Thesis for The Degree of Licentiate of Engineering

Introducing Continuous Experimentation on Resource-Constrained Cyber-Physical Systems

Federico Giaimo

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2018

Introducing Continuous Experimentation on Resource-Constrained Cyber-Physical Systems

Federico Giaimo

Copyright ©2018 Federico Giaimo
except where otherwise stated.
All rights reserved.

Technical Report No 183L
ISSN 1652-876X
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2018.

"There are three principal means of acquiring knowledge available to us: observation of nature, reflection, and experimentation. Observation collects facts; reflection combines them; experimentation verifies the result of that combination."

- Denis Diderot

Abstract

Software is ubiquitous and shapes our world, but at the same time it can be viewed as a plastic resource offering the possibility to be improved even after its deployment to better serve its purpose. Exploiting this possibility, the Continuous Experimentation practice is gaining momentum on connected software-intensive web-based systems, allowing the product owners to deploy "experiments" on their software systems, i.e., experimental instrumented versions of the software monitoring its performances with respect to a predefined set of target metrics, and to use this data to drive their products' evolution. Unfortunately the software that runs on physical units is not as easily re-deployed: cyber-physical systems, i.e., systems that interact with the physical world to perform their operations, may be in hard-to-reach places or moving in the environment, making the process difficult or energetically disadvantageous. Furthermore, such systems are often designed to have just enough hardware resources to perform their duties, having little computational resources left to perform additional tasks, such as performance monitoring.

This thesis explores the possibility to enable the Continuous Experimentation practice for distributed software running on resource-constrained cyber-physical systems on the example of self-driving vehicles, with the long-term goal of providing a way to continuously improve the quality of these systems' performances.

To achieve this, the included studies analyzed, proposed, and designed their contributions in order to provide suitable first steps for the adoption of this practice to the field which is still an open research question. Firstly, an analysis of the advantages and disadvantages that Continuous Experimentation could bring to the field was carried out. Then, key architectural characteristics capable to enable Continuous Experimentation on cyber-physical systems were identified. Successively, a more in-depth study was conducted to analyze how the Continuous Experimentation process could cope with the lack of adequate computational resources. Lastly, acknowledging the criticality of the software modules' intercommunication protocol, an analysis of the communication patterns highlighted how bandwidth-efficient alternatives can be developed using contextual knowledge.

The main results of this thesis are the key architectural features that allow the adoption of the Continuous Experimentation practice on resource-constrained cyber-physical systems.

Keywords
Continuous Experimentation; Cyber-Physical Systems; Software Engineering.

Acknowledgment

I wish to thank *in primis* my supervisors Christian Berger and Ivica Crnkovic for the generous guidance, support and advices they always offered during this journey. Without their help and understanding this goal may have not been achieved.

I am grateful also to my examiner Michel Chaudron and all the colleagues at the Software Engineering Division and Revere laboratory for their encouragement, constructive feedback and discussions that helped me grow as a researcher.

A special thanks to Martina and my friends Giuseppe, Lorenzo, Francesco and Angelo, who were always by my side in good and bad times. Last but not least, a great thank you to my family, for always believing in me.

This work has been supported by the COPPLAR Project - CampusShuttle cooperative perception and planning platform, funded by Vinnova FFI, Diarienr: 2015-04849.

List of Publications

Appended publications

This thesis is based on the following publications:

[A] Federico Giaimo, Hang Yin, Christian Berger, Ivica Crnkovic "Continuous Experimentation on Cyber-Physical Systems: Challenges and Opportunities"
Scientific Workshop Proceedings of XP2016 (XP '16 Workshops), Edinburgh, Scotland, United Kingdom, May 24-27, 2016.

[B] Federico Giaimo, Christian Berger "Design Criteria to Architect Continuous Experimentation for Self-Driving Vehicles"
Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, April 3-7, 2017.

[C] Federico Giaimo, Christian Berger, Crispin Kirchner "Considerations About Continuous Experimentation for Resource-Constrained Platforms in Self-driving Vehicles"
Software Architecture. ECSA 2017. Lecture Notes in Computer Science, vol 10475, Canterbury, United Kingdom, September 11-15, 2017.

[D] Federico Giaimo, Hugo Andrade, Christian Berger, Ivica Crnkovic "Improving Bandwidth Efficiency with Self-Adaptation for Data Marshalling on the Example of a Self-Driving Miniature Car"
Proceedings of the 2015 European Conference on Software Architecture Workshops (ECSAW '15), Dubrovnik, Cavtat, Croatia September 07-11, 2015.

Other publications

The following publications were published during my PhD studies, or are currently in submission/under revision. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

[a] Hugo Andrade, Federico Giaimo, Christian Berger, Ivica Crnkovic "Systematic evaluation of three data marshalling approaches for distributed software systems"
In Proceedings of the Workshop on Domain-Specific Modeling, pp. 71-76. ACM, 2015.

[b] Hang Yin, Federico Giaimo, Hugo Andrade, Christian Berger, Ivica Crnkovic "Adaptive Message Restructuring Using Model-Driven Engineering"
In Information Technology: New Generations, pp. 773-783. Springer, Cham, 2016.

Research Contribution

My contributions in literature are summarized as follow:

Paper A: In this work I contributed by investigating the state of the art of the Continuous Experimentation practice, as well as identifying some key challenges and opportunities that this practice may offer to cyber-physical systems.

Paper B: I led the research and was the main author of this work: my contributions lay in the formulation of the set of architectural properties that are the main findings of the work, plus all stages of the writing process.

Paper C: This paper extended the findings obtained in a master thesis that I co-supervised, my contributions relate to the additional work that led to the definition of the architectural features of interest and all stages of the writing process.

Paper D: All authors equally contributed to this paper. My personal contributions relate to the experimental phases of this work, i.e., development of the needed software, testing and data collection and analysis, tasks that were also reflected in the writing phase.

My contributions to the Grand Cooperative Driving Challenge and the CampusShuttle cooperative perception and planning platform (COPPLAR) project involve the integration of the high-level software with the CAN network interface provided by the vehicles' manufacturer. For the latter I also worked on the integration of the vehicle-to-vehicle and vehicle-to-infrastructure (V2X) interface and radar sensors to the software platform. The integration of the software on the actual vehicular platforms proved relevant not only for the context of my studies on Continuous Experimentation, which was the COPPLAR project, but also for the additional research projects running alongside my own that focus on safety and self-driving research.

