2nd International Workshop on Autonomous Systems Design

ASD 2020, March 13, 2020, Grenoble, France
converted to a virtual event due to COVID-19, held in April 2020

Edited by

Sebastian Steinhorst
Jyotirmoy V. Deshmukh

Editors

Sebastian Steinhorst
Technical University Munich, Germany
sebastian.steinhorst@tum.de

Jyotirmoy V. Deshmukh
University of Southern California, Los Angeles, CA, USA
jyotirmoy.deshmukh@usc.edu

ACM Classification 2012
Hardware → Analysis and design of emerging devices and systems; Computer systems organization → Robotic autonomy; Software and its engineering → Software safety; Computer systems organization → Dependable and fault-tolerant systems and networks

ISBN 978-3-95977-141-2

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-141-2.

Publication date
August, 2020

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/OASIcs.ASD.2020.0

ISBN 978-3-95977-141-2 **ISSN 1868-8969** **https://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/oasics

Contents

Preface
Sebastian Steinhorst and Jyotirmoy V. Deshmukh 0:vii

Towards a Reliable and Context-Based System Architecture for Autonomous Vehicles
Tobias Kain, Philipp Mundhenk, Julian-Steffen Müller, Hans Tompits, Maximilian Wesche, and Hendrik Decke 1:1–1:7

Fusion: A Safe and Secure Software Platform for Autonomous Driving
Philipp Mundhenk, Enrique Parodi, and Roland Schabenberger 2:1–2:6

Adaptable Demonstrator Platform for the Simulation of Distributed Agent-Based Automotive Systems
Philipp Weiss, Sebastian Nagel, Andreas Weichslgartner, and Sebastian Steinhorst 3:1–3:6

Agile Requirement Engineering for a Cloud System for Automated and Networked Vehicles
Armin Mokhtarian, Alexandru Kampmann, Bassam Alrifaee, Stefan Kowalewski, Bastian Lampe, and Lutz Eckstein 4:1–4:8

BreachFlows: Simulation-Based Design with Formal Requirements for Industrial CPS
Alexandre Donzé 5:1–5:5

2nd International Workshop on Autonomous Systems Design (ASD 2020).
Editors: Sebastian Steinhorst and Jyotirmoy V. Deshmukh
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the proceedings of the 2nd International Workshop on Autonomous Systems Design (ASD 2020). The workshop was originally planned to be held in Grenoble, France on March 13, 2020, and is co-located with the 23rd Design, Automation and Test in Europe Conference (DATE 2020). However, due to the global COVID-19 pandemic, the workshop was held as a virtual event along the virtual DATE 2020 conference.

In 2020, for the first time, we introduce the DATE initiative on Autonomous Systems Design, a two-day special event at DATE, which is the leading European conference on embedded hardware and software design. It focuses on recent trends and emerging challenges in the field of autonomous systems. Such systems are becoming integral parts of many Internet of Things (IoT) and Cyber-Physical Systems (CPS) applications. Automated driving constitutes today one of the best examples of this trend, in addition to other application domains such as avionics and robotics. ASD is organized as a Thursday Initiative day and Friday Workshop day to constitute a two-day continuous program covering different industrial and academic methods and methodologies in the design, verification and validation of autonomous systems.

The workshop for which the proceedings at hand are published was organized into sessions with peer-reviewed research and demo papers selected from an open call, complemented by invited talks and distinguished keynotes.

Five selected papers are included in this volume, complementing four talks and one demo. The papers included in this volume discuss recent development approaches for autonomous systems, presenting two perspectives on advanced automotive software and system platforms for autonomous driving, as well as a development and simulation platform for agent-based automotive architectures. Another contribution is targeting a cloud system perspective for automated and networked vehicles and an extended abstract introduces an approach to systematic simulation-based testing for CPS, considering formal requirements.

Support and Acknowledgement

We would like to thank all speakers for their valuable support of the workshop despite the challenging situation. We would also like to acknowledge the contributions of authors and the help of program committee members in the review process. A big thank you goes to Ege Korkan for creating and managing the workshop website. We extend our appreciation to the DATE organizing committee and K.I.T Group GmbH Dresden for their help and support in the logistics arrangements. ASD 2020 is partially supported by Autonomous Intelligent Driving GmbH, we would therefore like to acknowledge their financial contribution. Lastly, we would like to thank the editorial board of the OASICs proceedings.

Organizers

Workshop Organizers

Sebastian Steinhorst, Technical University of Munich, Germany
Jyotirmoy Vinay Deshmukh, University of Southern California, USA

Steering Committee

Rolf Ernst, TU Braunschweig, Germany
Selma Saidi, TU Dortmund, Germany
Dirk Ziegenbein, Robert Bosch GmbH, Germany

Technical Program Committee

Bart Besselink, University of Groningen, Netherlands
Paolo Burgio, Università degli Studi di Modena e Reggio Emilia, Italy
Jyotirmoy Vinay Deshmukh, University of Southern California, USA
Mohammad Hamad, Technical University of Munich, Germany
Arne Hamann, Robert Bosch GmbH, Germany
Xiaoqing Jin, University of California Riverside, USA
Xue Lin, Northeastern University, USA
Martina Maggio, Saarland University, Germany
Philipp Mundhenk, Autonomous Intelligent Driving, Germany
Saravanan Ramanathan, TUMCREATE, Singapore
Selma Saidi, TU Dortmund, Germany
Shreejith Shanker, Trinity College Dublin, Ireland
Sebastian Steinhorst, Technical University of Munich, Germany
Andrei Terechko, NXP Semiconductors, Netherlands

2nd International Workshop on Autonomous Systems Design (ASD 2020).
Editors: Sebastian Steinhorst and Jyotirmoy V. Deshmukh

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Towards a Reliable and Context-Based System Architecture for Autonomous Vehicles

Tobias Kain
Volkswagen AG, Wolfsburg, Germany
tobias.kain@volkswagen.de

Philipp Mundhenk
Autonomous Intelligent Driving GmbH, München, Germany
philipp.mundhenk@aid-driving.eu

Julian-Steffen Müller
Volkswagen AG, Wolfsburg, Germany
julian-steffen.mueller@volkswagen.de

Hans Tompits
Technische Universität Wien, Austria
tompits@kr.tuwien.ac.at

Maximilian Wesche
Volkswagen AG, Wolfsburg, Germany
maximilian.wesche@volkswagen.de

Hendrik Decke
Volkswagen AG, Wolfsburg, Germany
hendrik.decke@volkswagen.de

Abstract

Full vehicle autonomy excludes a takeover by passengers in case a safety-critical application fails. Therefore, the system responsible for operating the autonomous vehicle has to detect and handle failures autonomously. Moreover, this system has to ensure the safety of the passengers, as well as the safety of other road users at any given time. Especially in the initial phase of autonomous vehicles, building up consumer confidence is essential. Therefore, in this regard, handling all failures by simply performing an emergency stop is not desirable. In this paper, we introduce an approach enabling a dynamic and safe reconfiguration of the autonomous driving system to handle occurring hardware and software failures. Since the requirements concerning safe reconfiguration actions are significantly affected by the current context the car is experiencing, the developed reconfiguration approach is sensitive to context changes. Our approach defines three interconnected layers, which are distinguished by their level of awareness. The top layer, referred to as the *context layer*, is responsible for observing the context. These context observations, in turn, imply a set of requirements, which constitute the input for the *reconfiguration layer*. The latter layer is required to determine reconfiguration actions, which are then executed by the *architecture layer*.

2012 ACM Subject Classification Computer systems organization → Reconfigurable computing

Keywords and phrases autonomous driving, fail-operational systems, context-based architecture, application placement, optimization, monitoring

Digital Object Identifier 10.4230/OASIcs.ASD.2020.1

1 Introduction

Nowadays, vehicles are equipped with various advanced driver assistance systems that support the driver while operating the vehicle. Actions that modern vehicles are capable of doing are, for instance, keeping the distance to a preceding vehicle, autonomous parking, or switching lanes on highways. Although these functions are highly reliable and well tested, the driver is still constrained to monitor their behavior and take over control, if required [3].

As far as fully autonomous vehicles are concerned, such takeover actions by passengers are excluded [13]. Therefore, to guarantee the safety of the passengers and other road users in case an occurring failure causes a safety-critical driving application to misbehave, the system responsible for operating the car has to be designed in a *fail-operational manner*, i.e., the system has to handle hardware and software failures autonomously.

In this paper, we present an approach capable of quickly recovering a safe system state after an occurrence of a hardware or software failure so that the driving mission can be continued. Since various parameters of a system configuration depend on the context the vehicle is currently experiencing, our reconfiguration approach is based on system optimization actions which adjust the system according to the context at hand. Among the different dimensions which are taken into account, our context-aware reconfiguration approach allows to dynamically adjust the safety requirements to the present situation, enabling an increased safety of the system. In case an occurrence of a failure causes the system safety level to drop below a certain threshold, our approach performs an emergency stop.

The context-based reconfiguration feature of our method is based on a layered architecture, defining three interconnected layers, which are distinguished by their level of awareness: The top layer, referred to as the *context layer*, extracts context information from the given input. The output of the context layer is then in turn used as the input for the layer responsible for determining the configuration, called the *reconfiguration layer*. Finally, the application placement, i.e., the assignment of application instances with computing nodes, is then taken care of by the *architecture layer*, which also implements means to monitor the system state.

The paper is organized as follows: Section 2 introduces our general approach for a reliable context-based system architecture for use in autonomous vehicles. Section 3 discusses the methods used for extracting and representing the context. Section 4 illustrates the characteristics and challenges of a context-based reconfiguration. Section 5 gives an overview of the challenges involved in applying new configuration and monitoring system changes. The paper concludes in Section 6 with a discussion on related approaches and future work.

2 The General Approach

Figure 1 shows the general framework of our approach for a reliable context-based system architecture, which defines three interconnected logical layers, whereby each layer comprises a set of interrelated tasks, providing distinct levels of awareness, viz. *context awareness*, *safety awareness*, and *self awareness*.

The top layer, dealing with the first type of awareness, is accordingly referred to as the *context layer*. This layer determines the current context the vehicle is in and extracts requirements affecting the actions of lower layers. Mission goals, like the target destination or the level of entertainment requested by the driver, or environment information, like the current weather situation or road and traffic conditions, are examples for parameters influencing action decisions.

The requirements determined by the context layer are used as input for the *reconfiguration layer*. This layer evaluates the received requirements and plans further actions considering the current context. These actions include, for example,

- selecting a set of applications,
- determining their redundancy and hardware segregation requirements,
- computing valid reconfiguration actions, as well as
- optimizing the entire system architecture.

