, while accounting for restricted hardware and safety.

This goal can further be decomposed into three research questions, which shall be answered in this thesis.

RQ1: How can the idea of CE be transferred to the automotive domain?

RQ2: How can the additional computing resources needed for CE be acquired?

RQ3: How can we preserve the safety of the passengers during experimentation?

The current state of the art in CE is presented is Chapter 2. Based on this, a general framework for CE on autonomous cars is described in Chapter 4. For evaluation, a lane following algorithm was implemented and evaluated on a test track. The results are presented in Chapter 5. In Chapter 6 the presented work is related to the research questions, and Chapter 7 gives a conclusion and an outlook on follow-up research.

2 Related Work

In this chapter, the state of the art in automotive software and continuous experimentation is presented. CE is presented from the software-only side, as well as current efforts to transfer the idea to the embedded domain.

2.1 Automotive Software

Automotive software is implemented as a distributed system. This has historical reasons [4], but is also beneficial for mastering the complexity of the overall system [21]. Hardware is organized in multiple computing nodes, which reach a high two-digit number in current vehicles [4, 25, 29] and are organized in multiple hierarchical networks with gateways in between them [21]. This architecture is illustrated in Figure 2.1. Most software in the automotive domain is real-time critical [4]. Real-time applications are either event- or time-triggered [18, 21]. In an event-triggered application, the computation is initiated by an occurring event, which is normally an incoming message from another module. In a time-triggered application all activities are executed in predefined intervals [18], which is the case in modules representing hardware interfaces or assuming control tasks. Examples for time-triggered processes include fetching data from sensors or control algorithms.

In a 2016 e-print [29], Vöst and Wagner describe the process for continuous delivery used in the software-only domain. They afterwards give a detailed report about the

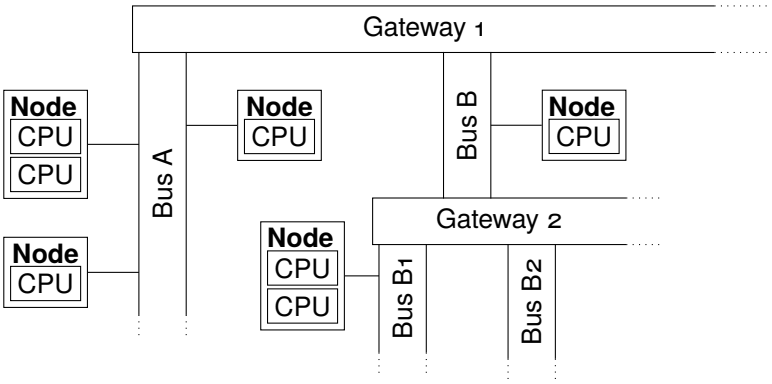


Figure 2.1: Physical system layout

steps which are performed in order to produce new software for the computing nodes in cars. Despite of most of the process being carried out manually these days, they highlight the steps where automation is possible, in order to outline how the concept of continuous delivery could be transferred to automotive software. It is claimed that all steps in the delivery pipeline can be conducted in a continuous fashion, with the exception of acceptance tests on the road, which need to be manual. It is concluded that, although grouping commits into releases continues to be present in automotive software delivery, continuous deployment is also applicable in the automotive domain. The second part of the paper reports on the deployment of firmware updates on Tesla vehicles. They identify three phases in the delivery of a new firmware version: the *release*, which marks the first day on which a large number of vehicles get the firmware update, the *ramp-up* phase, during which the new version keeps to be delivered to many cars, and the *fade-away*, during which only few cars get the update. This last phase is observed to last several weeks. Also, few customers get an update before the release date, which hints that these vehicles are used as canaries. This shows that Tesla is using delivering software in a continuous fashion, however with a delay. The obstacles for continuous delivery in the automotive domain can apparently be overcome, which is promising for transferring CE to this domain.

2.2 Continuous Experimentation

Software engineering in the past aimed solely at producing software of high technical quality [7]. Agile methods allow for short feedback loops, to react to changing requirements, and to re-prioritize requirements continuously during the development process, enabling constant validation during development. This leaves the question on how to determine the actual value of an item, in order to prioritize correctly. This gap is breached by systematic experimentation [19] and helps making design decisions based on data instead of opinions [6]. Experiments with end users or stakeholders make quick evaluation of hypotheses possible and therefore allow to focus development efforts on the features providing the highest value to the user or the company. They can range from measuring the demand for a new feature by providing a mock user interface (UI) element [8], up to evaluating user experience quality [13]. Experimentation is particularly popular in software-as-a-service and web-centric applications, as the centralized deployment facilitates rolling out modified and instrumented functionality [19], and good scaling allows for many experiments at once [15, 26].

Kohavi reported in 2004 about systematic experimentation from a practitioner's perspective at Amazon [17] and in 2009 and 2013 at Microsoft [13, 15]. At Amazon, metrics about site performance were also collected during normal operation. This included measuring response times, availability of the site and generated revenue. In addition to that, multiple A/B test were running every day (2004). Challenges for continuous experimentation were outlined. This included tests which cannot be run at the same time, as they

affect the same settings, or the problem of capturing long term effects, which might contradict short-term measurements. Also, statistical dependencies between tests, leading to distorted results are mentioned, as well as consistency issues. Example experiments were shown, to illustrate how experiment results, especially when observing human behavior, differ from intuitive judgment. Also, different metrics used in the online shopping area were mentioned, being revenue per user, session start and length, cart adds, page abandonment and number of orders.

The 2009 paper [13] shows many examples of online experiments at Microsoft and illustrates the imperative for experimentation in online businesses. It is stated that the deployment process and the proportion of hardware in a product are limiting factors for experimentation, therefore indicating one of the challenges for CE with cyber-physical system (CPS). The drawbacks of up-front customer research in contrast to experimentation are illustrated. These are that up-front analysis has its limits as long as there is not yet a functional product to let the customer use. Also, it is stated that actual customer behavior can differ greatly from statements made in an interview or survey, and that up-front analysis is expensive. It is concluded that experimentation works best in an agile environment. The cultural changes an organization faces when transitioning to experimentation are described. While experimentation can lead to ground-braking changes in a product, its usability for strategic business decisions is limited.

Tang et al. describe in their 2010 paper [26] the challenges faced when allowing only one experiment at a time in a distributed system, and present a solution. They start on the premises of a system which allows experimenters not to have programming skills. An experiment is defined in a configuration, where all parameters which differ from the production system are listed. Running multiple experiments at a time might lead to conflicting parameter sets, or worse, parameter sets which agree, but render the product unusable. This might be the case when two experiments are run simultaneously, one varying the foreground, and one the background color of an advertisement, leading to illegibility. Furthermore, each computing node in the distributed system hosts its own experiments. When allowing only one experiment at a time in the whole system, this necessarily leads to a competition between the computing nodes for experimentation, which is won by the first nodes in the processing chain, as downstream nodes have no choice but to neglect their own experiments when faced with the fact that there is already an experiment running on the current traffic. The solution addresses both the influence between experiments as well as the fair distribution of experimentation among nodes. The parameters of the system are partitioned into layers. Each layer consists of parameters which cannot be varied independently. As a consequence, there is at most one experiment per layer, but experiments on different layers are allowed to overlap. To still support experiments which need to modify many parameters at once, the concept of *domains* is introduced. Opposing to the horizontal division of layers, domains segment traffic vertically. Due to different layers, this allows multiple experiments in one domain, while having a single experiment which modifies parameters all over the layers in another domain. Finally, the paper proposes an approach for gradually rolling out

new features, by using *launch layers*. Launch layers always come before the experiment layers and spread across all experiment domains. Contrary to normal layers, they are not a partition of parameter values, but define new default values for parameters. Therefore they can be used to gradually increase the number of users getting a new feature while using metrics for noticing problems with the new feature. A new feature is entirely rolled out, once the associated launch layer is applied to all traffic. In this stage, the default parameter values are replaced by those of the new feature, and the associated launch layer is removed. The paper defines three *diversion types*, i.e. algorithms to partition traffic into the treatment or control group. In addition, experiments can define conditions based on which traffic is chosen (e.g. geolocation or language filtering). The unused traffic from one experiment layer is not recycled further downstream to prevent biasing.

Kohavi et al. describe in the 2012 paper [14] show multiple cases with strong trends in the metrics, which did not have their root cause in user actions, but rather in misinterpreted metrics, wrong choice of metrics or defect metric implementations. To prevent such errors, it is recommended to critically question tempting results and the metrics used and how they are computed. Also, continuously running A/A tests is recommended. This means dividing the users into treatment and control groups, but presenting the same web app to both groups. Also, strong short-term changes in the metrics should not be interpreted as a trend for the long term, but taken with a grain of salt. For explaining short-term changes, the concepts of *primacy* and *novelty* are introduced. Primacy refers to experienced users having to readjust to the treatment, while novelty refers to thorough exploration of changes by experienced users who spotted the difference, which obviously distorts the metrics. Both effects apply to experienced visitors only, so a possible workaround is presenting treatments susceptible to these effects only to new users.

The 2013 paper [15] of Kohavi et al. illustrates the increase of experimentation at Microsoft on the example of the Bing experimentation system. Apart from the caveats and warnings published in the previous papers, they explain their experiment system for many experiments at once, which resembles the one at Google [26]. Also, their *alerting* system is explained. This incorporates automatic notifications about degraded key metrics and, in severe cases, automatic abortion of experiments.

Amatriain reports about the experimentation system at Netflix [1]. Contrary to the systems at Amazon [17], Google [26] and Microsoft [15], the system is not designed to experiment with arbitrary changes of the user interface. Instead, it targets evaluating machine learning algorithms used for the recommendation system. Consequently, the approach of the system is unique compared to the other experiment systems. It features a two-step process which starts with offline cross-validation of a new algorithm with existing data sets. If this step proved to have better accuracy than previous solutions, an experiment with real users is conducted, in the style presented in the previous papers. As the experiments concern the recommendation system, an online experiment influences the choice of movies a user watches. As this data is used for training the algorithm, it

also affect the quality criteria for future recommendation algorithms. Therefore, the online experiments resemble to running an experiment in a closed control loop on a cyber-physical system, while the offline experiments parallel to having an open control loop.

In a paper [7] from 2017, Fagerholm et al. propose a process framework for continuous experimentation and relate it to case studies conducted with industry partners. The outer structure of the proposed process uses repeated *Build-Measure-Learn* cycles, as in the Lean Startup methodology [23]: Based on an assumption, a minimum viable product (MVP) is developed during the *build* phase. In the subsequent *measure* phase, data about the usage of the MVP is collected, which in the *learn* phase results in learnings, which in turn are used for forming the next assumption and therefore steering what is built in the next cycle. The proposed process model extends this cycle with systematic experimentation: a key part of the assumption is formulated in terms of a hypothesis. To test this hypothesis, an experiment is designed and the MVP is equipped with an according instrumentation, to collect the metrics needed for accepting or refuting the hypothesis. They also define an experiment infrastructure which consists of a data storage which they call the backend, analytics tools, the *frontend*, which consists of the instrumentation tools, and a continuous integration/continuous delivery pipeline. In the context of this terminology, the framework in this thesis focuses on providing an experimentation frontend for the automotive domain, including interfaces to analytics tools.

2.3 Continuous Experimentation in the Embedded Domain

Being able to change the software during operation is crucial for applying experimentation. In domains where changing the deployed software after delivery to the customer is either not desired or difficult, which also applies for the embedded domain, experimentation is not widely applied [10, 11, 24].

The 2012 paper [3] by Bosch covers a broad scope, by presenting different approaches for customer development [2] for the different stages of product development, systematic experimentation and A/B testing being among them. The focus is extended to embedded systems, emphasizing that the benefits of software as a service (SaaS), being cheap deployment and post-deployment data collection, are not restricted to pure software products, but can be extended to embedded systems with an internet connection. For software-intensive embedded systems, this can ease the transition from waterfall development to a feedback and data-driven development process. Insights from a former employer of the author are shared, where A/B experiments were successfully used in an SaaS product [3].

In the paper [6] from 2012, Eklund and Bosch put a stronger emphasis on embedded systems. They describe a development process for software-intensive embedded prod-

ucts with a 2–4 weeks iteration cycle. In each cycle, new functionality shall be evaluated by deploying a new software version to the client devices. For resolving safety concerns, a monitoring solution detecting unsafe states is proposed. The paper maps the experimentation process known from software-only systems to the embedded domain, without applying major changes. The restricted computing resources on embedded devices are not addressed. This contrasts to the approach of this thesis, where changes are made to preserve safety and account for limited computing resources (cf. Chapter 4).