Contents

Abstract **v**

Acknowledgement **vii**

List of Publications **ix**

Personal Contribution **xi**

1 Introduction **1**

1.1 Introduction 1
1.2 Motivation and Problem Domain 2
1.3 Research Goal 6
1.4 Research context 7
1.5 Methodologies and Research Results 9
1.6 Summaries of Studies 12
1.6.1 Paper A: Continuous Experimentation on Cyber-Physical Systems: Challenges and Opportunities 12
1.6.2 Paper B: Design Criteria to Architect Continuous Experimentation for Self-Driving Vehicles 13
1.6.3 Paper C: Considerations About Continuous Experimentation for Resource-Constrained Platforms in Self-driving Vehicles 13
1.6.4 Paper D: Improving Bandwidth Efficiency with Self-Adaptation for Data Marshalling on the Example of a Self-Driving Miniature Car 14
1.7 Discussion 14
1.8 Conclusion and Future Work 16

2 Paper A **19**

2.1 Introduction 20
2.2 Continuous Experimentation: State of the Art 20
2.3 Challenges and Opportunities for Cyber-Physical Systems . . . 21
2.4 Conclusions 22

3 Paper B **23**
3.1 Introduction . . . 24
3.1.1 Problem Domain & Motivation . . . 25
3.1.2 Research Goal . . . 26
3.1.3 Structure of the Document . . . 26
3.2 Related Work . . . 26
3.3 Functional and Non-Functional Properties . . . 28
3.4 Revere's Software Architecture and Development & Deployment Process . . . 30
3.4.1 Process . . . 30
3.4.2 Architecture . . . 32
3.5 Discussion and Threats to Validity . . . 32
3.5.1 Discussion . . . 32
3.5.2 Threats to Validity . . . 33
3.6 Conclusions and Future Work . . . 34

4 Paper C **37**
4.1 Introduction . . . 38
4.2 Related Work . . . 39
4.3 Assessing the Scarcity of Resources . . . 39
4.4 Software Architecture . . . 42
4.5 Discussion . . . 42
4.6 Conclusions and Future Work . . . 43

5 Paper D **45**
5.1 Introduction . . . 46
5.2 Related Work . . . 46
5.3 Self-Adaptive Marshalling . . . 48
5.4 Evaluation . . . 49
5.4.1 Evaluation Environment: Vehicle Simulation . . . 49
5.4.2 Evaluation Scenarios . . . 50
5.4.3 Data Collection . . . 51
5.5 Results . . . 51
5.5.1 Research Question 1 . . . 51
5.5.2 Research Question 2 . . . 52
5.6 Discussion . . . 53
5.6.1 Data Analysis & Interpretation . . . 53
5.6.2 Threats to validity . . . 53
5.7 Conclusion and Future Work . . . 54

Bibliography **55**

Chapter 1

Introduction

1.1 Introduction

According to the World Health Organization road accidents claim more than 1.25 million lives yearly [1] and are caused by human error around 90% of the time [2]. With the aim of mitigating these numbers, several academic and industrial parties are working and investing towards automated driving: as an example, 52 companies are registered for Autonomous Vehicle Testing Permits to the State of California's Department of Motor Vehicles by April 2018 [3]. To achieve this, the final goal and greatest challenge to be overcome is the guarantee of continued safety for driver, passengers and the other road users while the vehicle is maneuvering itself under all various driving conditions that can arise. This breaks down in several "smaller" co-acting technical challenges still to be solved, for example the complexity involved in sensing accurately the vehicles' surroundings, or the decision of which actions to take in complex or ambiguous situations.

Real-time reactivity is crucial for a vehicle approaching an obstacle as much as it is crucial that the same vehicle is able to create a consistent and complete representation of its surroundings in order to react in a correct and precise way. In computing, however, complex real-world models and short reaction times are two aspects often difficult to achieve at the same time on the same systems, given that the processing power is a finite and in many cases also a scarce resource. As such, the software powering the autonomous platforms require careful calibration in order to make the vehicle accurate in its decision-making process but also reactive enough to assess the unforeseeable situations it will face.

Moreover, due to the complexity of the context, the testing phase for automotive software already nowadays cannot truly guarantee to test it against all possible configurations, situations and road conditions that the vehicle will ever face during its operations. As a consequence there is a chance for defects in the software to slip through the testing phase and affect the production code and thus the users [4]: this happened in the "Toyota unintended acceleration case", where a software issue affecting vehicles sold between 2002 and 2009 resulted in some cases in unexpected and uncontrollable accelerations, causing hundreds of injuries and deaths among the vehicle occupants over the years [5]. Because

of the presence of this possibility, it is desirable that the vehicular software is based on an infrastructure that allows it to be improved and upgraded even after its deployment to the vehicle and delivery to the end user.

1.2 Motivation and Problem Domain

Cyber-physical systems are systems that integrate the physical world to a computational unit, or vice versa. They involve the sensing of or interaction with the system's surrounding environment in order to reach a goal.

In the automotive field such systems are widely used: a vehicle comprises and is operated by a network of interconnected cyber-physical systems called Electronic Control Units (ECUs) that sense the surroundings or the vehicle itself to provide a certain degree of automation and safety to drivers and road users, as it is shown in Fig. 1.1. A modern car can have more than 100 ECUs [6], collaborating to let the user easily and safely control the vehicle. Moreover, if the present-days race to achieve autonomous vehicles will be successful, the entire vehicle could be seen as a cyber-physical system comprising cyber-physical subsystems allowing people to interact with the world.

With such a high need for computation it is not surprising that software is a notable component of every vehicle. Already now the software in cars can approach 1GB in size, as shown by a Swedish automaker, which reached more than 900MB of software in their latest platform [6], excluding the speech and maps modules. This figure is bound to increase dramatically as the vehicles move towards more extensive automation functionality.