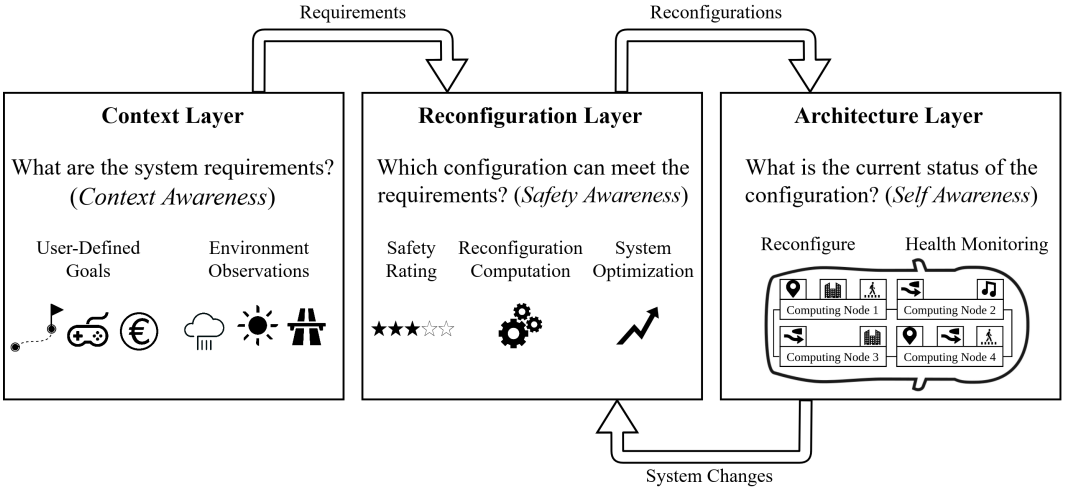


Figure 1 The three logical layers used in our approach and their relationships. The layers provide distinct levels of awareness.

The reconfiguration measures determined by the reconfiguration layer are then executed by the *architecture layer*. This layer is responsible for distributing application instances among the available computing nodes as instructed by the layer situated above, whereby a minimum level of safety has to be preserved. Furthermore, this layer also implements monitor mechanisms that control the health of the hardware and software components the car is equipped with. In case a system change is observed, the reconfiguration layer is informed such that a new configuration is determined.

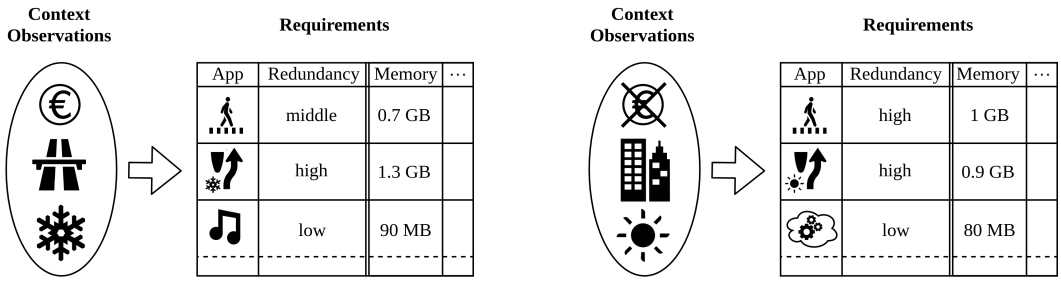
3 Context Extraction

Adjusting the system configuration according to the current context first requires the extraction of context observations from environmental parameters. These observations then imply a set of requirements, which are used as input for the subsequent reconfiguration actions.

Figure 2 illustrates two use cases that show that distinct sets of context observations imply distinct sets of requirements.

In the first use case, depicted in Figure 2a, the passenger of an autonomous taxi booked a premium ride. Furthermore, we assume that the vehicle is currently driving in snowy weather on a highway. From these context observations, a set of required software applications can be implied. This set may, for example, include applications for detecting pedestrians, planning trajectories taking the rough weather conditions into account, as well as entertainment applications that are included in the ride due to a booked premium package. Moreover, from the context observations, we can imply the safety-criticality of the respective applications and thus the required level of redundancy, as well as other performance parameters.

The use case illustrated in Figure 2b, on the other hand, assumes a low-budget ride in an urban environment under good weather conditions. Consequently, the set of required applications include, for example, a pedestrian detection module. The demanded level of redundancy of this application is high, as this application is considered safety-critical in the current context since many pedestrians are present in urban environments. Note that in the first use case, the required level of redundancy of the same application is lower since



(a) A premium ride on a highway in wintry conditions. These context observations imply, for example, that the set of required applications include a passenger detection, a trajectory planner which takes the rough weather conditions into account, and an entertainment application.

(b) A low-budget ride in an urban environment under good weather conditions. These context observations imply, for example, that the set of required applications include a passenger detection, a trajectory planner which takes the good weather conditions into account, and an application for computing tasks received from cloud services.

Figure 2 Two use cases showing the correlation between context observations and requirements. The two distinct scenarios imply a distinct set of requirements.

on highways, encountering pedestrians is unlikely. For the pedestrian detection module, a medium level of redundancy can be, for example, satisfied in case one redundant module is executed. On the other hand, the level of redundancy can be considered high if two redundant instances of this module are executed.

The discussed use cases illustrate the two main challenges of the context layer: Extracting context observations and implying requirements.

The former task, extracting context observations, necessitates perceiving environment parameters. These parameters are, for example, determined by sensors the car is equipped with, communicating with backend services, and interacting with the passengers.

The second task, implying requirements from context observations, requires methods for specifying implication rules. Therefore, the system architecture designers, as well as the application developers, have to define requirements for different contexts. A conceivable approach for representing such rules is employing *answer-set programming* [2], a declarative problem-solving approach based on logic programming for which sophisticated solver technology exists [7, 4].

4 Context-Based Reconfiguration

The task of determining a context-based reconfiguration, i.e., a mapping between application instances and computing nodes that respects the prevailing context, is not trivial since the placement decisions depend on various parameters. We refer to this problem as the *application placement problem*.

This problem is not only limited to our problem setting, but the placement of applications on computing nodes is indeed a well-studied topic in other fields too. In particular, research on cloud and edge computing has addressed this problem, like, e.g., approaches for optimizing properties like energy consumption [8], network traffic load [9], and resource utilization [5] have been discussed in the literature.

Generally speaking, the input of the application placement problem is a set of application instances and a set of resources, like, e.g., computing nodes, operating systems, or communication links. Furthermore, we define for each application instance and each resource,

a set of parameters including, for example, performance parameters such as the minimum required memory and CPU demand, as well as safety parameters like the minimum required level of redundancy and hardware segregation. These parameters have to be specified by the system architecture designers and the application developers. The output of the application placement problem is an assignment that maps each instance to exactly one node.

In order to restrict the number of valid assignments, constraints based on the specified parameters can be defined. Depending on the specified constraints, either none, one, or multiple valid assignments exist. In case that there are different solutions, an optimization function can be defined that specifies which assignments are the most desired.

This optimization function also depends on the current context. Therefore, an approach allowing a context-based update of the optimization goal leads to configurations that are well adjusted to the current situation.

For solving the application placement problem, various optimization approaches are applicable. The options range from integer linear programming and evolutionary game theory [12] to reinforcement learning approaches [1].

5 Architecture Interaction

The architecture layer of our approach comprises the tasks responsible for interacting with the architecture, i.e., the application instances and computing nodes.

One main task of this layer is to apply the reconfiguration actions determined by the reconfiguration layer. The challenge thereby is to ensure a fast, safe, and organized configuration roll-out. Furthermore, it has always to be guaranteed that the reconfiguration actions do not decrease the level of safety. Therefore, safety-validation operations have to be executed prior to the configuration roll-out.

Besides applying reconfiguration actions, also monitoring the state of the computing nodes and the executed application instances is an important task.

Self-awareness requires monitoring the status of the system to maintain an operational state. Monitoring the system, in turn, depends in general on the observation of several level-specific data. Concerning safety, different levels may define different requirements for a minimum operational capability.

Since full vehicle autonomy excludes human takeover actions, classical failure tolerance is not sufficient as errors may have various causes and interference effects. Failure handling requires knowledge of cross-layer dependencies. Thus, system monitoring and self-awareness are cross-layer problems [14].

In case a failure is detected, the reconfiguration layer is notified to reconfigure the system to obtain a safe system sate. If safety-critical applications are affected by the failure, the reconfiguration layer has to ensure that lost functionality is recovered within a short time. An approach addressing this challenge, called FDIRO, standing for "fault detection, isolation, recovery, and optimization", has been introduced in a recent paper [6], adopted from a similar method from the aerospace domain [16].

6 Conclusion

In this paper, we introduced a three-layered approach towards implementing a reliable and context-based system architecture in autonomous vehicles. By employing this approach, we anticipate an increase in safety, enabled by a fast and context-oriented reconfiguration in case a hardware or software failure is detected.

To the best of our knowledge, the introduced safety and context-aware configuration approach for autonomous vehicles is novel. However, in the past, efforts in the automotive research field focused on developing concepts for context-aware advanced driver assistance systems [15, 11]. Context-awareness of applications is also pursued in other research fields [10].

Since the advance of autonomous vehicles is imminent, further work concerning each layer of our approach for context-based system architecture is necessary. In our future research activities, we plan to implement a simulator to show the feasibility of our proposed architecture.

References

1 Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016. `arXiv:1611.09940`.

2 Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

3 Karel A Brookhuis, Dick de Waard, and Wiel H. Janssen. Behavioural impacts of advanced driver assistance systems–An overview. *European Journal of Transport and Infrastructure Research*, 1(3), 2019.

4 Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. clasp: A conflict-driven answer set solver. In *Proceedings of 9th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2007*), volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

5 Fatma Ben Jemaa, Guy Pujolle, and Michel Pariente. QoS-aware VNF placement optimization in edge-central carrier cloud architecture. In *Proceedings of the 2016 IEEE Global Communications Conference* (*GLOBECOM 2016*), pages 1–7, 2016.

6 Tobias Kain, Hans Tompits, Julian-Steffen Müller, Philipp Mundhenk, Maximilan Wesche, and Hendrik Decke. Fdiro: A general approach for a fail-operational system design, 2020. Submitted draft. Abstract accepted for presentation at *30th European Safety and Reliability Conference* (*ESREL 2020*).

7 Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

8 Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. EnaCloud: An energy-saving application live placement approach for cloud computing environments. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing* (*CLOUD-II 2009*), pages 17–24, 2009.

9 Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *Proceedings of the 2010 IEEE International Conference on Computer Communications* (*INFOCOM 2010*), pages 1–9, 2010.

10 Marco Mori, Fei Li, Christoph Dorn, Paola Inverardi, and Schahram Dustdar. Leveraging state-based user preferences in context-aware reconfigurations for self-adaptive systems. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods* (*SEFM 2011*), volume 7041 of *Lecture Notes in Computer Science*, pages 286–301. Springer, 2011.

11 Andry Rakotonirainy. Design of context-aware systems for vehicles using complex system paradigms. In *Proceedings of the CONTEXT 2005 Workshop on Safety and Context*, volume 158 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

12 Yi Ren, Junichi Suzuki, Athanasios Vasilakos, Shingo Omura, and Katsuya Oba. Cielo: An evolutionary game theoretic framework for virtual machine placement in clouds. In *Proceedings of 2014 International Conference on Future Internet of Things and Cloud* (*FiCloud 2014*), pages 1–8, 2014.

13 SAE International. Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems. *SAE Standard J3016*, 2014.

14 Johannes Schlatow, Mischa Möstl, Rolf Ernst, Marcus Nolte, Inga Jatzkowski, Markus Maurer, Christian Herber, and Andreas Herkersdorf. Self-awareness in autonomous automotive systems. In *Proceedings of the 20th Conference & Exhibition on Design, Automation & Test in Europe* (*DATE 2017*), pages 1050–1055. European Design and Automation Association, 2017.