In a paper [11] from 2013, Holmström Olsson and Bosch study the current use of data collection after deployment in the field of embedded products. They present a case study involving three companies. Two of these actually collect data after selling the product, but do not intend to use it for improving the current version of the product, but the next generation. Also, the facilities for changing the deployment over-the-air is missing. Therefore, the data collected is defined at deployment time, and the software on the target device cannot be changed, for example for pushing another experiment to the device or collecting a different type of data.

Giaimo et al. in their 2016 paper [10] list the challenges they see which need to be overcome in order to apply continuous experimentation in the automotive domain. They address the fact that hardware in the automotive domain is due to economic reasons often not fit for running experimentation-related software on top of their actual logic. Furthermore, automotive control units are part of a distributed system. Contrasting to A/B testing in the software domain, feedback does not only come from users but also from surrounding systems, and in consequence experiments also affect surrounding systems. Safety as a central issue for experimenting in the automotive domain is addressed. In contrast to the software domain, the cultural change required at a car manufacturer is likely to be more radical than in an organization that is familiar with agile methods, as it is the case in most software companies. Updating the software of a distributed system like a car over the air needs additional tooling and hardware compared to a web service, where all machines that need to be updated are in full control of the company.

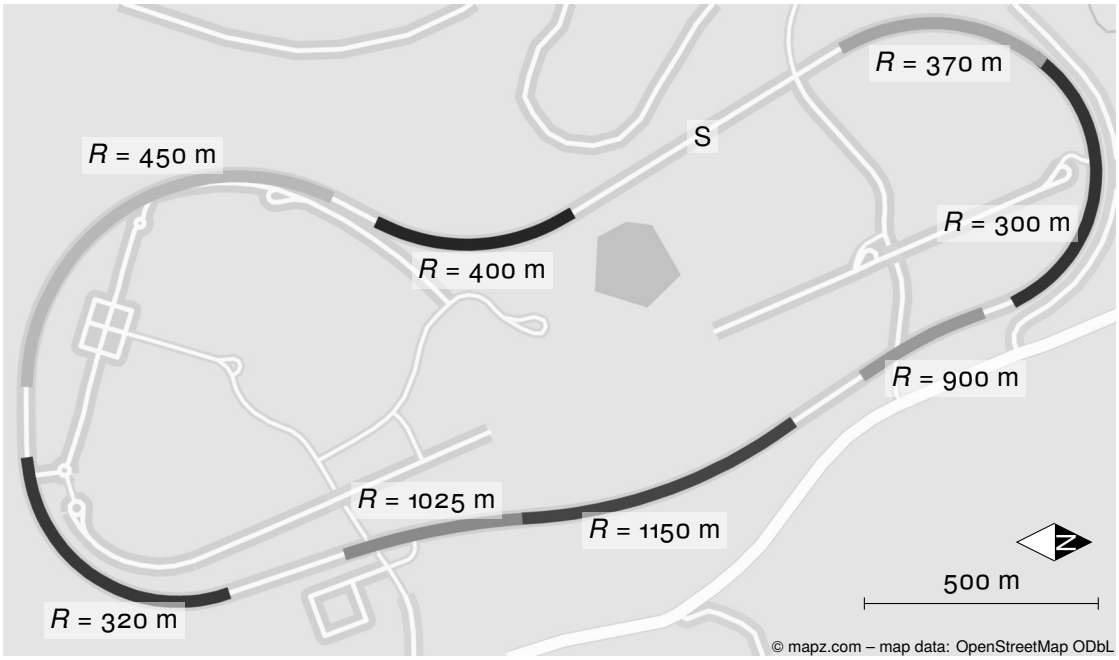
3 Methodology

In the previous chapter we outlined the characteristics of software in the automotive field and researched the current use of CE by practitioners, which is mainly in the field of SaaS. We saw that CE is rarely applied to the embedded domain and that the existing proposals of transferring it into this domain are not directly applicable to the special context of automotive software, due to shortcomings regarding restricted hardware and safety issues. Answering to RQ1, we define a framework for continuous experimentation which is fit for usage in environments with hardware restrictions and aims to have no impact on the safety of passengers (cf. Chapter 4).

To evaluate the real-life needs of experimentation with software for cars, we implemented a lane-following algorithm. The subsequent evaluation of this algorithm is discussed with respect to the proposed experimentation framework.

The implemented steering algorithm is camera-based. The first step consisted in an algorithm for vanishing point (VP) detection. Camera data for aiding the implementation was collected on a nearby highway. A subsequent data collection was done on the test track, with and without other traffic, as well as in different camera mounting positions. This data was used for evaluating a color-based lane markings detection. For robustness concerns, this algorithm was replaced by an edge-based VP detection. This was equipped with a subsequent delta-angle computation and a proportional controller, which sent steering signals to the car. Driving the test track, the points where intervention of the driver was necessary were noted. This allowed an evaluation of the quality of the algorithm. On the different testing days, there were different quantities of snow on the sides of the road, while the road itself was always free of snow. The experienced weather conditions were clouds and direct sunlight.

A mid-size sports utility vehicle (SUV) of which we can control the steering was used as the evaluation platform. As an image source, a customer-grade camera was used. While preliminary data collections were done on nearby Swedish highways in regular traffic, the main experiments were conducted on the AstaZero proving grounds in Sandhult, Sweden. The test track (depicted in Figure 3.1) imitates an undulating rural road with up to 4.5 % incline, changing lane marking quality, and sometimes guardrails. The course is mainly oval, with some minor turns in the opposite direction and 5.7 km length.



R: curve radius, S: longest straight stretch

Figure 3.1: Characteristics of the test track

4 Automotive Experimentation Framework

Due to safety considerations, an experiment framework for the automotive domain needs to differ fundamentally from its web-targeted counterpart. The core of web-based experimenting is first exposing a small number of actual users (the *canaries* [26]) to a new feature in order to detect catastrophic defects early, and then gradually increase and expose up to 50% of the actual users to the treatment to maximize statistical power [16]. As experimental functionality in the automotive domain is potentially unsafe, experimentation needs to differ fundamentally. Belonging to the treatment group must never compromise the safety of the passengers. This leads to the possibility to run experimental software on 100% of the customers' cars, but never in a closed control loop (see Figure 4.2). Summarizing, one can say that opposed to web-based experimenting, it is not the users who are exposed to the treatment, but the data present in the system, which is processed by two different algorithms.

First, the general proposed model for running a production- and experimental module is described. Section 4.2 discusses which data in an automotive system can contribute to metrics computation, and some sample metrics are presented. In Section 4.3 different ways for making use of unutilized computation resources in an environment with restricted computation power are discussed. Section 4.4 presents the types of experiments which we deem relevant for an automotive experiment system, while Section 4.5 illustrates the border conditions which are needed for a possible implementation of the framework.

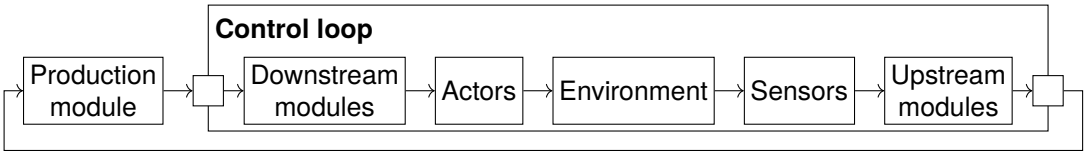


Figure 4.1: General control loop

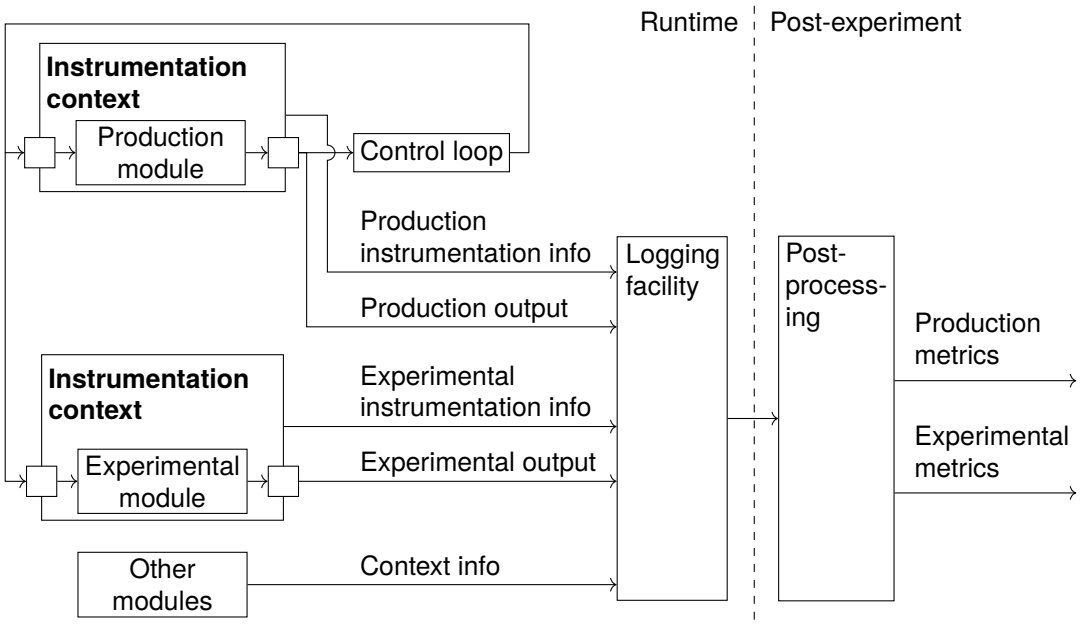


Figure 4.2: General experiment model. See Figure 4.1 for an illustration of the control loop.

4.1 General Experiment Model

In general, a module is embedded in a control loop, consisting of the environment, sensors, a graph of upstream modules, a graph of downstream modules, and actors (see Figure 4.1). An experiment consists of a production module and an experimental module. The production module is considered safe, is therefore incorporated into the control loop and can influence the environment. Production- and experimental module get the same input data stream. As the general communication model is a shared bus, this does not necessarily mean that they process the same data. The output of the experimental module is redirected, to influence neither the other modules in the system, nor the environment. Both modules run in an instrumentation context, to log data about their execution. This can be for example the computation time consumed by the module, or the data which it sent. Depending on the metrics which are collected, also data which is sent by other modules is logged, for example the position of the car. The logged data is post-processed to generate metrics for comparing the modules. Specializations of this model include: only instrumenting the experimental module in order to evaluate it, or having the experimental and production module only differ in the parameter sets they get. The general case is illustrated in Figure 4.2.

The modules run on a distributed system consisting of nodes with at least one central processing unit (CPU), which are interconnected with a message bus system. The bus system can consist of a hierarchy of multiple bus networks, and certain message types

might only be available in some of the subnetworks. See Figure 2.1 for an illustration of the bus architecture.

4.2 Metrics

Metrics are the overall evaluation criteria (OECs) of an experiment. Which metric is appropriate depends highly on the module which shall be evaluated and which properties are of interest. This is visible from the metrics used in software-only systems. Experimentation metrics from this domain are typically defined by the business model of the company and the kind of value which is provided to the user. For the example of a search engine, two typical metrics would be *distinct queries per month* (value) and *revenue per user* (business model) [14]. As the car industry sells goods instead of services, customer value and business model are in unison, therefore it is often not possible to assign metrics to one of the two purposes. The metrics in Table 4.1 serve to illustrate the diversity of metrics to be expected for an automotive application of CE and are discussed in the following paragraph.

M1–M4 are applicable regardless of the module type and examine non-functional qualities of a module. They can serve to assess the achievement of requirements, or for further optimization. M5–M6 both address A/B comparisons of two modules achieving the same task, M5 if the output is a continuous quantity, M6 if the output is the result of a classification. In general, one metric alone provides little value. The combination of metrics however is a useful tool: While M7 alone might not provide valuable knowledge, M7 in combination with M1 or M4 might point out problems with realtime capability. M8–M9 were inspired by the manual evaluation of a lane following module (see Chapter 5). M8 enables viewing the trajectory which was driven during the experiment, while M9 captures the torque which was applied to the steering wheel, enabling to see where the driver corrected the trajectory chosen by the algorithm.

Summarizing, the metrics needed are diverse both in the way they are collected, as in the way the data is post-processed, for example for relating the data to another metric. Therefore it can be assumed that a CE framework for the automotive domain can never be complete with respect to metrics and should therefore offer facilities to the user to define own metrics in an easy way. This consists of defining the data which has to be collected for a metric, as well as specifying how the collected data needs to be post-processed in order to get the metric (cf. Section 4.6.2).