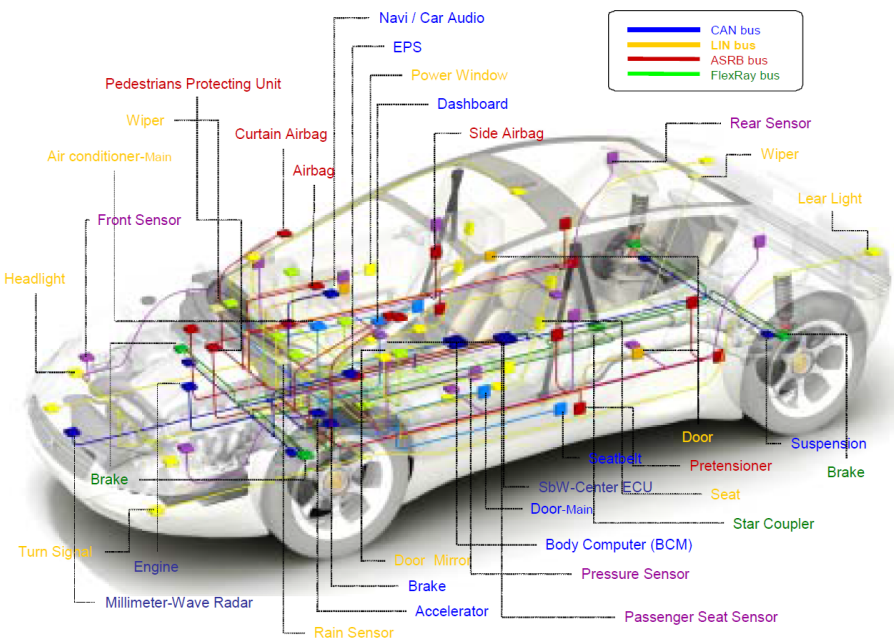


Figure 1.1: Example of a vehicular ECU network [7].

The vehicle's automation capabilities are often represented using the Autonomy Levels classification proposed by SAE International [8]. The SAE Levels are:

Level 0: No automation, where the driver is in charge of all aspects of the dynamic driving task.

Level 1: Driver Assistance, where the system aids the driver in performing either steering or acceleration/deceleration. The driver is expected to perform all remaining aspects of the dynamic driving task.

Level 2: Partial Automation, where the system aids the driver in both steering and acceleration/deceleration. The driver is expected to perform all remaining aspects of the dynamic driving task.

Level 3: Conditional Automation, where the systems performs all aspects of the dynamic driving task and the driver is expected to respond to an intervention request if necessary.

Level 4: High Automation, where in some driving modes the systems performs all aspects of the dynamic driving task even if the driver does not respond to an intervention request.

Level 5: Full Automation, where the full-time operation of the vehicle is done automatically, in all driving modes and conditions.

Nowadays there are manufacturers that offer semi-autonomous vehicles adhering to the SAE Level 2, i.e., vehicles that autonomously assist the driver controlling both steering and acceleration while still not bearing the full responsibility of monitoring the environment. Their rise in popularity due to the promise to increase safety for all road users has as a consequence that the role of software will only become more prominent.

While there has always been the need for thorough tests on the automotive software prior to its release, this will become even more true and necessary when the driving tasks will be automated. However, if testing software is in the general case a time-consuming and complex problem, it is even more so when taking into account the presence of the hardware that makes up a vehicle. Testing of automotive software is currently performed by using software test suites, Hardware-In-Loop simulations, and "Test farms" [9], but not all possible scenarios that a vehicle will face during its operational life can be recreated at testing time, nor can a vehicle be tested for an amount of time similar to its actual lifetime prior to be released in all possible scenarios. Using statistical calculations it derives in fact that an autonomous vehicle should be test-driven for 275 million miles without fatal accidents in order to assert with a statistical confidence level of 95% that said vehicle would cause less fatalities than humans did on average in the US in 2013. Reaching this result would require 100 of such autonomous vehicles to be tested non-stop for 12.5 years [10]. Due to the difficulty of the task, software errors can unknowingly be included into production code and reach the costumers. Two tragic yet emblematic cases took place close to each other in March 2018: during a test drive a self-driving vehicle made by Uber Technologies Inc. killed a pedestrian, tragically marking the world's first pedestrian death by an autonomous vehicle [11], and a car made by Tesla Inc. hit a highway lane divider killing its driver, while the "Autopilot" feature was active [12]. While the full extent of the legal consequences are yet to be seen, the immediate results of these incidents were the start of investigations and legal disputes between the authorities and the car makers, and the revoking

of the test permission given to Uber Technologies Inc. [11].

The traditional way to assess a vehicle defect is to issue costly vehicle recalls either to fix the issue at the workshop or to replace the malfunctioning part or vehicle altogether, and if the consequences of the defect affected the users the manufacturer may face legal investigations. With the advent and predominance of software in modern vehicle, the software-related issues can be resolved in a quicker way by installing the patched ECU software to replace the faulty one, but still requiring the customers to reach a workshop in order to perform the installation procedure. As Internet-connected vehicles increase in numbers, it can be expected that releasing Over-The-Air (OTA) software updates will be an increasingly adopted practice. The possibility of updating the vehicle's software after the vehicle has been handed to the customer has several advantages: firstly it allows to fix software issues without forcing the manufacturer to issue vehicle recalls, additionally it opens the possibility to deploy new features after the customer purchased the vehicle; as both these options can be performed remotely, this can prove a quicker and more user-friendly process than forcing the customers to go to a physical workshop to achieve the same result. One of the manufacturers that adopted this method to deploy software is Tesla Inc., which distributed its updates and automated driving features via wireless downloads over the course of years and to a fleet rapidly increasing in size [9]. Fig. 1.2 shows a chart of the number of Tesla cars delivered in the period between mid-2012 and mid-2016, the cumulative amount of miles driven and the software releases, provided increasingly often with the passing time. More miles driven means also more data that can be collected for testing or experimentation purposes, which could be one of the factors behind the increasing frequency of the software deliveries.

This example shows a trend in the automotive context towards increasing the delivery rate of new software releases. The final state of this trend would be continuous delivery of new functionality to the system, which has a contrasting relationship with testing time. If the goal is to rapidly deliver new functions then it would definitely be a benefit to shorten the time used for tests; on the other hand since the functionality and complexity of the system increases with every delivery, the total testing time is also increased. A possible way to mitigate this conflict is to adopt Continuous Experimentation [13], a practice based on the possibility to deliver software to already-deployed systems in order to validate it in its operational context before releasing it while the current "stable" software is still in use.

Continuous Experimentation is an Extreme Programming practice that involves the delivery and use of "experiments" as a mean for measuring improvements brought by different versions of a target software. Each of these experiments is a different instrumented version of the original software comprising slightly different features or configurations, capable of collecting relevant data to be retransmitted back to the product developer.