15 Gereon Weiss, Florian Grigoleit, and Peter Struss. Context modeling for dynamic configuration of automotive functions. In *Proceedings of the 16th International IEEE Conference on Intelligent Transportation Systems* (*ITSC 2013*), pages 839–844, 2013.

16 Ali Zolghadri. Advanced model-based FDIR techniques for aerospace systems: Today challenges and opportunities. *Progress in Aerospace Sciences*, 53:18–29, 2012.

Fusion: A Safe and Secure Software Platform for Autonomous Driving

Philipp Mundhenk
Autonomous Intelligent Driving GmbH, München, Germany
philipp.mundhenk@aid-driving.eu

Enrique Parodi
Autonomous Intelligent Driving GmbH, München, Germany
enrique.parodi@aid-driving.eu

Roland Schabenberger
Autonomous Intelligent Driving GmbH, München, Germany
roland.schabenberger@aid-driving.eu

Abstract

The vastly increasing amount of software in vehicles, its variability and complexity, as well as the computational requirements, especially for those built with autonomous driving in mind, require new approaches to the structure and integration of software. The traditional approaches of single-purpose embedded devices with integrated software are no longer a suitable choice. New architectures introduce general purpose compute devices, capable of high-performance computation, as well as high variability of software. Managing the increasing complexity, also at runtime, in a safe and secure manner, are open challenges. Solving these challenges is a high-complexity development and integration effort requiring design-time and runtime configuration, approaches to communication middleware, operating system configuration, such as task scheduling, monitoring, tight integration of security and safety, and, especially in the case of autonomous driving, concepts for dynamic adaption of the system to the situation, e.g., fail-operational concepts. We present Fusion, a next-generation software platform supporting the development of autonomous driving systems.

2012 ACM Subject Classification Computer systems organization → Embedded software

Keywords and phrases middleware, software platform, autonomous driving

Digital Object Identifier 10.4230/OASIcs.ASD.2020.2

1 Introduction

Traditionally, automotive software has been aligned with hardware, also known Electronic Control Units (ECUs). All communication is performed via signals, either between software components (e.g., AUTOSAR Classic), or between ECUs [2]. In more modern software architectures, especially in use for more complex systems such as vehicles with driver assistance systems, service-oriented communication is introduced (e.g., AUTOSAR Adaptive) [1]. This is supported by new means of communication, among other changes, introduced in the automotive domain, such as Automotive Ethernet[4]. These modern frameworks are typically built on top of real-time capable POSIX operating systems, using operating system processes as the atomic building blocks for applications. Communication is defined and performed between processes and devices. Inside the processes, there is limited to no support for the developer in terms of coordination (multi-threading), introspection, and communication between individual parts of the software.

Software stacks for autonomous driving are significantly larger and more complex than all existing software in vehicles, with a trend to grow larger and ever more complex over time. An example of the architecture of such a stack, including its internal and external interfaces is shown in 1. For developers to be able to focus on advancing functionality, some

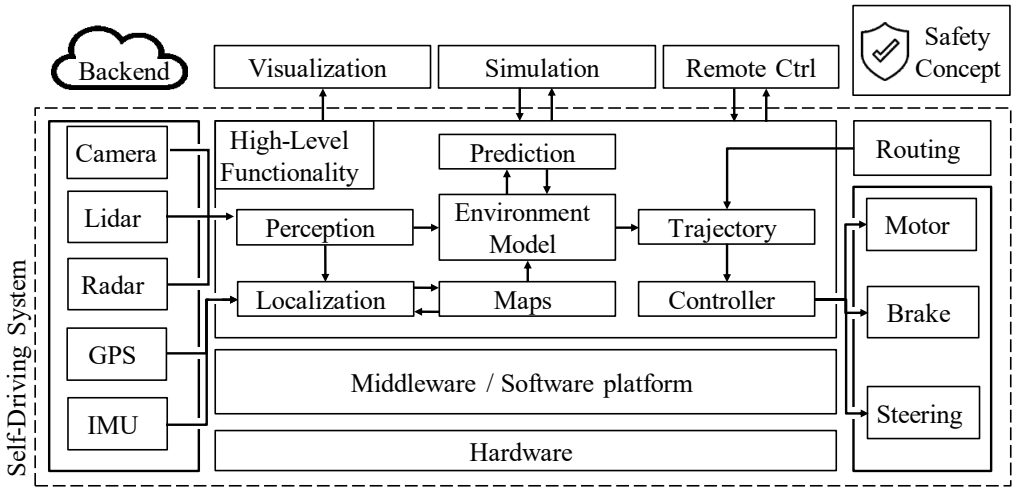


Figure 1 High-level view of a self-driving system and its interfaces.

support is required. This is especially relevant for performance reasons. Autonomous driving software requires a large amount of performance and the hardware needs to be utilized to the maximum possible degree. Additionally, when running an autonomous vehicle without supervision, the correctness of the software is fundamental.

2 Fusion Software Platform

In this paper, we propose the Fusion Software Platform to combat the complexity and performance requirements of autonomous driving software. As shown in Figure 2, each instance of Fusion is running inside a single operating system process. Through the use of a dedicated Operating System Abstraction Layer (OSAL), support for different operating systems is possible and has been implemented in the past. Fusion instances can communicate with each other, across processes and across devices, such as processors or hypervisor partitions, by the help of a pluggable transport mechanism (e.g., using Data Distribution Service (DDS)). All communication means are abstracted by Fusion, so that all application software is independent of the used transport mechanism.

Internally, Fusion improves upon existing software platforms by using a microkernel approach, and combining the benefits of service-oriented and component-based design. The atomic building block is an *Object*. Fusion objects contain properties, methods and events, defined in a dedicated *Object Specification Language (OSL)*. Additionally, custom data structures can be defined. Properties allow the storage of data, whereas events and methods are used to connect to other objects with (method) or without (method) return path. Changes of properties can automatically result in events being fired. Timers are objects, which can emit events at configurable times. OSL is generated into the target code (e.g., C++) to be used by developers when building their functionality (see Figure 3). Developers can implement the functionality, and for this use the full set of features in their target language (e.g., C++) if they wish to do so.

In order to support concurrency, all objects are assigned to execution contexts. Thus, developers are not required to use synchronization primitives, such as mutexes, condition variables, etc. Each execution context consists of a queue and (typically) one or multiple

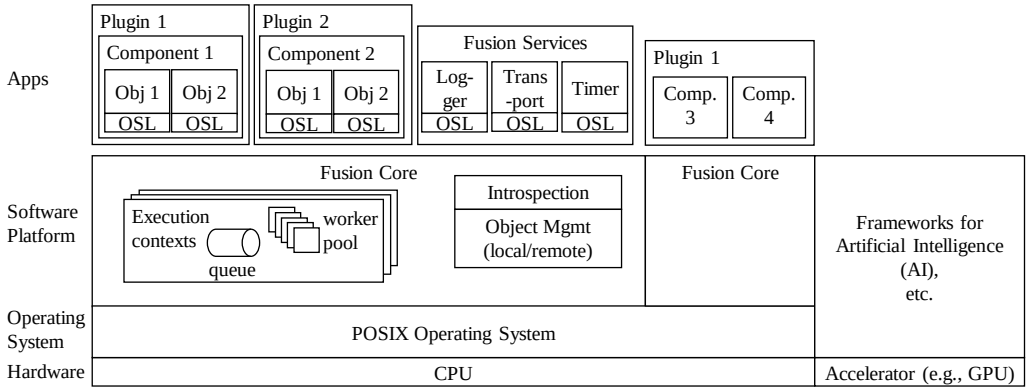


Figure 2 An example of a software system built with Fusion: Multiple identical instances of the Fusion engine (Fusion core) run in operating system processes and host components, which are implemented through Fusion objects (*Obj*). Execution contexts are used to process messages and services for users are provided. Artificial Intelligence (AI) frameworks already abstract accelerators and are running alongside Fusion.

operating system threads (worker pool). All events and method calls are enqueued in the queue of the execution context containing the receiving object and are processed in order of arrival by the worker(s). If the execution context is located in a different Fusion engine, the pluggable transport mechanism is used. Through this mechanism, objects can be local or remote to an engine. To the application, this is transparent.

As an additional layer of organization, components are introduced. These define interfaces, timing properties, etc. Objects are used to implement components. One to multiple objects or components are packed into plugins, which are stored as libraries and loaded by the Fusion core.

Each Fusion instance is configured through a manifest. These manifests define plugins to load, execution contexts to establish, objects to instantiate, as well as their configuration, among many others. Additionally, connections are explicitly configured in manifests. Connections originate from the event of one object and connect to the method of another object. This allows the passing of events and data, e.g., the change of a property, including the new value, between objects.

In addition to the means for communication, scheduling, management, etc., Fusion also provides a set of basic services, such as distributed Execution Management, Logging, Health Monitoring, Recording, Replay, Storage, etc. These services are implemented on top of Fusion with the same means and Application Programming Interfaces (APIs) as the applications.

Fusion is currently being used as a software platform for the development of autonomous driving functionality at the Autonomous Intelligent Driving GmbH (AID).

3 Challenges

Developing a new software platform for complex systems, such as autonomous driving, is a highly complex endeavor. This holds especially true, as safety and real-time behavior are critical aspects of the system under development. In the following, we will outline some challenges in the areas of complexity, performance, safety, and security that such a platform is facing.

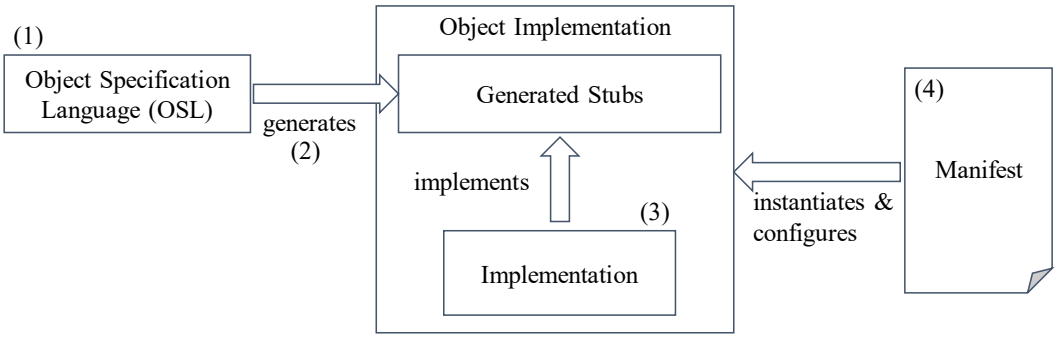


Figure 3 Process to develop and instantiate an object: (1) Specify in Object Specification Language (OSL), (2) generate stubs, (3) implement against generated stubs, (4) instantiate, configure and connect via manifest.

3.1 Complexity

While the introduction of the additional layers is increasing the development speed of applications significantly, while at the same time reducing the number of errors in development, it also introduces complexity. When building a system using Fusion and applications built on top of it, we have to consider Fusion objects, Fusion components, Fusion execution contexts, Fusion plugins, Fusion engines, Hypervisor partitions, physical devices, and potentially multiple systems (e.g., in a fleet of vehicles).