4.3 Execution Strategies

For running an experiment, additional computation power is needed for running the experimental module alongside the production module. As we have to assume that spare computation resources are rare in a typical automotive environment, different ways for

Metric		Instrumentation		Logging Communication		
				Output		
		Time	Cycles	PM	EM	Sensors
M1	Computing time	•				
M2	Time left in time slice	•				
M3	Power consumption		•			
M4	Output jitter				•	
M5	Differences in output			•	•	
M6	Precision/Recall/Specificity			•	•	
M7	Number of recognized objects				•	
M8	Resulting trajectory					📍
M9	Driver intervention					🔄

PM: production module, EM: experimental module, 📍: GPS position, 🔄: Steering wheel torque

Table 4.1: Sample metrics and how they can be collected

acquiring unused computation power are presented, to account for different possible situations. The execution strategies are illustrated in Figure 4.3.

The use case assumed is an A/B test, where a production module and an experimental module which process the same data are evaluated. The modules shall be executed at similar times, to collect performance metrics from both. If the modules process the same data, the execution is in general synchronous, no matter whether the modules are time- or data-triggered. In the case that experimental- and production module process different data, the experimental module is still tied to the execution frequency of the production module, as it uses the spare resources of the production module.

ES1: Parallel In case a second free CPU is available, it can be used for executing the experimental module synchronously to the production module. Problems with this execution strategy might occur if the computation time needed by the experimental module exceeds the computation time used by the production module and the experiment is therefore called with a too high frequency. A possible countermeasure would be suspending and resuming the module (see below). In this way the computation can be completed, and therefore metrics gathered. However, depending on the computing duration, input data which becomes available when the computation is not yet finished, the input data will be skipped, leading to a slower sampling rate.

ES2: Serial If a production component runs exclusively on a CPU, but there are idle cycles between the individual executions, these can be used for an experimental

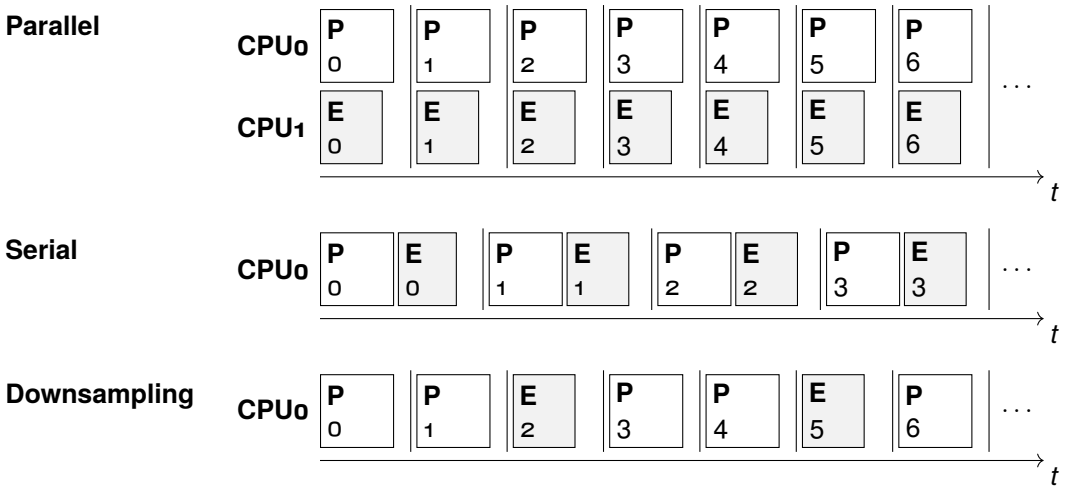


Figure 4.3: Possible experiment execution strategies. P: production module, E: experimental module. The numbers represent the processed input data.

module. The experimental module is invoked after the production module terminates, and has the same input data at its disposal as the production module. If the experimental module does not finish computation before the time slice is over (in case of a time-triggered host) or the next input data arrives (in case of a data-triggered host) it is aborted, possibly leading to metrics loss. A workaround would again be to suspend and resume the experimental module, as discussed below and illustrated in Figure 4.5.

ES3: Downsampling Due to the marginal computation power overhead in the embedded domain it might happen that neither the parallel, nor the serial execution strategy are applicable. In such cases, it can be an option to reduce the sampling rate of a production component and use the time slots where it is not executed for the experimental module. However this requires to check under which conditions how much downsampling is safely applicable (see ET3 in Section 4.4.2) and to enforce that downsampling is only performed in these conditions. The *downsampling rate* states how often per second the production module is replaced by the experimental module (see Figure 4.4).

In any case, the resources of unused functionality may be tapped. For example if an assistance function is deactivated when it rains, but an experimental module can be evaluated in spite of bad weather, the associated unused resources can be allocated to the experiment. However, the experimenter should be aware that this potentially introduces bias to the experiment results and should in such a case also run the same experiment with other resource acquisition methods, for cross-checking.

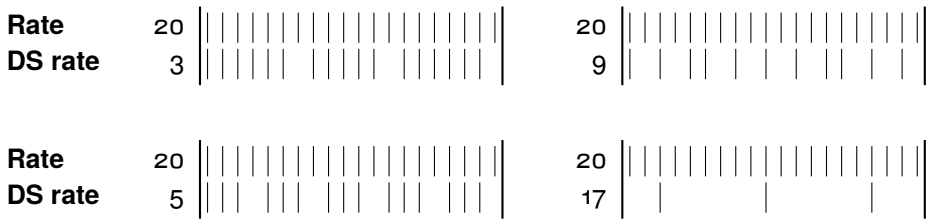


Figure 4.4: Different downsampling rates at production module frequency of 20 Hz. A vertical dash represents an execution of the production module, absence of a dash stands for execution of the experimental module.

Suspending Execution All execution strategies harbor the inherent risk that the acquired computation time does not suffice for executing the experimental module. The simple consequence would be that the execution is aborted, which in the worst case leads to no metrics and therefore an unsuccessful experiment. If this is probable to happen during the experiment, the execution of the experimental module can be suspended when its computation resources are exhausted, and continued in the next possible time slot. See Figure 4.5 for an illustration. This would lead to the experimental module to be executed less frequently, but always completely, leading to less, but complete metrics data.

Hybrid Strategies If a node consists of multiple CPUs, hybrid strategies for acquiring computing time could be applied. This means that the execution strategy is not fixed, but the experimental module makes use of whatever free computing resources are available on the node. For example, it might start its execution in parallel to a production module on a second CPU, then be suspended because that CPU is claimed by another production module. It could then be resumed on the first CPU, after the respective production module has finished computation (see Figure 4.6 for an illustration).

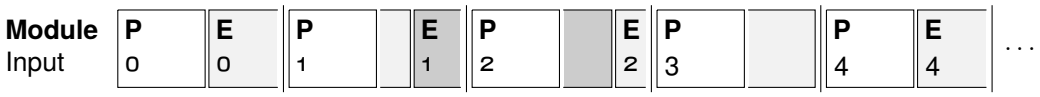


Figure 4.5: Suspending execution of an experimental module on the example of serial execution. In this example, the input data with input 3 is not processed due to the exceeded computation time of the experimental module.

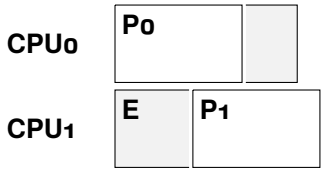


Figure 4.6: Using hybrid execution strategies to tap multiple vacant computing resources at once

4.4 Experiment Types

In the style of the Netflix experimentation system [1], the possible experiment types are divided into *online* and *offline*. While the first group comprises experiments which can be safely run in a production environment, they might require additional information to be conducted safely, or at all. The preliminary experiments which serve to gather this additional information need to be conducted offline, i.e. in a controlled environment, where the safety of passengers is not at stake.

4.4.1 Online

Experiment types of this section fulfill an experiment purpose, such as evaluating an experimental module or comparing two modules. They are designed to be deployed and ran on customer cars without endangering the passengers.

ET1: A/B Two modules fulfilling the same, or a similar task are compared. The experimental module might hereby use a different algorithm for computing the same output. It might be either a modification of the production module, or an unmodified copy running with an alternate parameter set. The comparison might concern non-functional or functional properties of the modules, for example comparing the needed computation time, or the output accuracy of the modules. To be able to compare the behavior of the experimental and the production module, the same conditions need to be enforced for both. Therefore it is important for this experiment type to systematically preempt side-effects of the modules, for example by copying the input data before the first module processes it, to make sure that both modules actually get the same input and hence produce comparable output. As described earlier, it is imperative that only the production module has access to actors, while the output of the experimental module is logged.

ET2: New module A module which is not yet present in the production setup is fed with realistic input data from a real-life environment, but in an open-loop setup, to ensure safety. The output of the new module might be post-processed to find situations in which a closed-loop setup might have led to potentially dangerous behavior.

In comparison to ET1, the production module and the experimental module are not related. Gathered metrics only apply to the experimental module, the production module serves as a host, administering its spare computation time. Acquiring computation resources for this experiment type therefore has less restrictions. It might be possible to use an entirely unrelated computing node, as long as the necessary data is available in the subnetwork it is located in.

4.4.2 Offline

Experiment types of this section need to be executed in a controlled environment. In spite of their results contributing to preserving the safety of passengers, these experiment types themselves are unsafe. This aspect is similar to the Netflix experimentation system [1], which also incorporates an offline cycle prior to the actual experiment, in that case for screening bad algorithms and protect the user experience.

ET3: Downsampling This experiment type is designed for determining the maximum acceptable downsampling rate of a production module. To achieve this, the sampling rate of a module running in a closed loop is gradually reduced. Using a metric characterizing the quality of service (QoS), and the minimum acceptable QoS, the maximum acceptable reduction of the sampling rate can be determined. This knowledge is valuable for running an experiment with the downsampling execution strategy (cf. Section 4.3). For finding the limit of the downsampling rate, it has to be exceeded. As this is potentially hazardous, this experiment type cannot be conducted in a production environment, but needs a controlled environment. The determined downsampling rate is therefore highly dependent on the conditions of the downsampling experiment. When conducting the subsequent online experiment, it is therefore imperative to enforce similar conditions. This could for example be achieved by using geofencing, or restricting the experiment to a certain speed range and automatically stop experimentation when it is exceeded.

ET4: Unused computation time If a time-triggered experiment uses the *serial* execution strategy (cf. Section 4.3), it is unknown whether the time left in each time slice is sufficient for fully executing the experimental module. In the worst case, and if suspending the module is not an option, the experimental module would never be fully executed, leading to no gathered metrics. Therefore it is desirable to assess prior to running the experiment, whether, how often, and under which circumstances the unused computation time is sufficient for the experimental module.

ET5: Engineering In software-only systems, experimentation aims at observing user behavior. Therefore, experiments necessarily need to be run at large scale in order to achieve statistic significance. The aim of our proposed framework is contrary to this, in particular the passenger is not designed to be part of the experiment.

The subject of experimentation in our case is the behavior of algorithms. As algorithm behavior is easier generalized, the large scale is not necessarily required. In this case the framework degrades from the online, large-scale scenario to a use case that is similar to a conventional engineering approach: New algorithms are gradually developed and continuously verified in experiments. However, the envisioned framework offers aid in data collection and post-processing in the form of metrics. These metric computation facilities provide value even in a conventional engineering setting, as they assist evaluation, no matter whether an experiment is run large-scale on a fleet of customer cars or on a single prototype vehicle. This experiment type shall support this exact use case, for testing modules which have not yet reached a maturity where online testing is sensible. The use case of this experiment type is the evaluation of a new module in a single prototype vehicle on a test track. As prototype vehicles in general do not suffer the restrictions of series production cars, but dispose of additional computers for running experimental functionality, the proposed execution strategies (see Section 4.3) are not needed; the resources needed for the experimental module are assumed to be available. The metric computation facilities are mandatory for large-scale requirements. For a singular manual experiment they are not, but as they are present they can still be used to the benefit of the experimenters. Therefore the idea of this experiment type is to run small-scale experiments, but still make use of automated metric computation for faster evaluation.

4.5 Context Measures

In order to coordinate the running of production- and experimental modules, and to ensure isolation of the experimental module with respect to the production modules in the system, the framework needs to offer the possibility to modify the context of the modules.