When a new hypothesis concerning the software is devised, the "experimenter" or "data scientist" designs an experiment to test it. This role has the task of creating the experiments and also of analyzing the collected data using statistical techniques. The results of this analysis will be used to allow the product owner to make informed decisions regarding which experiment performed well enough to be adopted globally according to one or more evaluation

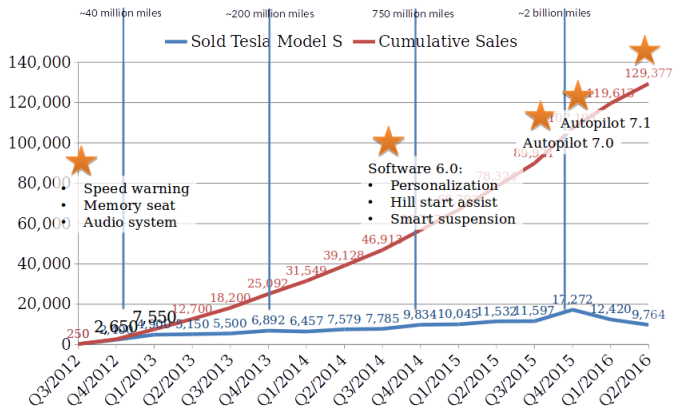


Figure 1.2: Per-quarter and cumulative sales of Tesla Inc.'s Model S, and corresponding cumulative distance driven (vertical lines) from mid-2012 to mid-2016. At the end of 2015 the company sold more than 100 thousands cars and accumulated around 2 billion driven miles. The stars represent software releases. Figure published in [14].

criteria. Following this pattern, the evolution of the product is influenced by steering the development process towards new functionality or features that have been validated by tests in the field.

The process of improving a software system using Continuous Experimentation is shown more in detail in Fig. 1.3:

Phase 1 : The user base available to experimentation is defined. The set of users affected by the overall experimentation phase is chosen in such a way that the results will be coherent and meaningful. Depending on the type of software and experiment the user base could be the users of a certain geographic area, or those operating the system during a certain period of the day, etc.;

Phase 2 : The user base is partitioned into non-overlapping clusters of users per experiment. In each cluster a different experiment will be deployed, with the exception of the control cluster, which will not receive any experiment. More than one experiment can be run in a cluster only if the different experiments do not interfere with each other [15];

Phase 3 : The experiments are alongside or in place of the official stable software and produce output in form of measurements, logs, etc. These results are finally collected and retransmitted to the experimenter to be analyzed;

Phase 4 : The collected results are analyzed and the best-performing experiment according to the objectives set is identified. The successful experiment is thus chosen and prepared for global integration into the system;

Phase 5 : The successful experiment is integrated as a new software feature or version, which can be deployed to the entire user base in form of an update. The overall system will then perform better than before.

The adoption of Continuous Experimentation is rising on software-intensive web-based systems like search engines, social media platforms, web applications, and web shops; in these cases relevant measurements can be parameters such as user retention or revenue per user [16]. The downside of Continuous Experimentation is that it requires an additional overhead on the back-end system, which has to keep track of the different software versions, the experiments that were run on them, and their results [13]. This may require the availability of additional processing power to perform these operations, which is usually easily available for web-based system, but not for cyber-physical systems.

Continuous Experimentation for cyber-physical systems, which includes the automotive context, has not been extensively explored neither in practice nor in literature so far, as it has been ascertained [17, 18]. The goal of this work is therefore to support the future adoption of Continuous Experimentation in the context of distributed software for cyber-physical systems and the overcome of the challenges that inevitably arise, like the lack of an adequate software infrastructure or the limitedness of computing resources.

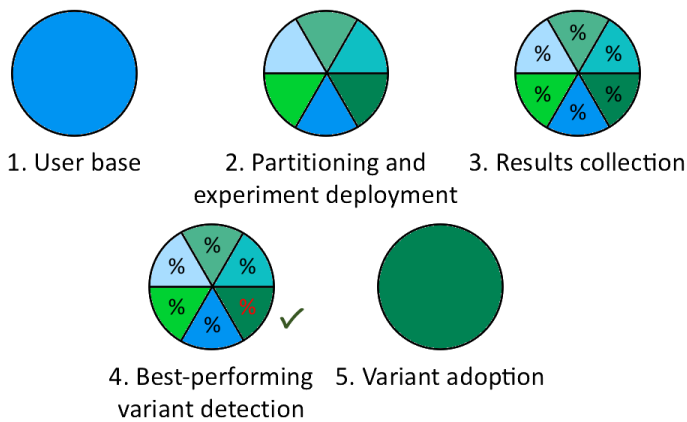


Figure 1.3: The phases of the Continuous Experimentation process.

1.3 Research Goal

The main aim that drove the work so far has been the adoption of Continuous Experimentation in the context of distributed software running on cyber-physical systems. Opposite to what happens in the field of web-based systems where Continuous Experimentation is increasingly adopted [17], different challenges complicate the creation of an experimentation setup: for example resources can be too limited, or there could be real-time or performance constraints that have to be abode, or even simply because the already-present software architecture is incapable of supporting it.

This aim can be summarized by the following Research Goal:

RG: To enable Continuous Experimentation on distributed software powering resource-constrained cyber-physical systems like autonomous vehicles.

The research goal can be further divided in the following research questions in order to explore more in depth the effects and scenarios that Continuous Experimentation would face in the field of cyber-physical systems:

$RQ1$: What advantages and additional challenges deriving from the adoption of Continuous Experimentation can be expected in the context of cyber-physical systems?

$RQ2$: What should the software architecture provide or abide to in order to enable a Continuous Experimentation process?

$RQ3$: What should the software development process allow in order to enable Continuous Experimentation on a system?

$RQ4$: What technical challenges are likely to emerge when Continuous Experimentation is applied to a resource-constrained system?

$RQ5$: What capabilities should the software architecture provide in a resource-constrained system to enable Continuous Experimentation?

$RQ6$: As the bandwidth in a distributed software system is a limited resource, although crucial for both the internal and external (OTA) data exchange, how can the communication among agents be made less resource-demanding?

In order to answer to these questions several studies have been undertaken. Paper A, summarized in Subsection 1.6.1, proposed a possible answer to RQ1; Paper B, summarized in Subsection 1.6.2, explored RQ2 and RQ3; Paper C, summarized in Subsection 1.6.3, faced RQ4 and RQ5; lastly Paper D, summarized in Subsection 1.6.4, attempted to answer RQ6.