Managing this complexity is not trivial. Objects and components need to be instantiated, often multiple times. The correctness of connections defined in manifests needs to be ensured. This includes syntactical correctness, as well as semantical correctness, e.g., are expected and provided update frequencies, latencies, ASIL level matching, etc. Another aspect of correctness is security: Are all communication participants authenticated and allowed to communicate? While Fusion makes it much harder for developers to create deadlocks across threads, due to the messaging approach between components, it is still possible to create deadlocks on a higher layer, through the connection of objects. Also the creation of loops is possible, leading to large increase of data processing in the system. As Fusion introduces new abstraction levels and means of communication, as well as offering new usage patterns, new approaches to finding such problems have to be developed.

In terms of scheduling, the additional abstraction layers also need to be considered. Anything being scheduled in the system needs to consider execution contexts, the operating system scheduler, potentially hypervisors, etc.

Furthermore, with safety-critical systems and specifically autonomous driving, concepts for redundancy and fail-operational scenarios are required [3]. As in a level 4 self-driving vehicle, no safety driver is available to take over in case of a system failure, the system needs to recover from failures whenever possible [5].

In normal operation, as well as in such failures, the system needs to be ensured to operate correctly. This includes handling crashes of applications, and slow-down due to system load, among many other possible concerns.

Furthermore, being the platform that all functional applications are based on, Fusion needs to support a large selection of different target systems, which in turn can benefit from the abstractions provided by Fusion (e.g., for simulation). These include, but are not limited to vehicles, hardware-in-the-loop setups, multiple generations of different processor architectures, virtualized systems, developer laptops, cloud, etc.

3.2 Performance

When building a software platform for autonomous driving, high performance is key. The software system needs to be able to process large amounts of data from a large number of cameras, LiDARs, radars and other sensors with high bandwidth, at low latencies. A software platform needs to keep introduced overhead minimal to avoid slowing down the system, while at the same time requiring all features and benefits listed in Section 2.

Furthermore, due to the complexity and nature of autonomous driving, hundreds of developers in an agile setup are required. Their objects and components need to be correctly orchestrated and connected in the system. Any issues need to be detected as early as possible at design time or, if not otherwise possible, at runtime.

The large amount of input from developers to the system also means a large number of changes. Almost all parameters of the system can change up to hundreds of times per day. Many of these changes might require new computations (e.g., schedule computations). To keep development speed high, all computations caused by such changes need to take a minimum amount of time. To give an order of magnitude: One hour of computation is considered very long in this context.

3.3 Safety

Despite the increased complexity of the system, no reduction in safety can be permitted. This is in strong contrast to many other complex automotive software systems, the most complex of which today are found in the area of infotainment, where safety-requirements are often secondary, due to isolation. On the contrary, safety and reliability of the system need to increase when moving from a level 2 self-driving vehicle to level 4, where no driver is available to take over in case of a fault. Thus, ensuring real-time behavior, determinism and integrity of the system are paramount. This holds for all components, such as hardware, operating system and also software platform. However, as Fusion adds the new layers described in Section 2, new ways for guarantees have to be found. Fusion objects might be assigned to the same components and Fusion engine. Furthermore, execution contexts can be shared, as well as address spaces, when objects and components are running in the same Fusion engine.

Thus, a trade-off between the performance requirements as listed above, and the safety requirements need to be found.

3.4 Security

Similar to safety, security is an important topic for any connected vehicle. The same holds for autonomous vehicles. Some data in the system needs to be specifically secured (e.g., cryptographic keys). To fulfill the requirements set up in the General Data Protection Regulation (GPDR), additional technical measures need to be taken. These requirements do not only hold for the driver of the vehicle and the passengers, but also the pedestrians around the vehicle, potentially recorded on camera. To secure communication, certain, if not all objects and components need to be authenticated, certain traffic needs to be encrypted.

All of these measures add a potentially very large performance overhead. As in safety, balancing the performance and security requirements is key.

4 Design-Time Verification

In the automotive domain, like in many industries extensive testing is used to determine the correctness of a system. However, for such testing approaches, the running system is required. Additionally, there is no guarantee that all cases are covered when using testing.

For complex systems like autonomous driving, runtime testing on the system level is not efficient. Instead, wherever possible, the system should be described in a model at design time, where the consistency and correctness can also be checked. This can solve some of the complexity challenges listed above. Furthermore, optimizations can be performed based on the model (e.g., assignment of software to hardware components). Fusion already offers some means to describe the system at design time through OSL and manifests.

Whenever using modeling approaches, multiple challenges exist. The key is to find the right level of abstraction. If the model is too detailed, it is difficult to handle and check. If the model is too abstract, the benefit of modeling vanishes. Additionally, the model needs to be integrated into the development process to avoid divergence of model and implementation.

With a system as complex as an autonomous driving software stack, the model easily becomes very large. The additional layers introduced by Fusion add to this challenge. Due to the high frequency of changes and the checks needing to be performed on the model, computations need to take minimal time (see also Section 3.2). Furthermore, the model needs to be used by a large number of engineers from different backgrounds (e.g., software, safety, systems, security engineering). Without presentation of the model and tools according to the developers background, the acceptance and usability will be insufficient for a high development speed.

Last, but not least, a model and its tooling face challenges when developed alongside the software (and not upfront). To increase acceptance of the approach, it is essential to maximize the benefit and prioritize parts of the model. E.g., for a first Minimum Viable Product (MVP) of a software, modeling the exact resource consumption to the last bit in memory does not add value, while this might differ when moving closer to the release of, especially a safety-critical, software.

5 Conclusion

In this work, we presented Fusion, a new generation of software platform, lowering the size of atomic building blocks for complex autonomous driving systems. Furthermore, we raised a number of challenges, which are relevant for Fusion, as well as other, similar systems, targeting complex software systems, such as those used in autonomous driving.

In future, the fine-grained defined building blocks of Fusion will be enhanced to increase predictability, freedom from interference, as well as real-time behavior. Initial proofs of concepts have been built and development is ongoing. Fusion is actively being used for the development of level 4 autonomous driving in AID.

References

1 AUTOSAR Consortium. Specification of Communication Management, 2019. Document Identification No 717, Release R19-11.

2 AUTOSAR Consortium. Specification of RTE Software, 2019. Document Identification No 84, Release R19-11.

3 Philip Koopman and Michael Wagner. Challenges in autonomous vehicle testing and validation. *SAE Int. J. Trans. Safety*, 4:15–24, April 2016. doi:10.4271/2016-01-0128.

4 OPEN Alliance Special Interest Group. 100BASE-T1 System Implementation Specification, 2017. Version 1.0.

5 SAE International. Taxonomy and Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems J3016, 2018. Release 201806.

Adaptable Demonstrator Platform for the Simulation of Distributed Agent-Based Automotive Systems

Philipp Weiss
Technische Universität München, Germany
philipp.weiss@tum.de

Sebastian Nagel
Technische Universität München, Germany
s.nagel@tum.de

Andreas Weichslgartner
AUDI AG, Ingolstadt, Germany
andreas.weichslgartner@audi.de

Sebastian Steinhorst
Technische Universität München, Germany
sebastian.steinhorst@tum.de

Abstract

Future autonomous vehicles will no longer have a driver as a fallback solution in case of critical failure scenarios. However, it is costly to add hardware redundancy to achieve a fail-operational behaviour. Here, graceful degradation can be used by repurposing the allocated resources of non-critical applications for safety-critical applications. The degradation problem can be solved as a part of an application mapping problem. As future automotive software will be highly customizable to meet customers' demands, the mapping problem has to be solved for each individual configuration and the architecture has to be adaptable to frequent software changes. Thus, the mapping problem has to be solved at run-time as part of the software platform. In this paper we present an adaptable demonstrator platform consisting of a distributed simulation environment to evaluate such approaches. The platform can be easily configured to evaluate different hardware architectures. We discuss the advantages and limitations of this platform and present an exemplary demonstrator configuration running an agent-based graceful degradation approach.

2012 ACM Subject Classification Computer systems organization → Distributed architectures; Computer systems organization → Redundancy; Computing methodologies → Self-organization

Keywords and phrases fail-operational, graceful degradation, agent-based mapping

Digital Object Identifier 10.4230/OASIcs.ASD.2020.3

Funding With the support of the Technical University of Munich – Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763.

1 Introduction

With the introduction of new automotive functionality such as autonomous driving, automotive companies see themselves confronted with increasing customer needs and the demand to frequently deliver new functionalities. To cope with complexity, electronic architectures are undergoing major changes. Instead of adding a new electronic control unit (ECU) for each new functionality, software is being integrated on more powerful ECUs. We expect this consolidation trend to continue such that future electronic architectures will consist of only a few powerful ECUs, similar to consumer electronic devices [4].

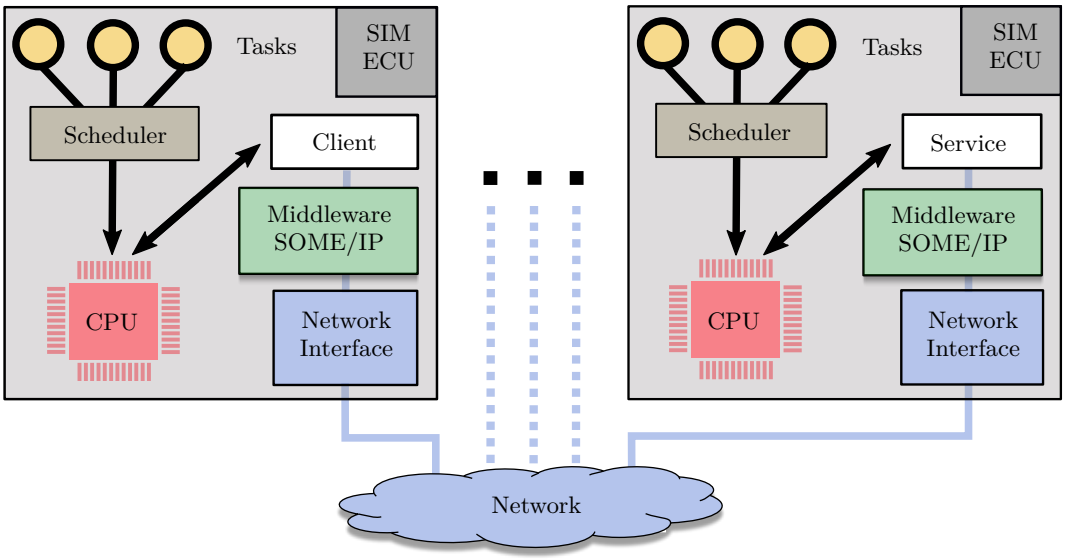


Figure 1 General structure of the simulator platform. The scheduler executes multiple tasks on the CPU. The tasks use the SOME/IP Middleware and the given network infrastructure to communicate. The platform simulates the ECUs instances and uses a physical network for communication.