Isolating communication Messages from the production system need to be forwarded to both the production and the experimental module. As for data-triggered modules this redirection at the same time serves for triggering the module, the framework needs to take control over the data flow that reaches both experimental- and production module, in order to coordinate the experiment. This is especially relevant when using the downsampling execution strategy, as the input redirection governs which module is executed. In the case of serial execution, the input needs to be copied and cached until the production module finishes execution. In case of parallel execution, the input also needs to be cloned. Copying the input is necessary to prevent side-effects of the production module to affect the experimental module and therefore bias the experiment. To make sure that the experimental module cannot interfere with the production modules in the system, the messages sent by the experimental module must be intercepted such that

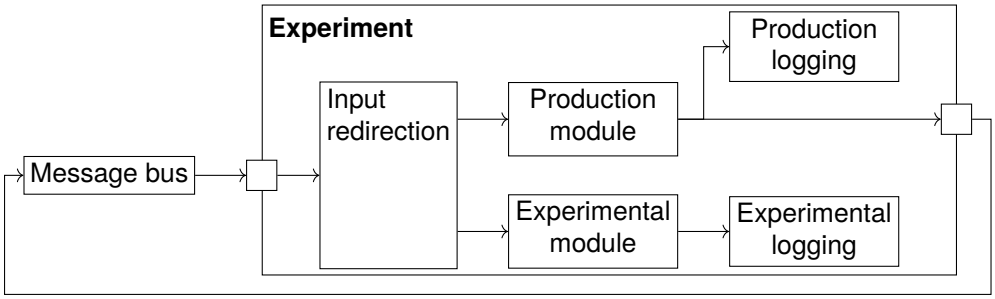


Figure 4.7: Isolating the communication of the experimental module

they will not end up on the message bus of the production system. As the bus systems present in automotive systems in general do not offer addressing, there are per default no means to check where a message was sent from. However, for computing metrics, it is necessary to associate sent messages with the production- or experimental module. Therefore, a facility to intercept sent messages of the production module to log them before forwarding them to the bus, and therefore become indistinguishable, is necessary. Furthermore, a facility for logging the output of the experimental module is needed. Figure 4.7 shows a setup which fulfills these requirements.

Modifying the clock While data-triggered modules rely entirely on incoming messages on the bus for their timing, time-triggered modules use a clock. This clock needs to be modifyable by the experiment framework in order to coordinate when which module is run, especially when using the downsampling experiment type. Furthermore, if a module is suspended and resumed later on, the closk should be paused as well, such that the module behaves as if it were run in a contiguous time slot.

4.6 Proposed Framework Design

In this section we give an overview on how an implementation of the proposed framework could look like.

4.6.1 Data- vs. Time-Triggering

As discussed in Section 2.1, we can assume that software modules in the automotive domain operate either data- or time-triggered. An experimentation framework therefore has to offer the according experiment types. A design which supports this is shown in Figure 4.8. We assume a middleware providing the abstract classes *Module*, *DataTriggeredModule* and *TimeTriggeredModule*, as scaffolding. In case of data triggered execution, the method *nextMessage()* is executed each time a new message arrives on

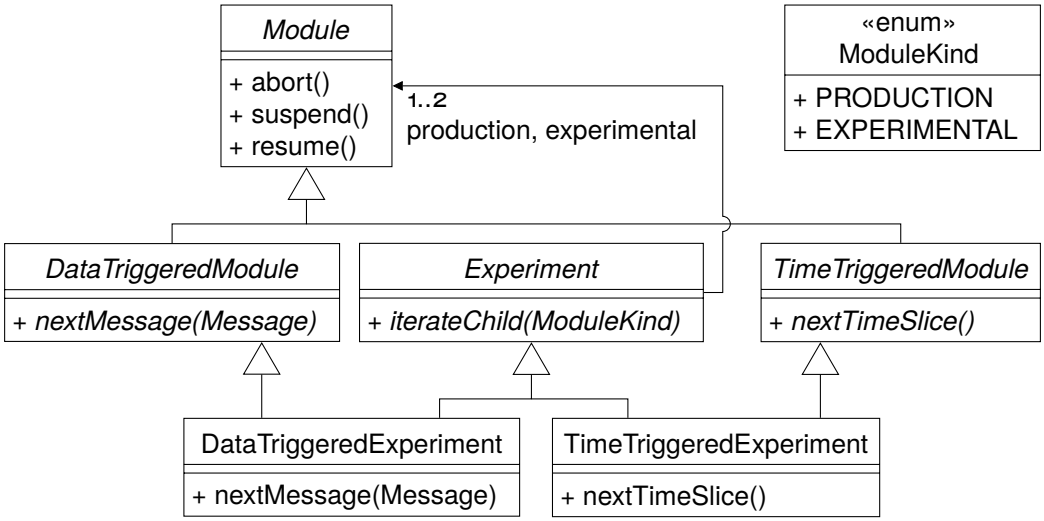


Figure 4.8: Class structure for data- and time-triggered experiments

the bus. The *Message* type is part of the middleware and is assumed to be an abstract data type which can hold an arbitrary payload. For time-triggered execution, it is assumed that the hook method *nextTimeSlice()* is called with the frequency of the module. The methods of the *Module* class are only necessary when suspending modules is desired (see Section 4.3).

The presented design is an implementation of the *composite* pattern [9]: An experiment contains modules and is itself a module. Aspects which differ between data- and time-triggered execution are dealt with in the respective classes *DataTriggeredExperiment* and *TimeTriggeredExperiment*. In comparison to the composite pattern, the *Experiment* class adds a layer of indirection and contains elements which are not affected by the way the experiment is scheduled. These are most prominent the production- and experimental module themselves. The number of child modules is not fixed to two; depending on the experiment type (cf. Section 4.4), an experiment might only have either of the two. The method *iterateChild()* provides an interface to trigger a child module, regardless of the triggering and is used by the execution strategy (see Section 4.6.3).

4.6.2 Metrics Collection and Post-Processing

To compute metrics, it is necessary to log data. As discussed in Section 4.2, the required data consists of three different types: information about the execution of a module gained through instrumentation, like the computation time, or the number of CPU cycles used. Furthermore, the output of either the production- or experimental module might be needed, and lastly the output from other modules in the system, like for example sensors. To unify this information in one log, the abstract type *LogData* is introduced,

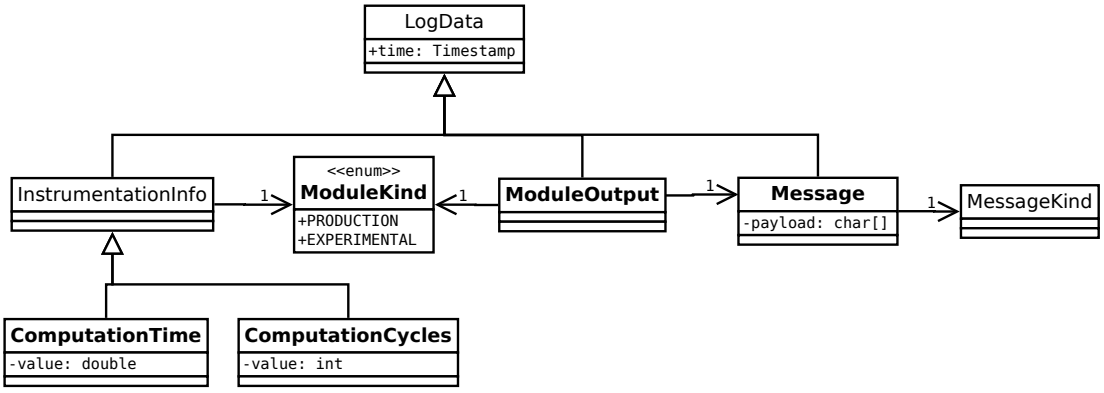


Figure 4.9: Data types that can be logged

which can hold any of the aforementioned types accompanied by a time stamp (see Figure 4.9).

A metric consists of two elementary parts: the information of what data needs to be collected, which is given in the form of a *LogDataKinds* object, and the post-processing logic. The latter forms part of the *Post-processing* block in Figure 4.2. The post-processing logic is defined by subclassing the *Metric* class. As the following evaluation, including possible dashboard applications is outside the scope of this thesis, we remain with having a not further specified complex type which can be used for visualizing the gathered metric.

4.6.3 Experiment Setup

All information which is needed to fully set up an experiment is depicted in Figure 4.11. The execution strategy is decoupled from the experiment. This eases offering all execution strategies for both scheduling types. When an experiment is triggered, it performs eventual copying of input data and then delegates to the *iterate()* method of *ExecutionStrategy*. The different implementations then use the back-connection from *ExecutionStrategy* to the *Experiment* to execute production- or experimental module, using the *iterateChild()* method. Furthermore, the execution strategy has an *EndPolicy* as a parameter which states how the experimental module shall be treated if the computation time is exhausted, aborted or suspended.

As a means for data collection, the experiment has a *Logger* instance which receives instances of the abstract *LogData* class (see Figure 4.9). What data should be collected is given by an instance of *LogDataKinds* (see Section 4.6.2). The LogDataKinds needed can be derived from the metrics which should be collected, but can also be defined without having a ready metric. This has the advantage of being able to collect data on suspicion, even if the experimenter does not know yet what to do with it. As LogDataKinds is essentially a collection, multiple instances can be merged using the *add()*

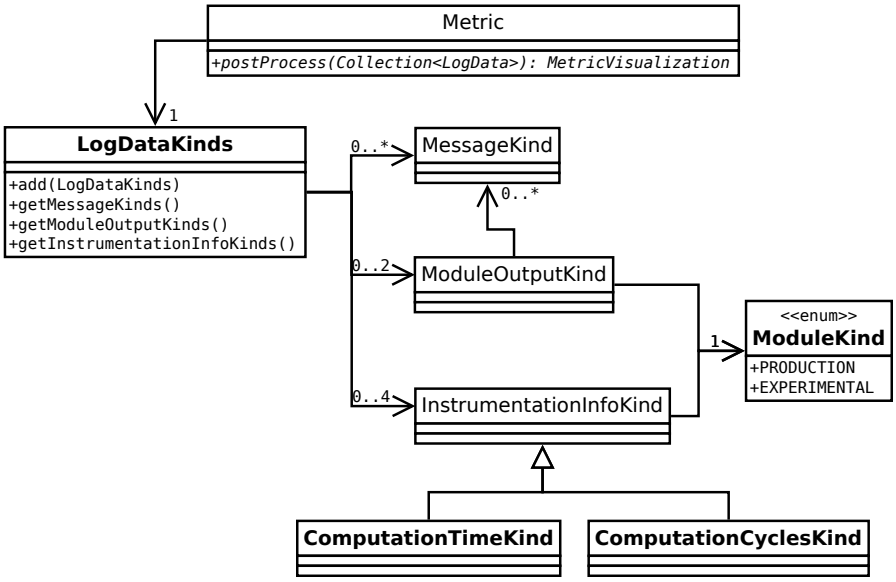


Figure 4.10: Specification of which data to log

method, for example when the data for more than one metric shall be collected during the experiment.

Experiments in general should not be carried out disregarding environment conditions. This could be due to the use of downsampling as an execution strategy (see Section 4.3), or might also be due to the nature of the experimental module, for example if the experimental module concerns highway driving, it is sensible to only execute the experiment on highways. This aspect is covered by the *ExecutionConstraint* class. The subclasses presented are intended as illustrative examples, not as an exhaustive list. As determining when the constraint is met is also highly dependent on the type of constraint, it shall be possible for the user of the framework to implement custom constraints.

4.7 Experiment Process

Based on the individual parts described in the previous sections and the ideas of the RIGHT model [7], we propose an experimentation process (see Figure 4.12). This process applies to the online experiment types. The offline experiments, with the exception of the engineering use case (ET_5), are an integral part of the process. We assume an existing continuous delivery process, as this can be assumed to be feasible even in the automotive industry [29]. An experiment is initiated by an hypothesis. The role mainly involved in conducting the experiment is that of the experimenter. Based on this hypothesis, an experiment is designed. The experimenter specifies what module is to be used