It could be noted that aspects connected to safety constraints and experimentation on low-level software or firmware are not explored in the Research Questions. The reason behind this choice is discussed in Section 1.7 and could constitute a possible future direction for this work.

1.4 Research context

The work done so far was planned and performed in a research context that aimed at remain close to real-world settings. This included using software capable of managing complex distributed software and the use of facilities provided by Chalmers University of Technology's vehicle laboratory "Revere" (Resource for Vehicle Research) [19].

The Revere lab runs projects in collaboration with research and industrial partners, with the goal of pushing forward the research in the context of automated driving and collision avoidance.

The laboratory equipment comprises a SUV (Volvo XC90) used in the self-driving project COPPLAR – "CampusShuttle cooperative perception and

planning platform"[1] [20] and shown in Fig. 1.4b, a truck tractor (Volvo FH16) which participated in the Grand Cooperative Driving Challenge (GCDC) that took place in May 2016 in the Netherlands [21] and shown in Fig. 1.4a, an active-steered truck dolly, a number of miniature vehicles used for educational purposes shown in Fig. 1.4c, and recently an autonomous racing car. Such different platforms, although used in different projects, may need to cooperate or exchange data, and in order to minimize maintenance and refactoring efforts the software needs to facilitate future evolution. For this reason the two middleware software called OpenDaVINCI [22] and OpenDLV (abbreviation for Open DriverLess Vehicle) [23] are run in all these vehicles, providing a coherent and common foundation for the project-specific high-level software that runs on top.

Among the projects that run at Revere, I contributed mostly to the COPPLAR project and the Grand Cooperative Driving Challenge competition.

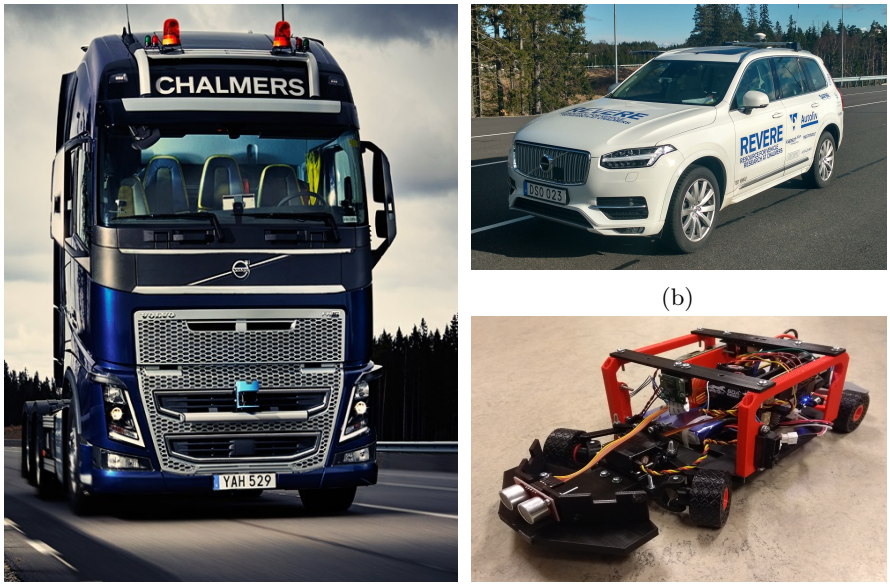
- **Grand Cooperative Driving Challenge (GCDC)** The GCDC competition took place in 2016 in the Netherlands, and it featured scenarios in which several autonomous vehicles would perform cooperative tasks, like vehicles communicating at a crossroad to avoid a crash, two vehicle platoons merging into one on a highway and platooning vehicles opening up their formation to give way to an emergency vehicle [24]. This project involved the truck tractor, which was equipped with four Ethernet-based cameras, eight ultrasonic sensors, a laser range finder, three inertial measurement units (IMUs), a GPS, and V2X capabilities in order to communicate with other vehicles and the outside world.

- **COPPLAR** The COPPLAR project instead focuses on a single vehicle, a Volvo XC90 SUV. The project aims at enabling the car to autonomously traverse the city of Gothenburg to connect the two campuses of Chalmers University of Technology. Since the envisioned route passes through the city center, the vehicle has to precisely and safely move in high-traffic and high-pedestrian density inner city roads. To do so, advanced sensing equipment is mounted on-board to provide the software with accurate data; it includes a 32-layers LIDAR scanner, a stereo vision camera, a GPS and IMU sensor, and vehicular radars.

- **OpenDaVINCI software environment** The software stack in use for our experiments and at the Revere vehicle laboratory is based on the middleware environment OpenDaVINCI. It provides uniform communication capabilities based on UDP multicast sessions for all the distributed software modules, simplifying the software "logic" needed to perform the system's goals, e.g. autonomously drive a vehicle. Other notable features are also the possibility to enforce real-time constraints using an rt-preempt enabled Linux kernel, and the deterministic and traceable scheduling of tasks.

  This middleware layer poses as basic layer of a software stack in which each additional level builds on the capabilities offered by the ones below it. Fig. 1.5 represents the case of the projects run at Revere, where all

[1] Funded by Vinnova FFI, Diarienr: 2015-04849.



(a) (b) (c)

Figure 1.4: Vehicles at the Revere laboratory: a Volvo FH16 truck tractor (a), a Volvo XC90 SUV (b), and an educational miniature vehicle (c).

software interfaces to the supported hardware components and devices like cameras, laser rangefinders, and GPS units are collected together on top of OpenDaVINCI in two software layers called OpenDLV.core and OpenDLV, additionally providing some reusable logic functions. Further on, project-dependent logic layers are posed on top of this "integration layers", so that the project-specific software can access the hardware of interest.

1.5 Methodologies and Research Results

The scope of this work is distributed software running on cyber-physical systems. This translates into a number of interconnected software modules potentially running on different hardware platforms that exchange data in order to achieve a goal, as shown in Fig. 1.6. Having in mind the objective of improving software quality or delivering more value to the customer, it may be needed to act on both the communication side and the software functionality side. This is because smarter agents have the potential of increasing the effectiveness of the entire system, and because a smoother communication can enable faster or more reliable data exchange among the software agents.

In order to improve the modules' functionality and obtain "smarter" agents, we selected Continuous Experimentation as method of choice due to its promising results in the field of web-based software-intensive systems [25]. Coming from the very different context of web engineering it was expected that the adoption of Continuous Experimentation on cyber-physical systems was not a straightforward process, so we explored relevant literature in 2016 in order to

understand how far it had been already applied in this field [18]. The literature search resulted in very few relevant articles, indicating that Continuous Experimentation was starting to be explored and studied in the context of web-based systems, but not yet in the context of cyber-physical systems. According to a recent study, the literature on the topic has not developed past the field of web-based systems [17].