As software is being decoupled from the hardware, future automotive software will be designed modular. This separation leads to new system-wide optimization possibilities. At the same time, new software functionality comes with higher resource demands and safety requirements. As there will be no driver as a fallback solution in autonomously driving cars, these systems have to be designed to maintain operation in the presence of critical ECU or software failures. However, such a fail-operational design requires redundancy which imposes an even higher resource demand and therefore cost. An approach to lower cost is graceful degradation where hardware resources which were formerly used by non-critical applications are repurposed for the use of critical applications.

With highly customizable software and unique customer configurations, each software system has to be optimized individually. Furthermore, users will change the configuration frequently and the software requirements might vary with each software update. Thus, design-time solutions to solve the mapping problem and optimize the system are insufficient as they would have to be re-evaluated for each change in the system. Therefore, new solutions are required that solve the mapping problem at run-time as part of the software platform. We presented such an agent-based approach in [5], which ensures fail-operational requirements at run-time using graceful degradation and is able to reconfigure the system after an ECU failure.

To be able to evaluate such emerging approaches, we present an adaptable demonstrator platform in Section 2 that is built on top of a distributed simulation environment. Users of the distributed simulation environment can configure computing components as distinct ECU instances to simulate task execution. For communication a physical network infrastructure can be used. This approach combines the advantages of an easily configurable simulation with a more experimental evaluation. The presented demonstrator platform is easily adaptable to different hardware configurations and provides a fast and accurate way to evaluate agent-based approaches for distributed systems. We present an exemplary demonstrator setup which runs our agent-based graceful degradation approach from [5] in Section 3 and discuss the scalability and limitations of the demonstrator platform in Section 4.

2 Demonstrator Platform

The general structure of our demonstrator platform is presented in Figure 1. We adopted a process-based Discrete-Event Simulation (DES) architecture and selected the SimPy framework [2] to simulate the ECU instances. Any ECU instance includes a task-scheduler to execute multiple tasks and a network interface to connect the ECUs with the deployed middleware. Using the middleware, the ECUs can subscribe to messages from other ECUs or publish messages themselves over the physical network. The system can be specified by the user according to our system model and uses the XML schema for specifications from the OpenDSE framework [3]. In the following sections, we describe the components of the demonstrator platform more precisely.

2.1 Simulation

On a simplified view, SimPy is an asynchronous event dispatcher [2]. SimPy places all events in a heap data structure. The events are then retrieved and dispatched in an event loop one after the other at discrete points in time. In SimPy's default mode the simulations advance as fast as possible, depending on how much computation power is available. As our demonstrator platform uses physical networks, a real-time simulation is required for realistic results. SimPy's real-time mode enforces the time steps of the simulation to be synchronized with the clock of the operating system: Each step of the simulation amounts to a certain amount of real time.

2.2 Task execution

Our software is modelled by independent non-critical and safety-critical applications. Each of the applications might consist of multiple tasks. The communication between the tasks is modelled with messages. Tasks in the system are triggered by incoming messages or periodically by a timer in case they are the anchor task of an application. The access to the CPU is granted by an exchangeable scheduler. The execution time on the CPU is then simulated by passing time according to the resource consumption of the task. Once a task has finished execution, it sends out messages via the network interface.

2.3 Middleware

Our framework uses a middleware, which handles the communication and which is based on SOME/IP [1], an automotive middleware solution. Such a middleware is necessary to enable the dynamic behaviour for moving tasks at run-time on the system. Tasks and agents exclusively communicate via this middleware and are modelled as services and/or clients. The middleware includes a decentralized service-discovery to dynamically find offered services at run-time. Furthermore, a publish/subscribe scheme is used to configure which data is sent between services and clients. Tasks that have outgoing edges in the application graph G_a are offered as a service whose events can be subscribed by the clients with incoming edges. In addition, remote procedure calls can be used for non event-based communication.

2.4 Communication

The SOME/IP protocol specification recommends UDP as a transport layer protocol for cyclic data and for hard latency requirements in case of errors [1]. Therefore to receive messages from the underlying network, we use a multi-threaded UDP Server. The UDP

Figure 2 Our demonstrator setup consisting of 4 Raspberry Pis, which are connected over a switch and Ethernet links. Each of the hardware nodes is running a single simulation instance with our agent-based graceful-degradation approach from [5]. The graphical user interface presents the simulated CPU utilization and the status of the agents and tasks.

server does not handle the incoming messages synchronously but instead dispatches a handler thread whenever it receives a new message. The handler thread post-processes the message and schedules the message in the SimPy simulation of the ECU. The correct scheduling of the incoming message is essential to avoid time deviations in the communication between the ECUs. To transmit data on the link-level, the sending entity serializes the data upfront in the network interface, and the receiving entity deserializes the data in the receive handler.

3 Case Study

We implemented our platform with the 4 simulation instances $HC1 - HC4$, each running on a dedicated Raspberry Pi 4, which are depicted in Figure 2. The Raspberry Pis are connected via an Ethernet link to a central switch. The graphical user interface presents the simulated CPU utilization and the status of agents and tasks on the corresponding simulation instance.

The presented example is running our agent-based graceful degradation approach from [5]. The system, which consists of a safety-critical task $FC1$ and seven non-critical tasks $FC2 - FC8$, depicts the situation after the failure of simulation instance $HC1$ and the first degradation. The safety-critical agent is marked with blue, while its passive task agent is marked with grey.

In the first phase of our approach, each task gets assigned to an agent that allocates required CPU and link resources for it, migrates to the corresponding ECU and starts the task. This ramp-up can be observed with the CPU utilization at the beginning of the simulation.

To ensure fail-operational behaviour, our approach uses passive redundancy combined with graceful degradation. In a critical ECU failure scenario, the passive redundant tasks of safety-critical applications can be reactivated. Using graceful degradation, safety-critical

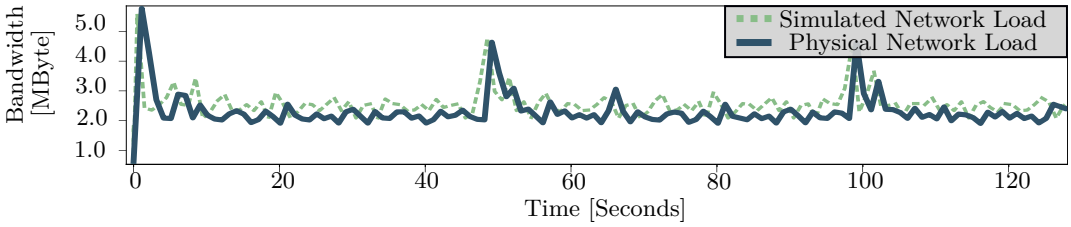


Figure 3 Measurements of the physical and simulated network traffic to and from a simulated ECU. The simulated link traffic and the physical link traffic show almost identical characteristics.

tasks can then take over the resources of non-critical tasks, which have to be shut down. As it has to be ensured that sufficient resources are available prior to an ECU failure, the agents of the passive tasks can reserve resources at the ECU and the agents of non-critical tasks. The reservation process ensures that it can be predicted if sufficient resources are available prior to a failure scenario. In our example the agent of $FC1$ has cloned itself at the beginning of the simulation and the corresponding passive task agent then reserved the resources at the agent of task $FC4$. After the failure of $HC1$, the passive task agent on $HC2$ detected the failure by a heartbeat timeout, claimed its resources at the agent of task $FC4$ and reactivated $FC1$. The degradation can be observed as $FC4$ is marked with red and the task status 'deactivated'.

After an ECU failure and the immediate failure reaction, the fail-operational behaviour of the safety-critical tasks can be re-established. The advantage of the agent-based approach is that no additional algorithm is required for the reconfiguration and the same mapping procedure can be repeated. In our example, after the immediate failure reaction on $HC2$, the task agent of $FC1$ cloned itself again to re-establish its fail-operational behaviour. The corresponding passive task agent is marked with grey on $HC4$. The agent status 'active' indicates that it has successfully reserved its resources and the system is able to endure another ECU failure without loosing its safety-critical functionality.

4 Discussion

We have compared simulated network traffic from our framework described in [5] with the physical network traffic of the demonstrator platform (Figure 3). The results validate that the previously simulated network is realistic and comes close to what we observe when using real network infrastructure. However, the simulated and observed behaviour is not entirely the same. For our future and current studies, the more realistic approach ensures that we do not miss any critical network behaviour, which we did not capture in our simulation.

The demonstrator platform is independent of the hardware, the operating system (OS) and the network architecture that the internet protocol suite (IP) uses on the link layer. It is extensible to any common computing platform. Also, the platform can simulate multiple ECU instances on a single hardware node in the network. The platform is, therefore, very flexible and scalable to assist in the rapid exploration and evaluation of distributed systems on various hardware setups and network topologies.

However, the scalability of the simulation is limited by the available computational power the nodes have: If a hardware node is not able to process its entire simulation workload within the foreseen real-time interval, the real-time constraint is violated leading to simulation delays.

5 Conclusion

In this paper we have introduced an adaptable demonstrator platform where multiple simulation instances can be used to simulate a distributed automotive system. Using a physical network, the platform combines the advantages of a rapid simulation configuration with a more realistic communication. The simulation instances provide a middleware for communication and a task execution model for computation. An exemplary implementation has been presented which runs an agent-based graceful degradation approach to achieve fail-operational behaviour. As long as the real-time factor of the simulations can be met, the demonstrator platform can be used to easily demonstrate emerging decentralized approaches on different hardware architectures.

References

1 AUTOSAR. *SOME/IP Protocol Specification R19-11*. URL: `https://www.autosar.org/fileadmin/user_upload/standards/foundation/19-11/AUTOSAR_PRS_SOMEIPProtocol.pdf`.

2 Ontje Lünsdorf and Stefan Scherfke. *SimPy Discrete Event Simulation Library for Python, Version 3.0.9*. URL: `https://simpy.readthedocs.io`.

3 Felix Reimann, Martin Lukasiewycz, Michael Glaß, and Fedor Smirnov. *OpenDSE – Open Design Space Exploration Framework*, 2019. URL: `http://opendse.sourceforge.net/`.

4 Selma Saidi, Sebastian Steinhorst, Arne Hamann, Dirk Ziegenbein, and Marko Wolf. Future automotive systems design: Research challenges and opportunities: Special session. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2018.

5 Philipp Weiss, Andreas Weichslgartner, Felix Reimann, and Sebastian Steinhorst. Fail-operational automotive software design using agent-based graceful degradation. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2020.