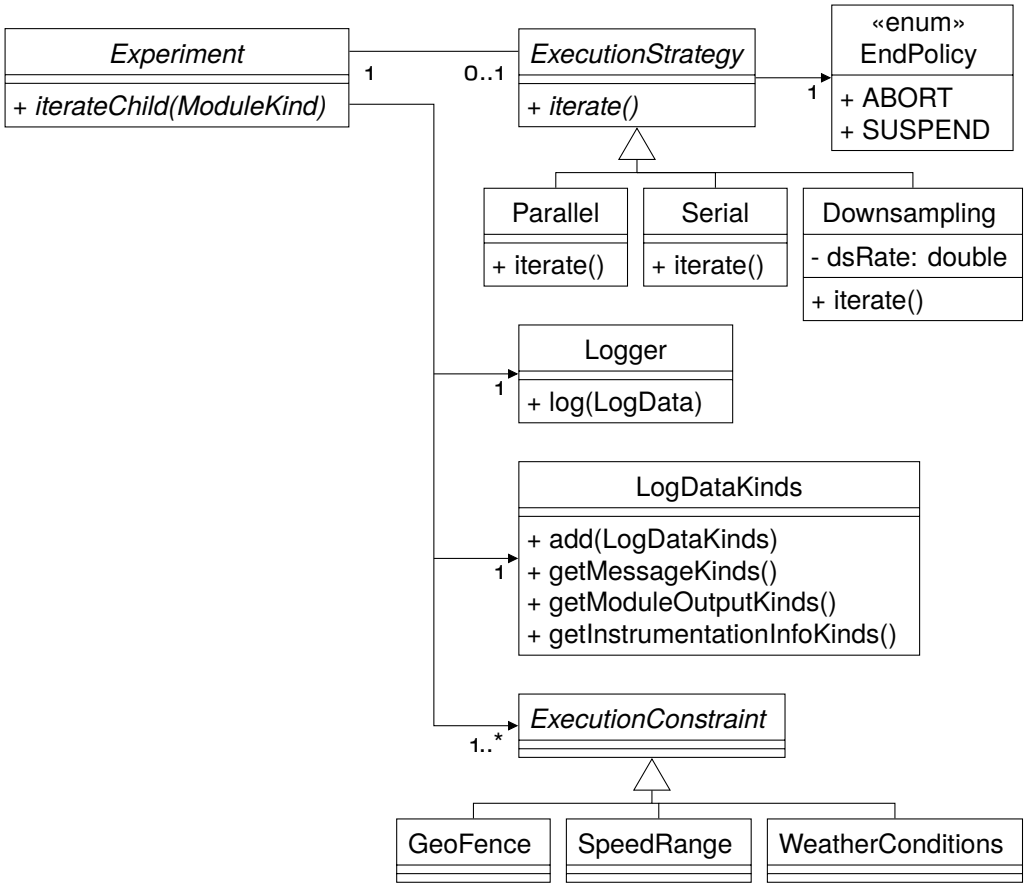


Figure 4.11: Information contained in an experiment setup

as an experimental module and which metrics shall be collected. Aspects regarding the limited hardware resources and safety of the experiment are decided by a domain expert. She chooses an execution strategy, which might result in the need of first conducting an offline experiment. As the offline experiment only involves the production module, in the role of hosting the experiment, the offline experiments can be conducted even when the experimental module is not yet available. Setting the constraints is a tasked shared by domain expert and experimenter. The domain expert shall ensure that the constraints are set such that the experiment can be conducted safely. The experimenter can add further constraints regarding the functionality of the experimental module, or negotiate constraints that might introduce a bias with the domain expert. Also, the choice of the production module can be decided by experimenter or domain expert: if the experiment is an A/B test, the experimenter will choose the production module. If new functionality is evaluated, the production module can be chosen by the domain expert, depending on what resources in terms of messages and computation time are anticipated to be required by the experimental module. When all artifacts are ready, the experiment can be orchestrated combining the necessary metrics, execution strat-

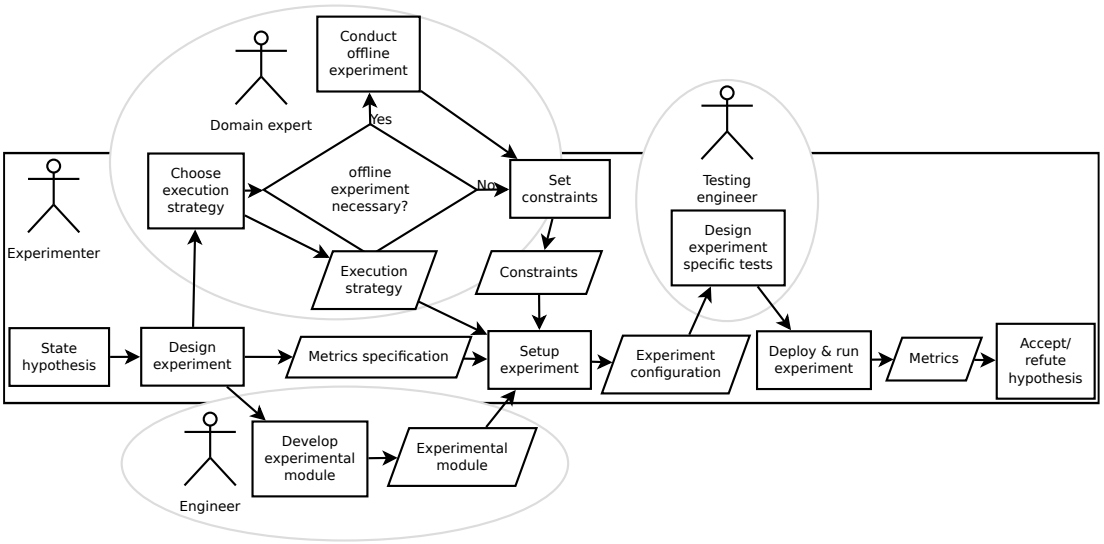


Figure 4.12: Experiment process

egy, constraints, experimental- and production module into the experiment. Using this compound information, a quality assurance engineer develops additional tests, tailored to the experiment. Subsequently it can be delivered using the existing continuous deployment pipeline. During and after running the experiments, the metrics are available to the experimenter for evaluation and analysis.

5 Steering Control

To evaluate the ideas of the engineering experiment type (see ET5), a steering control was implemented and evaluated. The algorithm uses a consumer-grade camera as input and controls the angle of the steering pinion, which is related to the steering wheel angle. To allow for evaluation during development, different data collection runs were conducted. They are described in Section 5.1. The algorithm consisted of two subsequent steps: The vanishing point detection, as described in Section 5.2. From the vanishing point, a correction angle was computed and translated into a steering angle, as described in Section 5.3. The results of the manual evaluation are presented in Section 5.4. The factors which influenced the quality of the algorithm are presented in Section 5.5. These results are then related to the proposed framework in the following chapter.

5.1 Data Collections

As a foundation for developing the lane following algorithm, different data sets were collected and used in the development and evaluation of the algorithm.

Public highway As an initial data set, camera data was collected on a public highway. The recorded track is approximately 1.8 km, the recording was done at daylight, with a covered sky. The driven speeds were 60 km/h to 70 km/h. The reconstructed trajectory is depicted in Figure 5.2a. The data set contains lane markings of different size and quality, and other traffic. The number of lanes is mostly two per driving direction. The camera was mounted in the top-right corner of the windshield (position A in Figure 5.1). Sample images from the data set are show in Figure 5.3a.

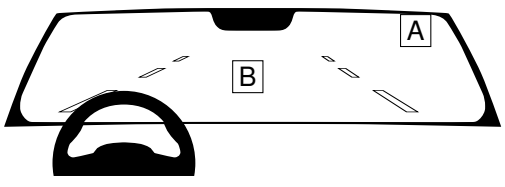
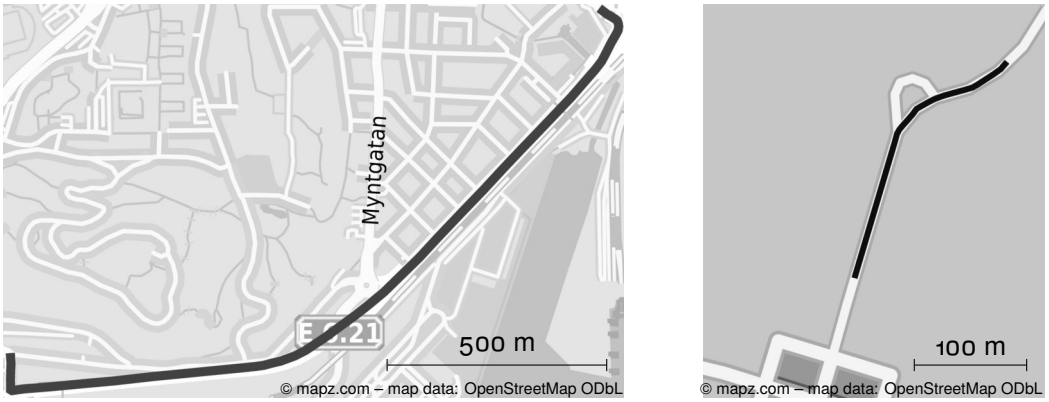


Figure 5.1: Used camera positions



(a) Public highway (b) Proving ground access road

Figure 5.2: Reconstructed trajectories of the data collections

Proving ground rural road For a controlled environment, two data collections were run on the proving grounds track (see Figure 3.1), one without, and one with other traffic. The track characteristics are given in Chapter 3. The driven speeds were 30 km/h to 40 km/h, the camera position was identical to the one used in the highway data collection. First, the lap without other traffic was recorded. To also capture possible influences of other traffic on the algorithm, the subsequent lap was recorded with two more vehicles. Maneuvers included the recording vehicle being overtaken by one to two vehicles, one to two vehicles coming towards the recording vehicle, and the recording vehicle overtaking one to two other vehicles. Samples from the data set are shown in Figure 5.3b.

Proving ground access road When the development of the VP detection algorithm had progressed enough to consider steering the car, it became clear that the asymmetric mounting used in the previous data collections would lead to problems when deriving the steering angle. Therefore the camera was mounted in the center of the windshield (position B in Figure 5.1). Two recordings were made on a short right-left turn segment on the proving ground, one in each driving direction. The recorded stretch is approximately 250 m, the speeds driven were 10 km/h to 30 km/h. See Figure 5.2b for a map and Figure 5.3c for sample images.

5.2 Vanishing Point Detection

Two versions of a vanishing point detection were implemented and evaluated with the collected data.

(a) Public highway

(b) Proving ground rural road

(c) Proving ground access road and RoI

Figure 5.3: Sample images from the data collections

5.2.1 Carolo Cup

The first algorithm evaluated followed the ideas from a participant team of the Carolo Cup [12]. The algorithm was implemented in C++. It uses a color-based thresholding for segmenting lane markings from the surrounding road. Lines representing the lane boundaries are fitted through the detected markings. The vanishing point is computed as their intersection.

Preprocessing A RoI, given by the user, marks the area in the camera image mostly occupied by the road. The image is converted into grayscale and blurred to filter noise. Afterwards, a threshold is applied. Pixels below the threshold are colored black and

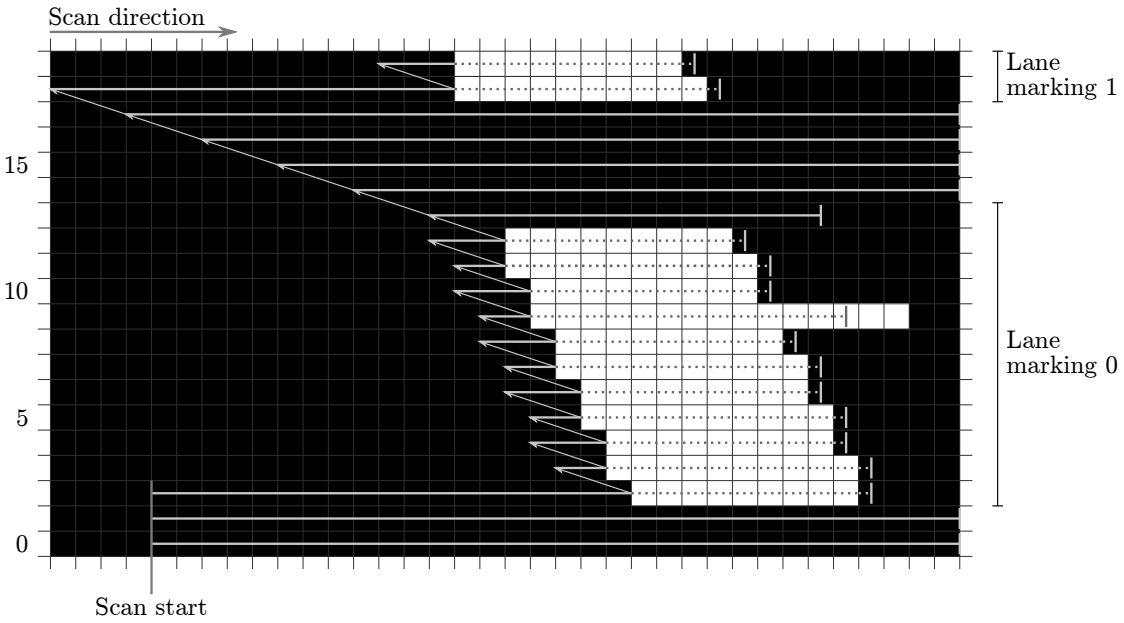


Figure 5.4: Scanning procedure of the Carolo Cup algorithm

pixels above the threshold are colored white. White pixels are assumed to belong to a lane marking.

Scanning A sophisticated scanning routine, which makes assumptions about the position and orientation of the lane markings, is used for detecting them in the preprocessed image. The algorithm is explained on the example of a lane marking on the right, the procedure for the left is analogous, with all operations working in the reverse direction.

Scanning starts at the bottom row of the RoI and at an x value which is known to be in between the left and right lane marking. The image is scanned to the right until either the image border, or a white pixel is reached (lines 0–2 in Figure 5.4). If a lane marking is found, all white contiguous pixels up to the 30^{th} on the line are counted as belonging to the lane marking. The limit is imposed in order to ignore a lane marking that appears wider due to an object next to it that is falsely colored white during the preprocessing (see line 9 in Figure 5.4; for illustration reasons the algorithm aborts at the 13^{th} pixel).