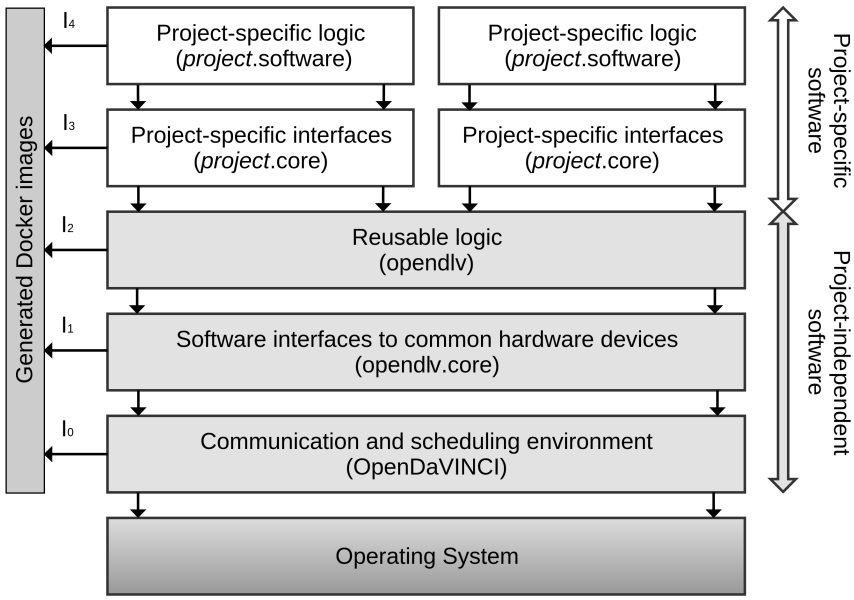


Figure 1.5: Software layers in use at the Revere laboratory. The final project-specific software is based on several underlying blocks that provide communication facilities, device interfaces or reusable logic modules. Figure published in [14].

As a first step after identifying this gap in the foundational literature of the field, we started by analyzing the possible effects of the adoption of Continuous Experimentation in this context, in order to run a feasibility/analysis study [26]. The resulting qualitative reflections [26] were compiled in Paper A and are illustrated as Contribution 1 (C1) in Fig. 1.7. This work provided us valuable feedback that encouraged us to continue exploring and studying in this direction.

Stemming from the discussion around Paper A and focusing on the software architecture, we conducted a more in-depth study to explore relevant and desirable aspects that we believe necessary for a cyber-physical system to successfully support Continuous Experimentation, resulting in Paper B. This design study [26] identified a set of criteria that relate not only to the features that are needed in the software architecture, but also to the software engineering process that has to be in place to produce the software itself. These criteria were both chosen from practical experience and extracted from relevant literature [26]. The two sets of criteria are the results referred to respectively as C2 and C3

in Fig. 1.7. This work also described the existing software infrastructure that is used in our University's automotive research laboratory Revere, where several automotive projects are conducted involving prototypes for self-driving vehicles.

After outlining the criteria deemed necessary to achieve Continuous Experimentation in the context of cyber-physical systems, we designed an exploration/method development [26] study to identify and assess some of the technical challenges that would arise when the experimentation process had to be run on systems with constraint computational resources, resulting in Paper C. The techniques [26] that we proposed as solutions rely on the presence in the software of certain capabilities that would allow to transparently manipulate the way the experimented software receives its input, in order to minimize the degradation of its performance while the experiments are run. The identified technical challenges and their possible solutions are labeled respectively as C4 and C5 in Fig. 1.7.

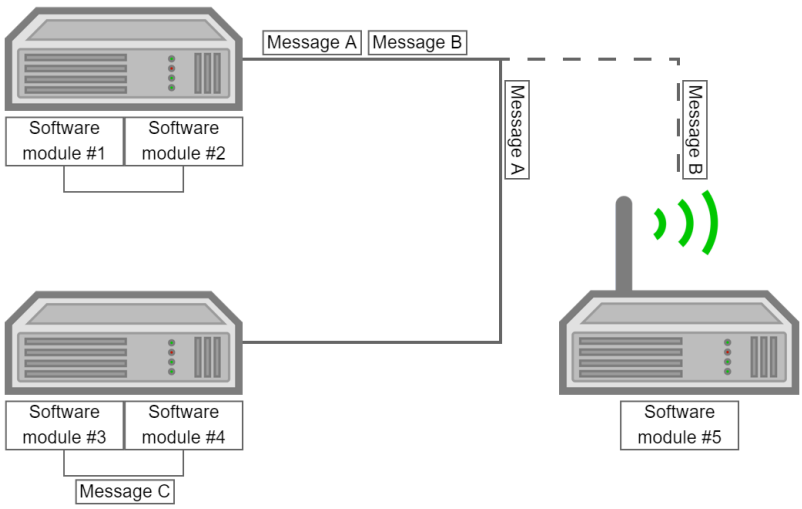


Figure 1.6: A distributed software system is comprised of several interconnected software modules that can be executed in one or more physical processing unit, communicating by exchanging predefined messages.

Connecting back to and focusing on the development of a "smoother" inter-module communication, in Paper D the adopted connection protocol was examined in order to understand whether it was possible to reduce the bandwidth consumption by creating more informative messages. The resulting reflections regarding the protocol's data redundancy were the starting point to design and implement a prototype [26] for a communication protocol that would make use of the contextual knowledge in the form of the common communication patterns to communicate more efficiently. This result, referred to as C6 in Fig. 1.7, has been validated by measuring the performance of the prototype on an example use case based on a realistic scenario.