Agile Requirement Engineering for a Cloud System for Automated and Networked Vehicles

Armin Mokhtarian
Informatik 11 – Embedded Software, RWTH Aachen University, Germany
mokhtarian@embedded.rwth-aachen.de

Alexandru Kampmann
Informatik 11 – Embedded Software, RWTH Aachen University, Germany
kampmann@embedded.rwth-aachen.de

Bassam Alrifaee
Informatik 11 – Embedded Software, RWTH Aachen University, Germany
alrifaee@embedded.rwth-aachen.de

Stefan Kowalewski
Informatik 11 – Embedded Software, RWTH Aachen University, Germany
kowalewski@embedded.rwth-aachen.de

Bastian Lampe
Institute for Automotive Engineering, RWTH Aachen University, Germany
bastian.lampe@ika.rwth-aachen.de

Lutz Eckstein
Institute for Automotive Engineering, RWTH Aachen University, Germany
lutz.eckstein@ika.rwth-aachen.de

Abstract

This paper presents a methodology for the agile development of a cloud system in a multi-partner project centered around automated vehicles. Besides providing an external environment model as an additional input to the automation, the cloud system is also the main gateway for users to interact with automated vehicles through applications on mobile devices. Multiple factors are posing a challenge in our context. Coordination becomes especially challenging, as stakeholders are spread among different locations with backgrounds from various domains. Furthermore, automated vehicles for different applications, such as delivery or taxi services, give rise to a large number of use cases that our cloud system has to support. For our agile development process, we use standardized templates for the description of use-cases, which are initialized from storyboards and iteratively refined by stakeholders. These use-case templates are subsequently transformed into machine-readable specifications, which allows for generation of REST APIs for our cloud system.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases agile requirements engineering, cloud architecture, automated vehicles

Digital Object Identifier 10.4230/OASIcs.ASD.2020.4

Funding This research is accomplished within the project "UNICARagil" (FKZ EM2ADIS002). We acknowledge the financial support for the project by the Federal Ministry of Education and Research of Germany (BMBF).

1 Introduction

Individual vehicles become part of a larger system for many use cases proposed for automated driving. Automated shuttles, for example, would require an entity that coordinates a larger fleet of vehicles and through which humans can summon vehicles through their smartphone. Automated parcel delivery vehicles, another use case pursued in various efforts,

also depend on an entity for coordination. As depicted in Fig. 1, these coordinating entities can be implemented as cloud services, which then become an essential component in the ecosystem of automated vehicles. In particular, they provide the infrastructure which enables communication within the overall system. Different actors can store or request data and coordinate with other parties through the cloud system. Thus, the cloud provides the main interfaces for data exchange between actors, such as a user requesting a taxi by using a mobile application or an automated delivery vehicle asking for its next destination. Furthermore, cloud computing provides more computational power, which can be used to enable several other use cases like fusing real-time sensor data of multiple vehicles to provide feedback on a more comprehensive environment model [6]. Therefore, a cloud system is not only of high importance regarding communication but can also aid the overall ecosystem of automated vehicles by collecting, combining and processing data.

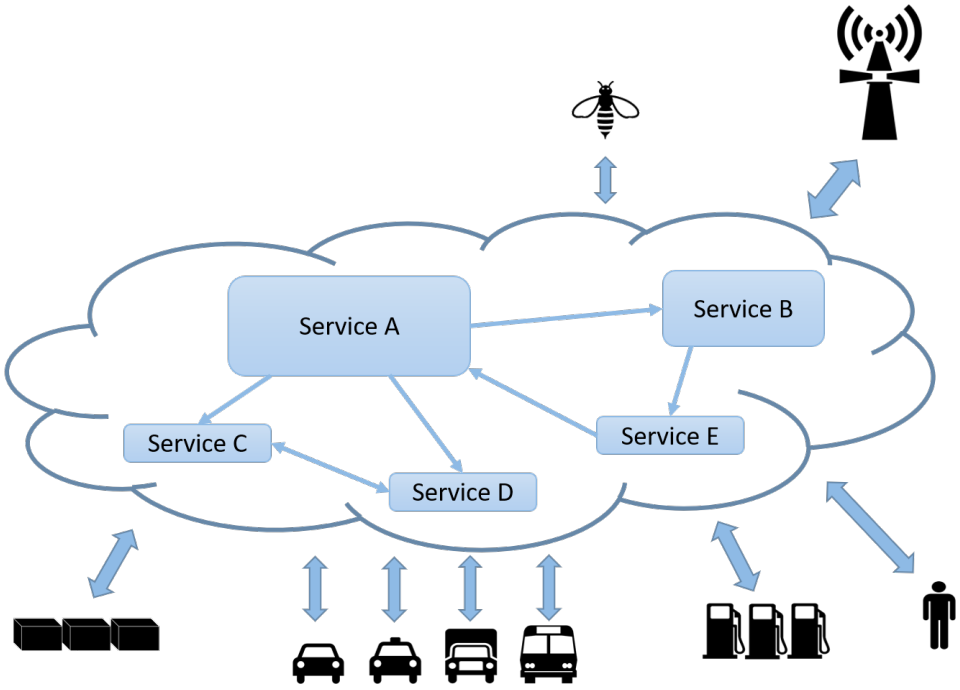


Figure 1 An overview of the cloud and its actors in the UNICARagil project. The ecosystem of UNICARagil includes four different vehicle types, a charging as well as a packing station, a control room and an info-bee (drone) [14].

The development of an ecosystem centered around automated driving is pursued in the UNICARagil project, which is a multi-partner project aiming to develop four fully automated vehicles of different characteristics [14]. As the vehicle on-board software is implemented following a service-oriented software architecture (SOA) [5], the cloud software also follows a SOA approach. A total of over 100 developers spread over 15 university chairs and 6 industrial partners are involved in this four-year project. Having a large number of participants, the communication and coordination of the development becomes extra challenging. First, because researchers and developers are spread among different locations. Second, because they have different academic backgrounds. This may result in numerous, non-uniform and poorly communicated specifications. Additionally, our cloud system has to serve four vehicle types which all share the same platform concept but have a different use.

- autoELF: A family vehicle intended for private use [12].
- autoTAXI: A vehicle for short-term general use.
- autoCARGO: A fully automated parcel service.
- autoSHUTTLE: Public transportation of multiple people.

The vehicle variants rely on a cloud system for coordination and for optional input to the automation algorithms through an external environment model. Although there are several commonalities between these vehicle types, they have different requirements for the cloud. The autoCARGO, for example, is the only platform that needs additional interfaces to communicate with a logistics management service. Hence, we have platform specific requirements in addition to requirements arising from automated driving scenarios. Furthermore, due to the early development stage, different platforms and a large number of developers with different backgrounds, the process from requirement engineering to implementation becomes very challenging. In this case, agile development is unavoidable. In particular, since the requirements change during the implementation, we need to simplify the overhead for adjustments.

Another challenging aspect arises since the cloud system and the project are developed simultaneously. In the early stages of a project, the requirements and specifications of the final system are often unavailable. Thus, the simultaneous development of the cloud and the overall system requires a scalable and adaptable system for being able to adjust to changing requirements. Consequently, this requires a modular, expandable and adjustable concept. The necessity of an adaptable cloud system increases with an increasing number of stakeholders. In this paper, we present our approach for the design of a cloud system to overcome the aforementioned challenges.

1.1 Related Work

Cloud systems have various use in the automotive domain. [7] claims, that cloud systems are the go-to solution for deploying frameworks suitable for automotive tasks. A cloud-based artificial intelligence framework for continuous training and self-driving is presented by [8]. Their system supports the collection of data which is used to develop and train machine learning models to leverage the cloud as a model performance booster. Hence, the cloud is an essential component in their ecosystem.

According to [3], the most common reason for software project failures is bad or incomplete requirements engineering. Therefore, requirement engineering plays a critical role in our approach. Several methods and guidelines were developed in order to not only prevent project failures but also to allow an efficient approach. A novel approach, which is investigated by Paetsch et al. [9], is to combine classic requirement engineering concepts, e.g. Waterfall model [1], with agile methods like Scrum [13].

2 Method

A system development lifecycle usually consists of the following steps. By starting with a *requirements analysis*, the developers investigate the properties and qualities their system should provide. At this phase, detailed communication with the customer is required to elaborate a solid groundwork. The *system design*, which includes the complete hardware and software setup is derived based on the requirements. Consequently, the system design is broken down into modules in the *architectural design* phase. In the *module phase*, these modules are designed in detail to be implemented afterwards.

If it turns out that the requirements were not levied correctly or changed during the development lifecycle, it may require to restart the development process from scratch in the worst case. Thus, in order to reduce the overhead, it is desired to have robust requirement engineering and an adaptable development process. Agile requirements engineering methods like Scrum are prepared for new or changing requirements as their incremental mode of operation intends to provide adaptability to the current set of tasks. Furthermore, a modular groundwork enables the basis for implementing new requirements.

In this section, we present a methodology, which relies on template-based requirement engineering. This method enables to adapt to new requirements with lower effort and thus provides the basis for an incremental way of working.

2.1 Requirements Engineering

In order to have consistent feedback, we created a survey with concrete questions, which was handed out to all project stakeholders. Moreover, to take care of the difficulty regarding the different academic backgrounds, we asked for use cases instead of operational requirements. To illustrate the interaction of the use cases as well as the actors within the system, UML diagrams can be used [10]. However, the number of attributes and details we want to capture would cause UML diagrams to appear confusing. Furthermore, due to a large number of participants, we could not assume that everyone is familiar with UML. Hence, we decided on a questionnaire with 8 attributes which are listed in Table 1 and adapted from [2]. This assures that the collected use cases all have the same format and enable easy extraction of the essential information. Our survey resulted in 47 use cases that had to be evaluated and processed.

Although we tried to get uniform results by providing a specific template, the survey showed that the participants answered the questionnaires with a varying level of detail. Thus, the next step of our method transforms the questionnaires into a uniform, understandable and transparent representation. For this purpose, as recommended by Christine Rupp [11], we investigated the results concerning completeness, consistency, understandability, necessity, feasibility, clarity and traceability. To assure this quality features, we decided to use a pattern-based approach shown in Fig 2. This approach maps a use case into four parts.

Table 1 One of 47 use cases gathered with the template-based survey in the UNICARagil project.

ID	3.5
Use Case Title	Deliver Notification
Description	Customer is notified about the delivery of his parcel.
Actors	Cloud, Customer (App), Parcel Box
Frequency	Daily
Condition	Parcel Box has booked delivered parcel correctly.
Guarantee of Success	Customer can pick up his parcel at the parcel box.
Trigger	Parcel was delivererd to the parcel box by autoCARGO.
Actions	1. Cloud notifies Customer via App about delivered parcel. 2. Customer enters pin code at parcel box. 3. Parcel box provides parcel. 4. Parcel box notifies the cloud about the parcel pickup status.

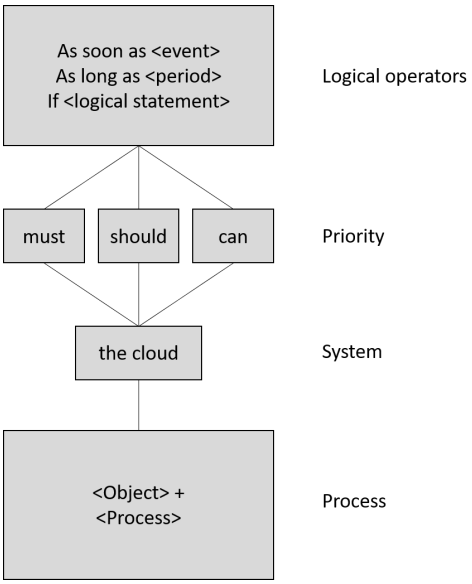


Figure 2 Pattern based approach for requirement description adapted from [11].