The algorithm then continues at the next line of the picture. If it is currently processing a lane marking (lines 3–13 and 18–19), the algorithm starts the scan three pixels left to the first white pixel on the previous line, omitting most of the line and therefore saving computing time. The scanning start therefore follows a sawtooth pattern (lines 2–12 in Figure 5.4). If the algorithm scans 33 black pixels (30 maximum plus three offset) after a lane marking was found, or the maximum line count for a lane marking is reached, the

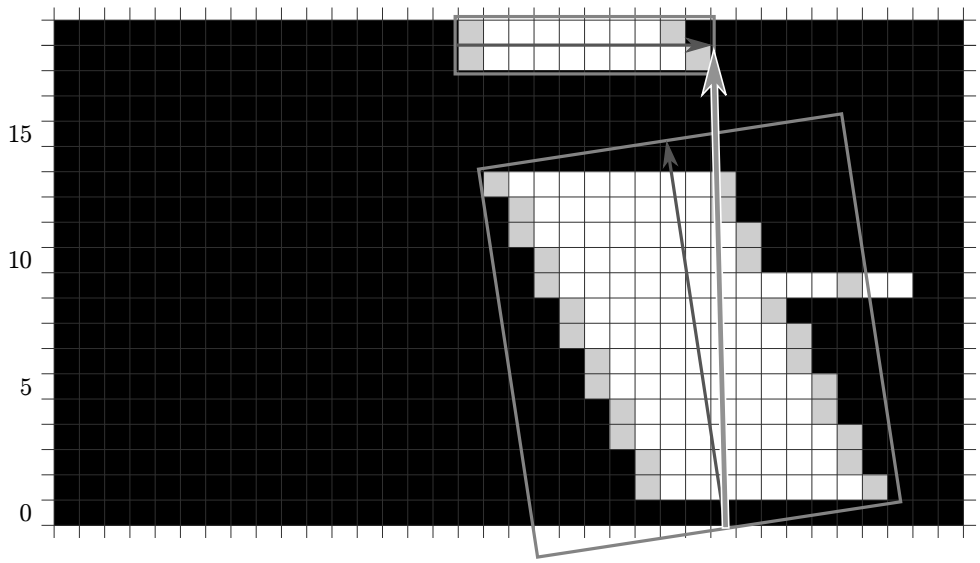


Figure 5.5: Fitting procedure of the Carolo Cup algorithm

lane marking is stored (line 12 in Figure 5.4). The maximum line count is introduced to split long lane markings into several sections, which makes the fitting step more accurate. The algorithm then proceeds with exploring for a lane marking. The scanning start in this case follows a ramp pattern (lines 12–18 in Figure 5.4). The scanning terminates after reaching the top of the RoI, or having found two lane markings. This number was chosen because in our test data, in the case of dashed markings, there were most of the time two lane markings inside the region of interest which were big enough to be recognized.

Fitting After scanning, the found lane markings are represented as a collection of their bordering pixels (shaded pixels in Figure 5.5). Around these, a rotated rectangle is fitted, such that it contains all pixels of the lane marking and is of minimum size. The *orientation* of a rectangle is defined to be upward-pointing, parallel to the longer edge of the rectangle, and along its center axis (thin arrows in Figure 5.5). This is done for all detected lane markings on each side of the road. If there are two lane markings, the start of the orientation of the lower one is connected with the end of the orientation of the upper one (thick, white-framed arrow in Figure 5.5) to form the lane boundary. If only one lane marking is detected, it represents the boundary of the lane. Both lane boundaries are interpreted as infinite lines, the vanishing point being their intersection.

Issues While this algorithm worked well in the artificial environment of the Carolo Cup, it did not perform to our expectations on real data. The fixed threshold was not suitable for usage on real image data. Near the top of the region of interest, the road

became brighter in all our test data, due to the flattening angle and therefore more reflection of light. This led to distorted results and false positives at the top border of the RoI. At the same time, these false positives had a strong influence on the direction of the resulting lane border, which displaced the vanishing point seriously. Therefore, even with strong lowpass filtering, the algorithm output was not calm enough to be used for steering.

5.2.2 Hough

To overcome the problems with the Carolo Cup VP detection, being the fixed threshold and high impulse errors in the result, an alternate solution was implemented. This version has no concept of lane markings, but instead looks for straight lines in the region of interest and computes a weighted average. To enable rapid prototyping, the algorithm was implemented in Python.

Preprocessing A user-defined RoI is extracted. As before, it is the portion of the image that in all driving situations contains the road ahead of the vehicle. The image is blurred to filter noise and the Canny edge detection algorithm [5] is applied. Afterwards, we apply a probabilistic Hough transform [20] to extract lines from the image. Line segments which are perfectly vertical or horizontal are discarded, as these often proved to be false positives (for example reflector posts) in our test data.

Vanishing point computation The resulting line segments are of different length. Both the road boundaries and the lane markings are expected to be long edges. Therefore we use the detected edge length as a quality indicator. The longest 20 lines are selected (dark lines in Figure 5.6). In the subsequent steps, the lines are regarded as infinite, to be able to determine the intersection points. For computing intersection points (small white crosshairs in Figure 5.6), only pairs of lines which enclose an angle greater than 20° are considered. This limit is included to not compute intersections between lines that represent the same side of the road. The intersections are weighted with the product of the lengths of the participating lines. Using these, a weighted average P_0 is computed ($\times$-mark in Figure 5.6). To remove outliers, only intersection points closer than 25 pixel to P_0 are considered (white circle in Figure 5.6). If there are no points inside this radius, it is increased at maximum two times by the factor 1.5, until there are at least 40 intersection points inside the radius. Of these candidate points, the weighted average is taken as the vanishing point (big crosshair in Figure 5.6). The search radius represents the confidence of the value; the smaller it is, the higher is the quality of the result.

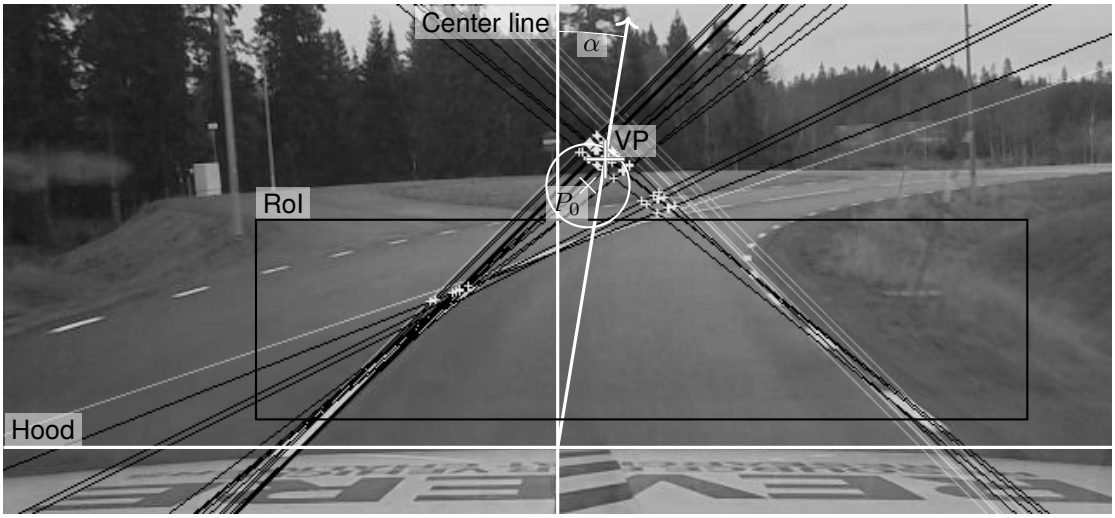


Figure 5.6: Hough-based vanishing point computation. RoI: region of interest, P_0 preliminary vanishing point, VP: vanishing point, α: correction angle.

5.3 Downstream Processing

To account for noise in the signal, a lowpass filter was applied to the detected vanishing point (see Figure 5.7). The currently detected vanishing point was weighted with $K_i = 0.5$. For steering the car, the detected vanishing point needs to be translated to a pinion angle. With the camera mounted in the center of the windshield, the optical axis is parallel to the driving direction. In the lower part of the camera images, the engine hood is visible. To compute a correction angle from the vanishing point, a straight line is drawn from the center of the hood through the vanishing point. The angle between this line and a vertical line through the center of the image is taken as the correction angle (α in Figure 5.6). This angle is again lowpass-filtered, this time with a weighting of $K_i = 0.1$.

The resulting angle was fed to the steering controller. The controller was intended

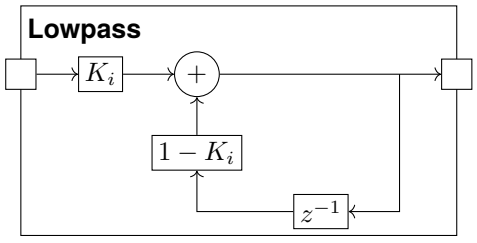


Figure 5.7: The employed lowpass filter. $K_i \in [0..1]$.

Figure 5.8: Overall feedback loop of the steering controller. VP: vanishing point, α: angle between center line and vanishing point. β: pinion angle.

as a full PID controller. However, during parametrization it became obvious that a simple P controller was sufficient for the task. The limiting factor turned out to be the vanishing point detection, not the controller. After observing that the steering was too extreme on high error angles, we applied an arcus tangens to the error before weighing it with the P gain, which due to its logistic character eliminated that problem. The full control loop is shown in Figure 5.8. To account for translation differences when mounting the camera, we introduced a shift of the center line. For compensating rotations around the optical axis, a constant offset was introduced, which was added to the error angle.

5.4 Closed-Loop Evaluation

The steering controller was evaluated manually on two days on the test track. Each evaluation day consisted of a re-calibration of the controller and the subsequent data collection.

5.4.1 Calibration

After re-mounting the camera, the steering controller needed recalibration of its parameters. The calibration incorporated two parameters: an angle offset, which is added to the error, to compensate a constant roll angle of camera or car, and the P-gain of the controller, which is dependent on the pitch angle of the camera due to the way the steering angle is computed (see Section 5.5). To ease calibration, all parameters could be modified via a graphical user interface during operation. The shift of the center line, which had been introduced in the hope of being able to fine-tune the car's alignment on the road, led to asymmetric steering and was therefore kept disabled.

If the car is steering to one side on straight tracks, then the offset angle is not correct. The offset angle was calibrated first, because without it the P-gain could not be determined, as an incorrect offset led to asymmetric values for left- and right curves. For calibration, we drove the longest straight stretch on the test track (see Figure 3.1), modifying the offset angle, until the car was not leaving the road anymore while on a straight stretch.

The P-gain determines, figuratively speaking, how strong the algorithm steers. To determine this parameter, we went around the track, gradually reducing the gain when the

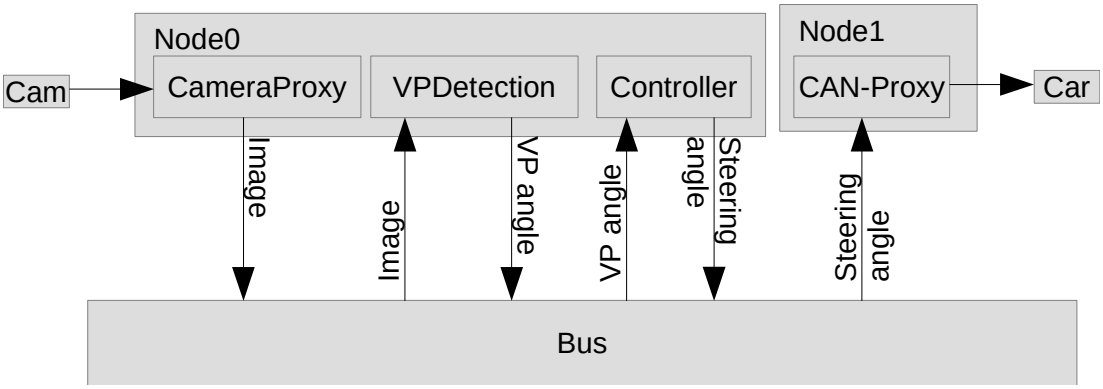


Figure 5.9: System setup during evaluation

steering was too much, and raising when the steering was too little, until a compromise for all bends, in which the vanishing point detection produced plausible results, was found.