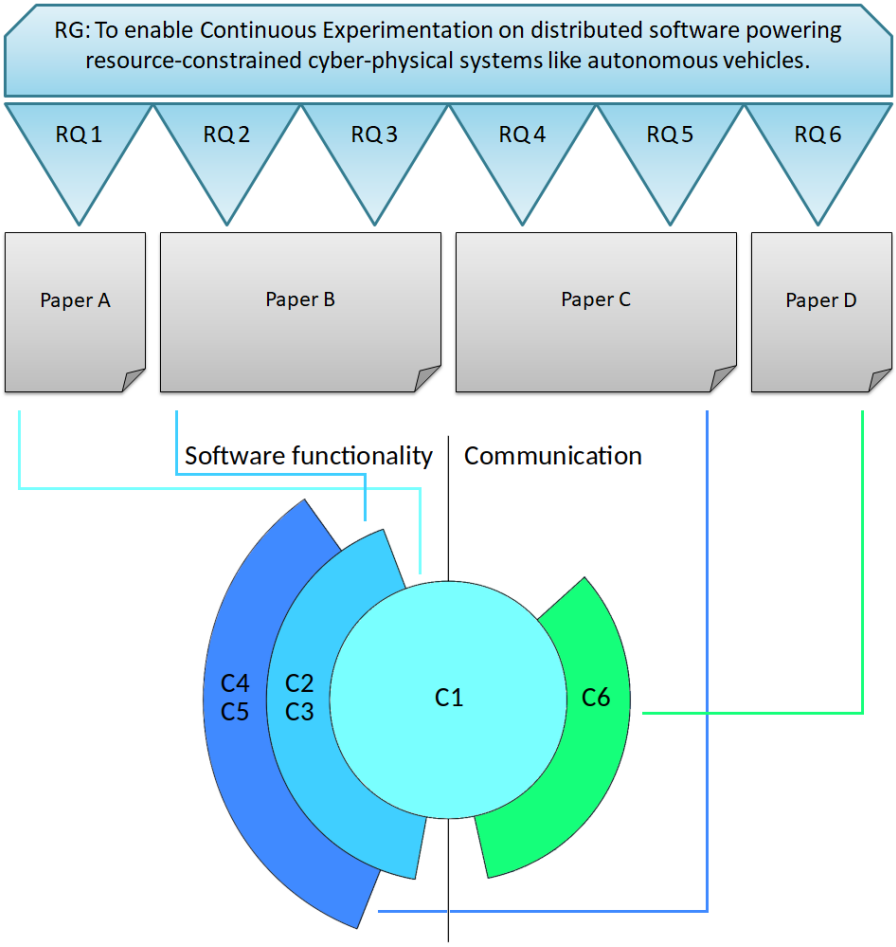


Figure 1.7: Overview of the Research Questions stemming from the overall Research Goal, the papers that assess them, and the relationship among their corresponding Contributions building upon each other.

1.6 Summaries of Studies

1.6.1 Paper A: Continuous Experimentation on Cyber-Physical Systems: Challenges and Opportunities

This position paper aims at assessing the feasibility and introducing the Continuous Experimentation practice to the field of cyber-physical systems, exploring the possible advantages and disadvantages that it can cause, seeking to find a possible answer to RQ1.

Continuous Experimentation is an Extreme Programming practice that introduces the concept of testing different versions of a software on different subsets of the user base in order to collect relevant feedback. The qualitative reflections it depicted show that while Continuous Experimentation is more

and more adopted in web-based systems to drive the software evolution process, there are several possible challenges that prevent its adoption in the context of cyber-physical systems. The possibility of improving the software using real-world data is however a valuable way to increase the quality of the software and provide more value to the customers, making it worth for both academia and industry to invest efforts to overcome the identified challenges.

This work acts as a foundation block for the contributions of the other appended papers, which develop and expand the discussion and studies around the practice and its adoption in the cyber-physical systems field.

1.6.2 Paper B: Design Criteria to Architect Continuous Experimentation for Self-Driving Vehicles

This design study explores the architectural properties that a software for cyber-physical systems, in the example of autonomous vehicles, should offer and fulfill in order to enable the Continuous Experimentation practice as a tool for quality improvement, thus exploring RQ2 and RQ3.

The proposed set of properties relate to both the software architecture and the software development process that needs to be in place to enable and facilitate Continuous Experimentation. These properties were chosen based on both the analysis of relevant literature of similar systems achieved in the past and the practical experience we gained in our automotive laboratory at Chalmers University of Technology, called Revere [19].

Finally, the work describes the software system that is used in the context of the aforementioned automotive laboratory, which abides to the identified conditions.

1.6.3 Paper C: Considerations About Continuous Experimentation for Resource-Constrained Platforms in Self-driving Vehicles

Since the final goal of these works is to enable the Continuous Experimentation practice to resource-constrained cyber-physical systems, this exploration/method development work aims at identifying and assessing the effects that the lack of resources has on the execution of this practice.

To cope with the limitedness of the computational resources, this study defines and proposes three “execution strategies” for Continuous Experimentation on resource-constrained systems: the Parallel, Serial, and Downsampled execution strategies. These strategies are devised for a system where the software is distributed as well as two important design criteria that the software architecture has to satisfy in order to enable them. Lastly, it proposes a new professional figure to the Continuous Experimentation model proposed by Fagerholm et al. [13], with the task of deciding which execution strategy is best suited to the envisioned experiment.

1.6.4 Paper D: Improving Bandwidth Efficiency with Self-Adaptation for Data Marshalling on the Example of a Self-Driving Miniature Car

In line with the aim of RQ1, the goals of this work were to analyze the data flow between software components in a cyber-physical system and to extract properties related to its application domain that would allow to improve the communication performance. This study was conducted in the context of distributed software for cyber-physical systems, more specifically using the software that powers miniature self-driving vehicles used in education and its simulation facilities. The advantage of using this platforms lies in the simplicity of prototyping and testing new software without going through all the necessary validation steps that are needed when a real vehicle is involved in the process. This in turn limits the transferability of the results but it is an effective way to obtain preliminary results to guide the subsequent steps of development.

A simulated environment, part of the OpenDaVINCI middleware, was used to simulate the track and the behavior of the vehicle in it. During the initial phase, the distributed software without any modification was used to have the car drive autonomously on different sections of the track, while the messages exchanged between the different software modules were recorded. The collected data were used to reflect upon the data structures exchanged during the test, and a certain degree of redundancy was expected and noticed. In order to reduce the amount of non-informative data exchanged among the software modules, a proof-of-concept marshalling algorithm was proposed with the purpose of comparing the data supposed to be transmitted with the data previously sent and transmitting the difference or "delta" of information. The advantage of it is that values with smaller absolute value, thus closer to zero, can be represented in data structures that are smaller in number of bytes than the original values. The savings in terms of transmitted and received bytes meant that the overall communication needed to consume less bandwidth for the same messages, reducing the communication overhead and leaving a bigger portion of this constraint resource for information carrying. The performance of the delta marshalling approach in terms of bytes exchanged was compared to the unchanged original marshalling approach and to the original marshalling approach with an added compression step using the publicly available *zlib* compression library, showing that the delta marshalling performed noticeably better than both alternatives thanks to the context-related knowledge used to define it.