1. **Logical operator**
   Most of the functionalities and processes start after a series of preconditions or are triggered through events. Therefore, the temporal component is mapped here.
2. **Priority**
   Different use cases have different priorities for the same functionalities in the cloud. For being able to distinguish between the relevance of a requirement, a three-level rating system is introduced.
3. **System**
   Requirements that are created by this approach are phrased in an active sentence. Since these requirements are gathered in the context of the cloud only, it is always the subject.
4. **Process**
   The main focus of every requirement is the functionalities of the system. At this point, the desired system behavior is described.

After transforming 47 use cases into this pattern with regard to quality features, 42 requirements were derived.

2.2 System Design

After processing the gathered use cases into requirements, the next step consists of breaking down the cloud system into individual cloud services. We will use the exemplary use case *3.5 Deliver Notification* shown in Table 1 to explain the procedure. This use case describes the situation of an autoCARGO delivering a parcel to the parcel box. Afterwards, the user gets a notification via the App and receives a pin code that is needed to pick up the parcel at the parcel box.

In this case, there exist three actors: The user, the parcel box and the cloud. This scenario creates the need for two services to manage user and parcel box data. Hence, the *user management* and the *logistics management* are introduced. Any endpoint can use their API to store and request data. Now, we use the Actions from Table 1 to derive the necessary communication interfaces of the aforementioned cloud services. The resulting relation table is shown in Table 2.

Table 2 System view on an exemplary use case.

3.5 Deliver Notification			
Action	From	To	HTTP-API
3.5.1	Logistics Mgmt	User Mgmt	PUT:/parcel/status
3.5.2	App	User Mgmt	GET:/user/parcel/receive
3.5.3	Parcelbox	Logistics Mgmt	HEAD:/infrastructure/parcelbox/{PBOX_ID}/pincodecheck
3.5.4	Parcelbox	Logistics Mgmt	PUT:/infrastructure/parcelbox/{PBOX_ID}/parcel/{PARCEL_ID}/status

In the course of our process, we maintain traceability by linking use case questionaries, derived requirements and final system specifications with an ID. This allows for identifying components that may be affected by a change in the underlying requirements. Besides an ID, the system view contains Action, From, To and API.

For the first Action of this use case (3.5.1), the user management provides two endpoints. One is offered by the cloud internal HTTP API for the logistics management and the other one is offered by consumer mobile application HTTP API. We determined

PUT /parcel/status (1)

to be accessed by the logistics management for notifying the user management about parcel status changes. Further, we determined

GET /user/parcel/receive (2)

to be used by the App for requesting status information about parcels. Its response also provides the pin code for the parcel pickup. The next Action (3.5.2) does not describe any interaction with the cloud and can be skipped. Action 3.5.3, in turn, needs to be handled by

HEAD /infrastructure/parcelbox/pbox_id/pincodecheck (3)

in order to check if the combination of parcel and pincode is correct. Finally, Action 3.5.4 asks the system to update the status of the parcel which is done by

PUT /infrastructure/parcelbox/pbox_id/parcel/parcel_id/status (4)

This approach proved itself to be structured and expedient, as it allowed us to quickly come to an initial system design consisting of more than 20 cloud services with over 70 endpoints. The resulting HTTP-APIs are specified with OpenAPI[1] which is an API description format for REST APIs. The specifications are written in YAML and thus are readable to both humans and machines.

The YAML files are processed by the OpenAPI Generator to generate a Python flask server [4]. Furthermore, bootprint[2] is used to generate static HTML pages of the OpenAPI specification. We decided for this setup, in order to enable rapid prototyping during our development process.

[1] `https://swagger.io`
[2] `https://github.com/bootprint/bootprint`

2.3 Code Generation

Since OpenAPI was used to specify the HTTP API for every service, it is possible to convert those into a static HTML page by using bootprint. Furthermore, those specifications are used to generate a Python flask server with code stubs for the implementation of all HTTP APIs. This code does not contain the actual functionality of the service but is runnable and allows for speeding up the implementation process. For this purpose, a Git repository with GitLab Continuous Integration was set up. Tagging a commit with generate_code, generate_doc or generate when pushing to the master branch, the Git runner is triggered. This generates the HTML documentation, Python flask servers or both for all services. Finally, everything just generated is pushed to the Git repository. Hence, for every change made to the specification, the code is generated automatically and can be tested directly.

3 Conclusion

Based on the Cloud system in the UNICARagil project, we presented a methodology to deal with arising challenges in a multi-partner project. Together with a large number of different stakeholders, covering four different platforms makes the development challenging. Thus, our method was chosen to provide consistent and scalable requirements engineering. Furthermore, due to the early stage within the overall project, it was of high importance to use a method which provides fast prototyping and is adaptable to upcoming changes in the requirements

Therefore, we first created a template-based survey that was handed to everybody who intends to work with the cloud. The survey returned 47 use cases which then were examined regarding quality measurements and processed afterward. A pattern-based approach helped to transform the use cases into 42 requirements. Based on these requirements, we derived relation tables to provide a structured way to define interfaces and endpoints. The interfaces were specified with OpenAPI, which enabled the generation of Python flask server code and an HTML documentation. Both code and documentation then are pushed to Git and allows for automated test with GitlabCI[3].

We presented a method that allows processing new requirements into the system with lower effort to allow an incremental way of operation. At the same time, a modular and lightweight groundwork derived by the template-based requirements engineering enabled fast prototyping. This methodology combined classic requirements engineering and agile development. Neither of both would have suited our needs if they were used separately. Our methodology proved itself as suitable for our project since it allowed fast prototyping and easy adaptability to new requirements, while still capturing system specifications from the beginning of the project.

References

1 Barry W Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

2 Alistair Cockburn. Basic Use Case Template. *Humans and Technology, Technical Report*, 96, 1998.

3 Jeremy Dick, Elizabeth Hull, and Ken Jackson. *Requirements engineering*. Springer, 2017.

[3] `https://docs.gitlab.com/ee/ci/`

4 Miguel Grinberg. *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.", 2018.

5 A. Kampmann, B. Alrifaee, M. Kohout, A. Wüstenberg, T. Woopen, M. Nolte, L. Eckstein, and S. Kowalewski. A dynamic service-oriented software architecture for highly automated vehicles. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 2101–2108, October 2019. doi:10.1109/ITSC.2019.8916841.

6 Bastian Lampe, Timo Woopen, and Lutz Eckstein. Collective driving-cloud services for automated vehicles in unicaragil. In *28. Aachen Colloquium Automobile and Engine Technology: October 7th–9th, 2019, Eurogress Aachen, Germany / scientific management: Univ.-Prof. Dr.-Ing. Lutz Eckstein, Univ.-Prof. Dr.-Ing. Stefan Pischinger ; organizational management: Michaela Heetkamp (M.Sc.), Jonas Müller (M.Sc.) ; organized by: Institute for Automotive Engineering, RWTH Aachen University; Institute for Combustion Engines, RWTH Aachen University.*, pages 677–703. Institute for Automotive Engineering, RWTH Aachen University, 2019. doi:10.18154/RWTH-2019-10061.

7 Andre Luckow, Matthew Cook, Nathan Ashcraft, Edwin Weill, Emil Djerekarov, and Bennie Vorster. Deep learning in the automotive industry: Applications and tools. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3759–3768. IEEE, 2016.

8 Cristian Olariu, Haytham Assem, Juan Diego Ortega, and Marcos Nieto. A cloud-based ai framework for machine learning orchestration: A "driving or not-driving" case-study for self-driving cars. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1715–1722. IEEE, 2019.

9 Frauke Paetsch, Armin Eberlein, and Frank Maurer. Requirements engineering and agile software development. In *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.*, pages 308–313. IEEE, 2003.

10 Bernhard Rumpe. *Modellierung mit UML.* Xpert.press. Springer Berlin, 2nd edition edition, September 2011.

11 Christine Rupp et al. *Requirements-Engineering und-Management: Aus der Praxis von klassisch bis agil.* Carl Hanser Verlag GmbH Co KG, 2014.

12 Tobias Schräder, Torben Stolte, Inga Jatzkowski, Robert Graubohm, and Markus Maurer. An approach for a requirement analysis for an autonomous family vehicle. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1597–1603. IEEE, 2019.

13 Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

14 Timo Woopen, Bastian Lampe, Torben Böddeker, Lutz Eckstein, Alexandru Kampmann, Bassam Alrifaee, Stefan Kowalewski, Dieter Moormann, Torben Stolte, Inga Jatzkowski, Markus Maurer, Mischa Möstl, Rolf Ernst, Stefan Ackermann, Christian Amersbach, Hermann Winner, Dominik Püllen, Stefan Katzenbeisser, Stefan Leinen, Matthias Becker, Christoph Stiller, Kai Furmans, Klaus Bengler, Frank Diermeyer, Markus Lienkamp, Dan Keilhoff, Hans-Christian Reuss, Michael Buchholz, Klaus Dietmayer, Henning Lategahn, Norbert Siepenkötter, Martin Elbs, Edgar v. Hinüber, Marius Dupuis, and Christian Hecker. UNICARagil – Disruptive Modular Architectures for Agile, Automated Vehicle Concepts; 1st edition. In *27. Aachener Kolloquium Fahrzeug- und Motorentechnik : October 8th–10th, 2018 – Eurogress Aachen*, pages 663–694, Aachen, October 2018. 27th Aachen Colloquium Automobile and Engine Technology 2018, Aachen (Germany), 8 Oct 2018 – 10 Oct 2018, Aachener Kolloquium Fahrzeug- und Motorentechnik GbR. Zweitveröffentlicht auf dem Publikationsserver der RWTH Aachen University. doi:10.18154/RWTH-2018-229909.

BreachFlows: Simulation-Based Design with Formal Requirements for Industrial CPS

Alexandre Donzé
Decyphir SAS, Moirans, France
http://www.decyphir.com
alex@decyphir.com

Abstract

Cyber-Physical Systems (CPS) are computerized systems in interaction with their physical environment. They are notoriously difficult to design because their programming must take into account these interactions which are, by nature, a mix of discrete, continuous and real-time behaviors. As a consequence, formal verification is impossible but for the simplest CPS instances, and testing is used extensively but with little to no guarantee. Falsification is a type of approach that goes beyond testing in the direction of a more formal methodology. It has emerged in the recent years with some success. The idea is to generate input signals for the system, monitor the output for some requirements of interest, and use black-box optimization to guide the generation toward an input that will falsify, i.e., violate, those requirements. Breach is an open source Matlab/Simulink toolbox that implements this approach in a modular and extensible way. It is used in academia as well as for industrial applications, in particular in the automotive domain. Based on experience acquired during close collaborations between academia and industry, Decyphir is developing BreachFlows, and extension/front-end for Breach which implements features that are required or useful in an industrial context.