5.4.2 Evaluation

During evaluation, longitudinal control was done manually, assisted by an automatic speed limiter. The speed driven was 35 km/h. On each day, we drove multiple rounds on the track clockwise and counter-clockwise. The VP detection turned out to be very dependent on the conditions (see Section 5.5). We publish the laps where the algorithm performed best. Whenever manual steeering was necessary in order to not leave the road, the car was stopped and the current position was noted down. To log the intervention points, the Android application *Osmand~*[1] was used. The posterior data analysis was done with the desktop application *Viking*[2]. The stretches highlighted in the Figure 5.10 and Figure 5.11 are reconstructed from the logged intervention points, they do not represent logged GPS data. Only stretches which were successfully driven for more than 300 m are considered in the evaluation. The software used in the experiment ran distributed on two computing nodes (see Figure 5.9). The first node ran the camera interface, image processing and the controller. The second node provided the interface to the car and sent the appropriate controller area network (CAN) messages to the car's message bus. As OECs, the length of successfully driven stretches as well as the curve radius were used, as the algorithm was during development observed to be less accurate in narrow curves. The stretch length served as an overall criterion for the performance of the algorithm.

From the results in Table 5.3 it can be observed that the algorithm performed distinctly better on day one. This was mainly due to weather conditions. On day one the sky was

[1] osmand.net Accessed on February 4, 2017
[2] sf.net/projects/viking Accessed on February 4, 2017

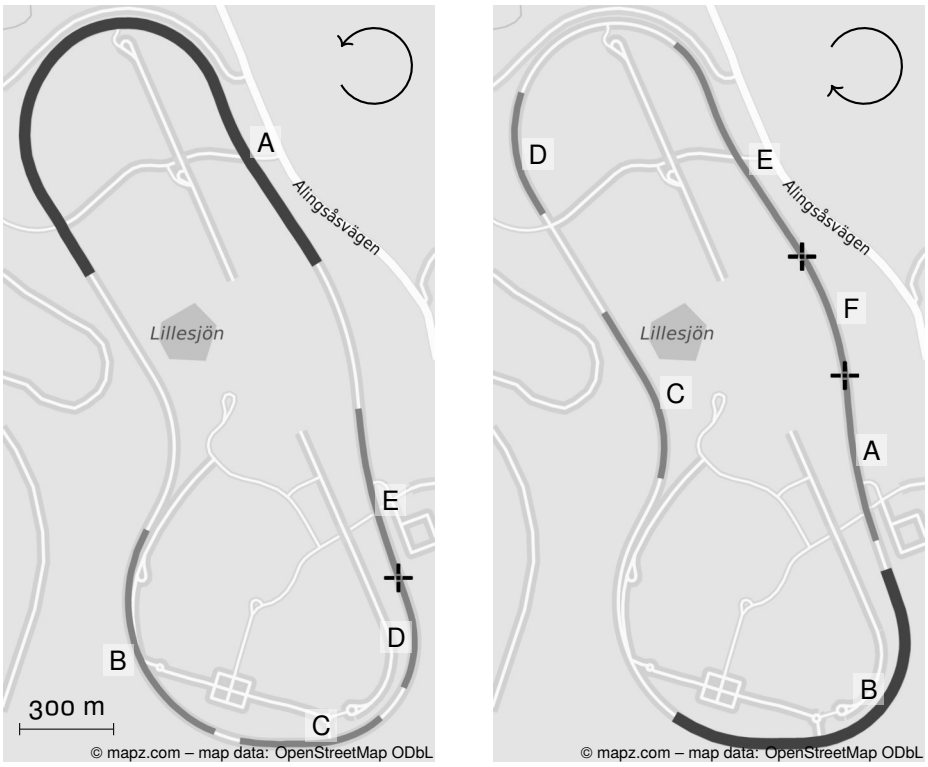
Stretch	Length [m]	Minimum curve radius [m]
A	2000	300
B	740	450
C	445	320
D	349	320
E	541	1025

(a) Counter-clockwise

Stretch	Length [m]	Minimum curve radius [m]
A	522	1025
B	1130	320
C	551	400
D	386	370
E	776	300
F	405	1150

(b) Clockwise

Table 5.1: Evaluation results on day one



(a) Counter-clockwise

(b) Clockwise

Figure 5.10: Stretches driven without manual intervention on day one

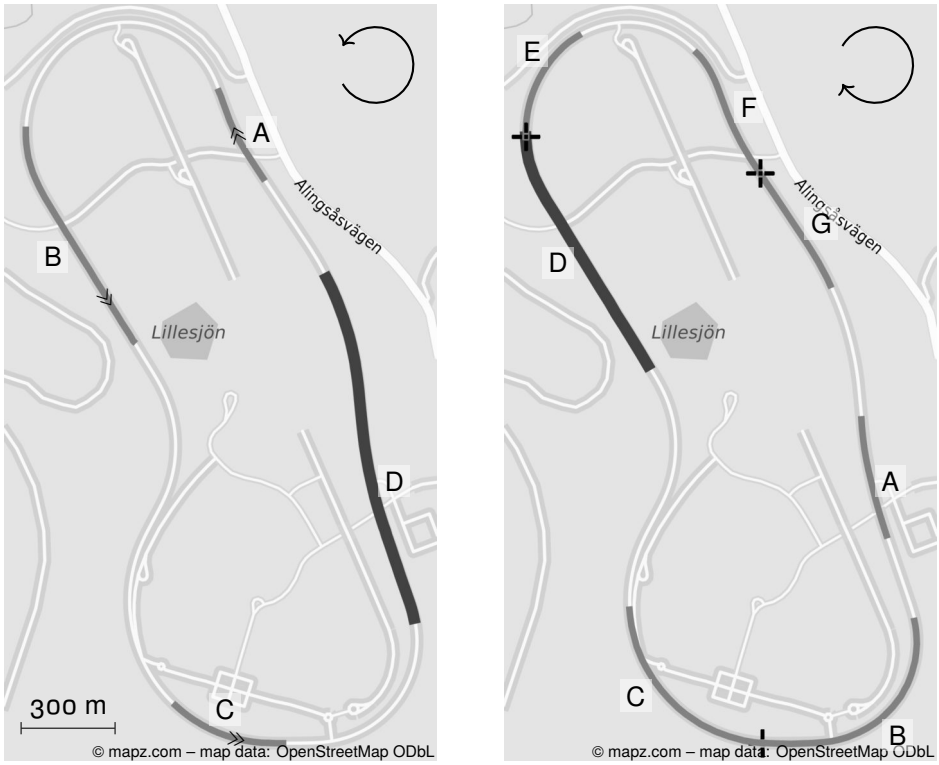
Stretch	Length [m]	Minimum curve radius [m]
A	309	900
B	758	370
C	370	450
D	1110	1025

(a) Counter-clockwise

Stretch	Length [m]	Minimum curve radius [m]
A	376	1025
B	734	320
C	648	450
D	824	370
E	378	370
F	432	900
G	416	1150

(b) Clockwise

Table 5.2: Evaluation results on day two



(a) Counter-clockwise

(b) Clockwise

Figure 5.11: Stretches driven without manual intervention on day two

		Stretch length					Min. curve
		Total		Min	Max	Average	radius
Day	Run	[m]	[%]	[m]	[m]	[m]	[m]
One	CCW	4075	71	349	2000	815	300
One	CW	3770	66	386	1130	628	300
Two	CCW	2547	44	309	1110	637	370
Two	CW	3806	66	376	824	544	320

CCW: counter-clockwise, CW: clockwise, percentage values are relative to one full lap of the test track (5.75 km)

Table 5.3: Evaluation results summary

covered, while there was often direct sunlight on day two, which impaired the vanishing point detection.

5.5 Influence Factors

This section discusses the influences which were observed to degrade the QoS of the algorithm.

Camera mounting position The camera used was not fixed in the car. Therefore it had to be mounted manually each time the algorithm was used. As it is impossible to mount the camera exactly the same way each time, there was some variability. As a counter-measure, we implemented a tool which displayed the camera image along with guidelines. The guidelines showed the desired positioning of the upper hood boundary and the center line (cf. Figure 5.6). To align this, we marked the center of the hood with tape. This way we achieved reusability of the RoI. However, as the correction angle was computed using the y value of the engine hood (cf. Section 5.3), small differences in this y value lead to different steering angles. Therefore, the P gain of the controller had to be re-calibrated in each run. An alternate angle computation, which would reduce the influence of the mounting error, is discussed in Chapter 7.

Sunlight Sunlight influenced the algorithm in two ways: glare and drop shadows. Direct sunlight shining into the camera led to a reduction of contrast between the lane markings and the road. As a consequence, far less high-quality edges were detected, resulting in a strong decrease of VP detection rate and increased inaccuracy. The second effect produced by direct sunlight were drop shadows on the road. The edges of the shadows produced edges which were weighted high in the VP estimation and therefore

disturbed it. This effect was particularly impairing with drop shadows across the road, leading to many false positive edges which made the resulting VP jump to the sides, and therefore seriously decreased the detection performance. Summarizing, the algorithm worked best under a covered sky.

Objects on the side of the road The RoI of the VP detection is necessarily larger than just the lane in front of the car, to account for curves. The left-hand side additional space contains the opposite lane and possibly additional lane markings, contributing to the quality of the detection. However, the right-hand side additional space extends to the ditch. During the test rides, this area accounted for many wrong vanishing points. The reason were different objects in that area with well visible edges, for example snow blocks, bushes, guardrails and different kinds of marker posts. These objects often led to the vanishing point being detected too far right and therefore making the car steer off the road. On the test track, there are different types of guardrails mounted. Some contributed to a more accurate VP detection, while others worsened it.

Snow The influence of snow cannot be accounted as purely positive or negative. The snow blocks on the side of the road, as mentioned in the previous paragraph, had a negative influence. However, the snow also marked the side of the road with a high contrast, thus helping in situations where lane markings were missing. On the other hand, the snow-road edge was an irregular line, which increased the noise in the VP signal. Furthermore, as snow reflects light well, it certainly contributed to the sunlight effects which lowered the contrast of the lane markings and therefore the performance of the VP detection.

Lane marking quality The test track offers a variety of lane marking quality. A missing lane marking on the right often led to the car aligning further right than desired, as instead of the lane marking, the road boundary was used for computing the vanishing point. At some locations the center lane markings were small and with big gaps between them, which also degraded the performance of the VP detection and sometimes misled the car to driving onto the oncoming lane.

Other traffic and curve radius Beforehand, these two factors were deemed to have a high influence on the quality of the algorithm. During development, it was observed that other traffic was unlikely to cause disturbances of the algorithm, as the test data we had collected suggested that the other traffic was mostly outside of the RoI. Also during development, we could observe that the algorithm quality decreased in narrow bends, as the misdetection rate increased and there was therefore a higher noise in the VP signal. As can also be seen from the evaluation data, this problem was not present on the test track, but the aforementioned factors prevailed this effect. This was most likely also due to the fact that the curve radius of the data set used during development

(cf. *proving ground access road*, Section 5.1) was smaller than the curve radii on the test track.

6 Discussion

To address RQ1, a framework for continuous experimentation for the automotive domain was proposed. The framework adapts the concept used in the software-only domain [15, 26]: Eklund and Bosch propose to expose the user to the experimental functionality, as practiced in the software domain [6]. While this approach is possible for non-safety-critical components, like for example infotainment systems, it is not promising for safety-relevant features. To challenge this aspect, and in answer to RQ3, we propose to run experimental software in isolation, without access to actors. Instead of exposing the *user* to experimental functionality, we expose the *data*, by processing it with experimental algorithms, in addition to the existing ones. While this is an important measure for safety, it cannot be the only one: RQ2 addresses the fact that the computing resources in cars are generally at a premium. This also indirectly affects safety, because in order to run a safe experiment, it must be ensured that the resources taken by the experimental software are actually available, and not drawn from safety-relevant modules. Different techniques to overcome limited resources are proposed in the framework. We also propose *offline* experiments, in the style of [1]. These experiments can be conducted in a controlled environment, in order to assess the available resources, and therefore prepare for large-scale experiments in the field, as well as using the metrics provided by the framework also for conventional testing on a test track. With respect to the experiment architecture proposed by Fagerholm et al. [7], our proposed framework occupies the spot of an experimentation frontend.

6.1 Lessons Learned from the Steering Experiment

To evaluate the ideas of the framework, we conducted an evaluation of steering algorithms. While the evaluation was done manually, it allows us to generalize the techniques we applied, to assess whether they are covered by the experimentation framework.

During the evaluation, we had to stop the car each time the algorithm drove us to the edge of the road, to log where intervention was necessary. There are multiple ways of automating this process, by either monitoring the car position in relation to the road, checking when the car leaves it, or by checking the torque applied to the steering wheel, to capture manual intervention. Looking at the observed influence factors of the algorithm, the illuminance in front of the car could be captured with an appropriate sensor, serving for a quantification of the influence of direct sunlight. As a quality property of the

algorithm, the minimum curve radius successfully driven could be extracted from the global positioning system (GPS) log.

An implementation of the framework would aim to automate the extraction of the aforementioned metrics. The developer would therefore after a day at the test track dispose of lots of data about the different factors which influence the algorithm, and could possibly correlate which influence the quality of the algorithm the most, therefore being able to focus on these issues which promise the most progress.