1.7 Discussion

The goal of the work described in this thesis is to introduce, explore and evaluate criteria to facilitate the adoption of Continuous Experimentation for a distributed software in the cyber-physical systems field. Continuous Experimentation is a powerful and increasingly known tool in contexts like web-based systems, allowing to validate changes to the software in its operational scenarios and resulting in tangible improvements to the systems' performances. The main advantage of this field lies in the simplicity of obtaining the necessary resources that are needed to perform the additional tasks required by the

experimentation process, as web-based software is usually run on mainframes or server clusters, with the ability of spawning additional processes or virtual machines whenever the workload requires so. Due to the inherent differences between this world and cyber-physical systems, the same level of adoption is still a far away goal.

In order to achieve the same, the numerous challenges that arise need to be studied and possible solutions identified. An important aspect to consider in distributed software is the economy of the communication facility, meaning that the communication among software components should be effective in order to spare bandwidth; while this concept is always true and relevant, it acquires additional importance in light of the fact that the ordinary communication patterns have to involve large amount of data moving from one software module to another, and a practice like Continuous Experimentation can only add to it. For example, the system may need to transmit back raw data from sensors, data resulting from experimental software modules, additional diagnostics or performance measurement data; on the other hand it has to receive updates or new experimental software modules to be run when conditions arise. It becomes thus clear the need for good communication strategies and protocols to satisfy a growing demand for data from the deployed systems despite the limitation posed by the finite bandwidth.

Some degree of arbitration of the software in execution is necessary on the system, which among other choices has to decide whether to run the experimental software or not, depending on the fulfillment of application-dependent conditions. This leads to the main obstacle, which is the scarceness of resources like computational power and memory: while web server clusters can dynamically adjust to the computational load, a cyber-physical system is bound to its often limited hardware capabilities. This is particularly true in an economy of scale like the automotive industry, where the computational systems that are included in the vehicles are dimensioned to provide "just enough" computational power to perform their operations, leaving little room for additional tasks. If the resources are too scarce and an overhead is nonetheless introduced to the system, grave consequences can arise, like violating time or performance constraints because of a too high computational load. For this reason it is difficult to envision current commercial safety-critical cyber-physical systems in an experimentation context: in order to avoid unsafe behaviors the system should have a high enough surplus of resources to cover the computational overhead necessary to run an experiment while at the same time monitoring its performance, ensuring that all safety constraints are met. Additionally it should be able to stop the experimentation procedure and restore the original functionality if those safety thresholds are violated. However most cyber-physical systems are also resource-constrained systems, ruling out for the time being the possibility of safely experimenting if the service they provide is a critical one. Nonetheless, both industry and academia look more and more often in this direction, for the many technical advancements and possibilities that it can offer. The first steps that need to be addressed are, however, connected to the software stack that is deployed to the systems. The software and its architecture are in fact the enabling factors that can make it possible to overcome the challenges posed by the physical limitations of the hardware, e.g. the scarceness of computational resources. The development and extension

of suitable software stacks and frameworks capable of circumventing at least some of these limits should therefore be a priority if the goal of achieveing Continuous Experimentation on resource-constraint cyber-physical systems is to be reached.

It should be noted that the experiments which are taken into account in this work are not involving the low-level software or firmware, but refer to software running on systems that, although on limited resources, can guarantee certain facilities, e.g. instrumentation and monitoring of the software and a data connection to the product owner [14]. These capabilities, while not advanced *per se*, may still require abstraction layers usually provided by operating systems, which exclude at the moment the possibility to experiment on those cyber-physical systems powered by software that is not sophisticated enough.

While it is true that challenges still linger ahead of the road for the adoption of Continuous Experimentation on cyber-physical systems, it is worth noting that the interest shown by academia and industry in this practice is increasing in time thanks to the successes of the early adopters. Contributing to this growth is the widespread adoption of Continuous Experimentation by renown names of the web as a mean to improve their user experiences and revenue.

1.8 Conclusion and Future Work

The works presented in this thesis aimed at identifying possible ways to improve the quality of distributed software running on resource-constrained cyber-physical systems. As both inter-module communication and module-specific logic are crucial assets of such targets, both aspects have been considered and explored. The Continuous Experimentation practice was the tool of choice to seek improvements for the software modules. This practice is becoming more and more common in the field of software-intensive web-based systems, where it has been studied by academy and it is applied regularly by industrial parties, but several challenges are still deterring it from being adopted on distributed software running on resource-constrained cyber-physical systems. The contributions of the included papers aimed thus at extending the state-of-the-art in this sense, building upon each other in order to better define the unresolved challenges and to propose possible solutions. The first step was posing the theoretical basis for our chosen practice, Continuous Experimentation, to the field of cyber-physical systems identifying challenges and benefits of its possible application. Judging this field promising and still understudied, the following step was the identification of the required characteristics that a software architecture had to offer in order to enable the adoption of Continuous Experimentation on its system. The successive study pursued a similar goal but on a stricter scope, as it analyzed what limitations would a resource-constrained cyber-physical systems impose to a software under a Continuous Experimentation process and how to circumvent them with the help of practical features that the software could offer.

Regarding the communication aspect, an analytical study highlighted a possible way in which the communication patterns could be exploited to develop a protocol capable of condensing the exchanged information in smaller data structures, on the example of an educational miniature vehicle.

There are several future directions in which the work done so far can be expanded and refined. To seek further evaluation is certainly one, in the sense of seeking additional internal and external evaluation. For the former case, new measurements can be performed on additional implementations of the proposed concepts to evaluate in a more precise way their feasibility. Prototypical implementations of a Continuous Experimentation process are planned to be developed and tested on top of the OpenDaVINCI middleware, using the equipment provided by the Revere laboratory. For the latter case instead, additional feedback and experience could be collected in order to recognize and assess potential criticalities that have not been identified at earlier stages.

On this same direction goes our recent work based on a series of industry workshops in which professional figures from automotive companies answer a series of questions regarding the feasibility, usefulness, and risks of adopting Continuous Experimentation in the automotive field (the work is in its writing phase and it is planned to be submitted to the Journal of Systems and Software).