We also observe that the possible metrics are most likely more than can be anticipated. An implementation of the framework should therefore provide an easy interface for users to define and share own metrics.

6.2 Threats to Validity

For situations where neither an additional CPU, nor superfluous computing time is available, we propose the *downsampling* execution strategy. In order to be safely applicable, this requires prior offline experiments to determine how much downsampling is acceptable before the system is not operable in a safe manner any longer. Using this execution strategy in a production environment requires to meticulously keep track of the situations in which the determined downsampling rate is safely applicable, and during the experiment use constraints to make sure that it is not executed outside these limits.

With the different execution strategies, we offer techniques to exploit the available computing resources to the maximum. However, collecting the necessary metrics to evaluate an experiment also means collecting big amounts of data. This data harbors two unaddressed concerns: preliminary it needs to be stored, requiring a sufficiently big data storage in the car. Second, it needs to be transferred to the experimenter. With over-the-air software updates already being a challenge for the industry, one can expect that transferring large volumes of data is not an easy endeavor.

If an experiment compares two possible solutions for the same problem, metrics need to be collected of both. In order to run the experiment and collect the metrics, even the production module needs to be encapsulated into the experiment context (cf. Section 4.6.1) and its communication needs to be isolated (see Figure 4.7). This necessarily causes an overhead. For real-time critical and safety-relevant features, this slight overhead might already compromise the safe operation. It is therefore, regardless of the context, always necessary to run thorough preliminary tests, to ensure safe operation. The experiment we conducted was an offline evaluation of new functionality. As we dispose of an experimental vehicle, in contrary to a series production vehicle, computing resources are available in abundance. The experiment therefore did not allow an evaluation of resource aspects of the framework.

7 Conclusion and Future Work

All products comprised of both hardware and software are limited with respect to agility, as hardware typically has long release cycles [13]. Furthermore, embedded hardware is often tailored to the use case, and therefore only disposes of limited spare computing resources. However, these limits can be exploited to make the most of the situation. In this thesis, we presented an approach to maximize resource usage on the constrained hardware of cars, with the goal of collecting data about the behavior of new algorithms in real environments fast, in order to get quick feedback in the development process. We proposed a framework aiming to transfer the idea of continuous experimentation, as applied in the development of software-only systems, to the automotive domain. We addressed the challenges of restricted resources on preliminary defined hardware platforms, as well as measures to preserve the safety of passengers during experimentation. One key distinction of the framework in comparison to the approach in e.g. web application development is that not the user is exposed to experimental functionality, but the data in the system is processed by different algorithms, which is a core warrant for maintaining safety. A manual evaluation of a steering control was conducted in order to gain knowledge about the process, which was used to see how our framework fitted this practical use case.

The following paragraphs address the possible next steps in the research.

Decoupling experiment parts In most scenarios, we assume a high coupling between production- and experimental module, given that they process the same data and therefore run with the same timing. Running the production- and experimental module on the same node facilitates this setup. This also couples the parts of the experiment in terms of computation resources. The resources required by the experimental module need to be superfluous on the side of the production module, or it even needs to provide some of its own processing power. However, considering all the modules in the system, each of them might be able to provide some processing time it does not need. Introducing a concept of *computing time donors*, all modules in the system could contribute to a pool of unused computation time, available for experiments. This could lead to a better utilization of free resources. Even different computing nodes might be chosen for the experimental- and the production module. As long as they reside inside the same subnetwork, they should dispose of the same communication facilities and should therefore be interchangeable. This could also be a solution to the problem of a production

module not being able to spare resources, which might be the case for highly safety critical modules.

Generating simulation environments For safety reasons, experimental modules cannot be run in a closed loop on the client side. Potentially unsafe situations need to be found in the logged data. Assuming a method to automatically identify potentially dangerous situations, combined with measurements of the surroundings, a simulation environment could be derived from the real-world driving situation. In this simulation, the experimental module could be run in a closed loop, to check whether the critical situation would actually have proven hazardous if the experimental module was running in the real-life scenario.

Full experimentation stack In the architecture proposed by Fagerholm et al. [7], our proposed framework occupies the spot of the experimentation frontend. However, a full experimentation stack has advantages; ideally the experimenter needs no programming skills at all to set up and deploy experiments. The concept of post-processing facilities as proposed in our framework is an important interface to analytics tools, and therefore takes a first step into the direction of a full stack. As shown by Vöst and Wagner [29], continuous deployment is possible in the automotive domain, which leaves rich analytics tools in combination with a centralized experiment database as future extension points on the way to a full experimentation architecture.

Improve angle computation The method to compute the correction angle from the vanishing point was highly dependent on the mounting of the camera and required a re-parametrization of the controller gain each time. Instead, an alternate solution would be to use the knowledge about the horizontal field of view angle γ of the camera in conjunction with the width w of the camera image in pixels to compute the angle α corresponding to an x value (see Figure 7.1):

$$\alpha = \gamma \cdot \left(\frac{x}{w} - \frac{1}{2} \right)$$

That way, only the x component of the computed vanishing point would be used, thus eliminating the complicated trigonometric dependence on the pitch angle of the camera. The remaining dependence on the yaw angle of the camera would be linear.

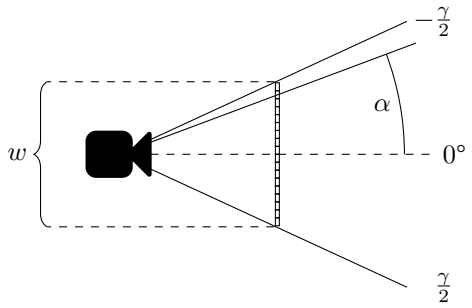


Figure 7.1: Computing the horizontal angle corresponding to a pixel. w is the image width in pixels and α is the horizontal field of view angle.

Bibliography

[1] Xavier Amatriain. "Beyond data: from user information to business value through personalized recommendations and consumer science." In: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. CIKM '13. San Francisco, California, USA: ACM, 2013, pp. 2201–2208. DOI: 10.1145/2505515.2514701.

[2] Steven Blank. *The Four Steps to the Epiphany. Successful Strategies for Products that Win*. 2nd ed. K&S Ranch, July 2013. ISBN: 978-0989200509.

[3] Jan Bosch. "Building Products as Innovation Experiment Systems." In: *Software Business: Third International Conference, ICSOB 2012, Cambridge, MA, USA, June 18-20, 2012. Proceedings*. Ed. by Michael A. Cusumano, Bala Iyer, and N. Venkatraman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 27–39. DOI: 10.1007/978-3-642-30746-1_3.

[4] Manfred Broy. "Challenges in Automotive Software Engineering." In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 33–42. DOI: 10.1145/1134285.1134292.

[5] John Canny. "A Computational Approach to Edge Detection." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (Nov. 1986), pp. 679–698. DOI: 10.1109/TPAMI.1986.4767851.

[6] Ulrik Eklund and Jan Bosch. "Architecture for Large-Scale Innovation Experiment Systems." In: *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*. Aug. 2012, pp. 244–248. DOI: 10.1109/WICSA-ECSA.212.38.

[7] Fabian Fagerholm, Alejandro Sanchez Guinea, Hanna Mäenpää, and Jürgen Münch. "The RIGHT Model for Continuous Experimentation." In: *The Journal of Systems and Software* 123 (2017), pp. 292–305. DOI: 10.1016/j.jss.2016.03.034.

[8] Brian Fitzgerald and Klaas-Jan Stol. "Continuous Software Engineering: A Roadmap and Agenda." In: *Journal of Systems and Software* 123 (2017), pp. 176–189. DOI: 10.1016/j.jss.2015.06.063.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN: 0-201-63361-2.

[10] Federico Giaimo, Hang Yin, Christian Berger, and Ivica Crnkovic. "Continuous Experimentation on Cyber-Physical Systems: Challenges and Opportunities." In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: ACM, 2016, 14:1–14:2. DOI: 10.1145/2962695.2962709.

[11] Helena Holmström Olsson and Jan Bosch. "Post-deployment Data Collection in Software-Intensive Embedded Products." In: *Software Business. From Physical Products to Software Services and Solutions: 4th International Conference, ICSOB 2013, Potsdam, Germany, June 11-14, 2013. Proceedings*. Ed. by Georg Herzwurm and Tiziana Margaria. Berlin, Heidelberg: Springer, 2013, pp. 79–89. DOI: 10.1007/978-3-642-39336-5_9.

[12] Ibtissam Karouach and Simeon Ivanov. *Lane Detection and Following Approach in Self-Driving Miniature Vehicles*. Bachelor thesis. Gothenburg, Sweden: Chalmers University of Technology, Department of Computer Science and Engineering, June 2016.

[13] Ron Kohavi, Thomas Crook, and Roger Longbotham. "Online Experimentation at Microsoft." In: *Data Mining Case Studies and Practice Prize III*. 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. Paris, France, 2009. URL: http://www.dataminingcasestudies.com/DMCS2009_ExP_DMCaseStudies.pdf.

[14] Ron Kohavi, Alex Deng, Brian Frasca, Roger Longbotham, Toby Walker, and Ya Xu. "Trustworthy Online Controlled Experiments: Five Puzzling Outcomes Explained." In: *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '12. Beijing, China: ACM, 2012, pp. 786–794. DOI: 10.1145/2339530.2339653.

[15] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. "Online Controlled Experiments at Large Scale." In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. Chicago, Illinois, USA: ACM, 2013, pp. 1168–1176. DOI: 10.1145/2487575.2488217.

[16] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M. Henne. "Controlled Experiments on the Web: Survey and Practical Guide." In: *Data Mining and Knowledge Discovery* 18.1 (Feb. 2009), pp. 140–181. DOI: 10.1007/s10618-008-0114-1.

[17] Ron Kohavi and Matt Round. "Front Line Internet Analytics at Amazon.com." 2004. URL: http://ai.stanford.edu/~ronnyk/emetricsAmazon.pdf.

[18] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997. ISBN: 0-792-39894-7.

[19] Eveliina Lindgren and Jürgen Münch. "Raising the Odds of Success: The Current State of Experimentation in Product Development." In: *Information and Software Technology* 77 (2016), pp. 80–91. DOI: 10.1016/j.infsof.2016.04.008.

[20] Jiri Matas, Charles Galambos, and Josef Kittler. “Robust Detection of Lines Using the Progressive Probabilistic Hough Transform.” In: *Comput. Vis. Image Underst.* 78.1 (Apr. 2000), pp. 119–137. DOI: 10.1006/cviu.1999.0831.

[21] Nicolas Navet and Françoise Simonot-Lion, eds. *Automotive Embedded Systems Handbook*. CRC Press, Dec. 2008. ISBN: 978-0-8493-8026-6.

[22] Claudio Pinello, Luca P. Carloni, and Alberto L. Sangiovanni-Vincentelli. “Fault-Tolerant Distributed Deployment of Embedded Control Software.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.5 (May 2008), pp. 906–919. DOI: 10.1109/TCAD.2008.917971.

[23] Eric Ries. *The Lean Startup. How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. 1st ed. Crown Business, Sept. 2011. ISBN: 978-0307887894.

[24] Olli Rissanen and Jürgen Münch. “Continuous Experimentation in the B2B Domain: A Case Study.” In: *IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*. May 2015, pp. 12–18. DOI: 10.1109/RCoSE.2015.10.

[25] Kasilingam Senthilkumar and Ramesh Ramadoss. “Designing Multicore ECU Architecture in Vehicle Networks Using AUTOSAR.” In: *2011 Third International Conference on Advanced Computing*. Dec. 2011, pp. 270–275. DOI: 10.1109/ICoAC.2011.6165187.

[26] Diane Tang, Ashish Agarwal, Deirdre O'Brien, and Mike Meyer. “Overlapping Experiment Infrastructure: More, Better, Faster Experimentation.” In: *Proceedings 16th Conference on Knowledge Discovery and Data Mining*. Washington, DC, 2010, pp. 17–26.

[27] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. SAE International, Sept. 2016.

[28] Henry Tucker. “The Car's the Technology.” In: *ITNOW* 56.1 (2014), pp. 10–12. DOI: 10.1093/itnow/bwu004.

[29] Sebastian Vöst and Stefan Wagner. *Towards Continuous Integration and Continuous Delivery in the Automotive Industry*. Dec. 2016. arXiv: 1612.04139.

[30] Shige Wang, Jeffrey R. Merrick, and Kang G. Shin. “Component Allocation with Multiple Resource Constraints for Large Embedded Real-Time Software Design.” In: *Proceedings. RTAS 2004. 10th IEEE Real-Time and Embedded Technology and Applications Symposium*. May 2004, pp. 219–226. DOI: 10.1109/RTTAS.2004.1317267.