2012 ACM Subject Classification Software and its engineering; Computer systems organization → Embedded and cyber-physical systems; Theory of computation → Timed and hybrid models; Theory of computation → Streaming models; Mathematics of computing → Solvers; Computing methodologies → Model verification and validation; Computing methodologies → Simulation evaluation; Computing methodologies → Simulation tools; Computing methodologies → Machine learning; Software and its engineering → Software creation and management; Theory of computation → Mathematical optimization

Keywords and phrases Cyber Physical Systems, Verification and Validation, Test, Model-Based Design, Formal Requirements, Falsification

Digital Object Identifier 10.4230/OASIcs.ASD.2020.5

Category Extended Abstract

1 Context: CPS Design, Verification and Validation

Cyber-Physical Systems (CPS) such as cars, planes, robots, medical devices, etc, have been steadily growing in sophistication and complexity. As a consequence, their construction require advanced design tools to ensure that they achieve their functional and safety goals. Model-based design (MBD) has become a standard practice to cope with the complexity and cost of development. In this paradigm, models of the system and its environment of increasing realism are developed and iteratively verified and validated against a suite of requirements until the real or production design is achieved. Stemming from the domains of logic, computation, digital circuits and later software verification, formal methods were developed to automate the process of proving that a given design satisfy a given requirement (Model-Checking) or alternatively, creating a design from a given requirement, which is then proven to be satisfied by construction (synthesis). For decades, various attempts have been made to bring these approaches to MBD for CPS. However, *proving* requirements in

the strictly formal sense, is possible only for the simplest CPS models, e.g., finite state machines, or in some cases timed automata. Most mathematical and computational models for CPS are so-called *hybrid systems*, mixing non-linear continuous and discrete dynamics. For these, existing formal methods do not scale well in general. This is the case for examples of most models created with modeling frameworks such as Mathworks tool Matlab/Simulink [6], which is ubiquitous in the industry. These frameworks are used by engineers mostly for simple testing using simulation. They sometimes implement formal methods but their application is restricted to simple models or small components.

In the last decade or so, an intermediate approach between simple testing and formal methods has emerged. On one side, it is still based on simulations; but as simulator technology has progressed, it has become easier to produce large amount of simulation data, making it possible to perform different types of analysis such as statistical (a la Monte-Carlo), guided search, learning, etc. On the other side, this approach retain some characteristics of formal methods, e.g., the use of formal specifications languages such as temporal logics and their quantitative semantics. One popular example is falsification. Given a system S with inputs u and a formal requirement φ, its goal is to find some input u^* such that the behavior of S using u^* falsifies or violates φ. The most common approach to solve this problem makes use of numerical black-box optimization and quantitative semantics. The satisfaction of φ can be estimated by a function $u \mapsto \rho(\varphi, S(u))$ where $\rho(\varphi, S(u)) < 0$ implies that u falsifies φ. Therefore looking to minimize $J(u) = \rho(\varphi, S(u))$ can lead to finding an input u^* such that $J(u^*) < 0$, meaning that u^* is a falsifying input. Conversely, if $J(u^*) \geq 0$ can be proven to be a global minimum, then we have proven that φ is always satisfied by S.

2 Breach Features Overview

The falsification concept and core ideas can be described in a few words but its application in practice can be much more daunting. Breach [3] is an open source Matlab/Simulink toolbox that implements the required ingredients in such a way that each one can be dealt with in a modular and reusable way, thus applying a separation of concern approach. In the following, we use the automatic transmission system pictured in Figure 1 to describe and briefly illustrate these components. They can be broadly categorized as follows:

- **Interfaces**, which define which signals in the models are inputs and outputs for Breach. In our example, throttle and brake are inputs, RPM, gear and speed are outputs for the model, but Breach can also monitor internal signals such as the OutputTorque or ImprellerTorque. Various parameters can also be part of an interface.
- **Input generators**, which define the search space or variable domain for the inputs. E.g., we might consider steps, pulses, piecewise-linear signals, etc. An input generator is often responsible for converting infinite domains (dense time, real-valued set of signals) into finite sets of variables suitable for an optimization problem. For example, if throttle is chosen to be a step signals going from zero to some value, then only two variables are enough to define the throttle signal at all times t: the time of the step and and its amplitude.
- **Requirements**, which define formally the requirements to be falsified. Breach supports Signal Temporal Logics (STL) [7], a formal specification language adapted for CPS. An example of a requirement easily expressed in STL is the following: φ= "whenever the car is in gear 4, the speed is above 30 miles per hour." Breach implements the efficient quantitative monitoring algorithm of [4], so that computing the quantitative satisfaction of a requirement is generally a negligible overhead compared to computing a simulation of the system.

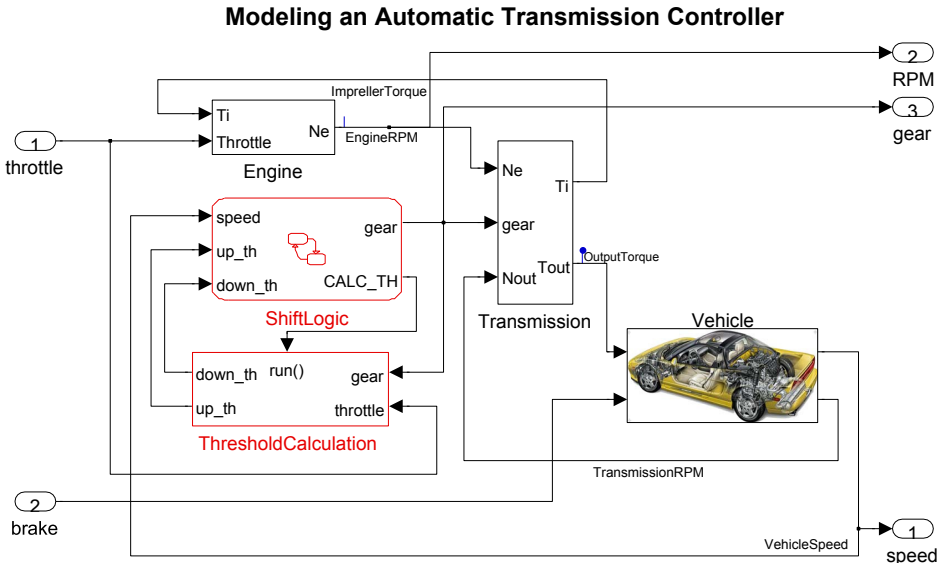


Figure 1 Automatic transmission system. This model simulates the behavior of an automatic gearbox as a function of throttle and braking from the driver.

- **Solvers**, which define the automated strategies that will solve the underlying optimization problem defined to find a falsifying input for the requirement. Solver go from "barbaric" random searches to genetic algorithm, local search with gradient estimation, etc.

Breach provides a collection of classes for each of these components (interfaces for data and Simulink models, basic input generators, STL, and optimization solvers) but one of its design goals was that this collection could be also extensible with reasonable ease. Examples of such extensions can be found in [5], where Breach is interfaced with a driving simulator written with the Unity engine and a neural network controller in Python; in [1], a different quantitative semantics for STL is implemented; in [2] a new stochastic local search solver is implemented; etc.

3 BreachFlows: An Engineer Friendly Interface to Breach

When applying falsification methods (or similar simulation-based approaches with formal requirements) in an industrial context, one is faced with the following challenges:

- **Models complexity, heterogeneity and changeability.** As one may expect, industrial-scale models are typically larger and more complex than theoretical/academic ones. The first consequence is that simulation has a higher computational cost so that performing as few simulations as possible is of paramount importance. Furthermore, a Simulink model can typically embed legacy code, or co-simulate with a different external simulator, import data or post-process outputs. Finally models are often under constant development by different engineers or teams of engineers so that new version with changes can be pushed frequently.
- **Abundance of requirements and lack of *formalized* requirements for inputs (test cases) and outputs.** Requirements for a given system design are about as complex and heterogeneous as models, but they are also typically informal and qualitative.

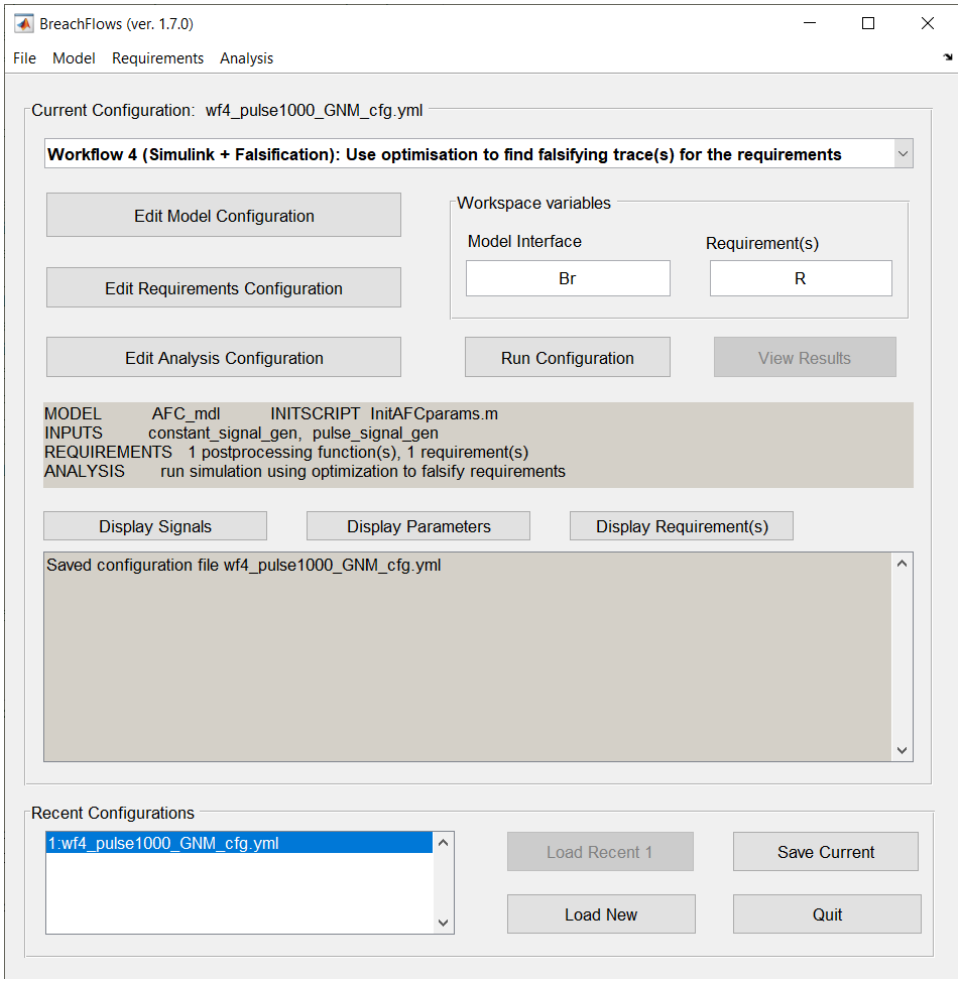


Figure 2 Top level GUI of BreachFlows.

Furthermore, although formalism such as (signal) temporal logic are languages that relatively simple in terms of their syntax and core semantics, they can be tricky to use properly. What is worse, engineers are rarely trained to use them and often reluctant or not able to invest time in learning them. Test cases used for testing those requirements are also typically under-specified and custom-made at the discretion of the developers or test engineers.

BreachFlows was developed on top of Breach to address these specific problems. It is essentially a user friendly front-end for Breach which can be used to build requirements sets and setup and maintain falsification problems and other types of analysis, so that they can be applied iteratively to support a CPS design at all stages of developments in a consistent way. Several features and characteristics and how they try to answer to the various challenges described above are the following:

- **Configuration management:** at the top level, BreachFlows is a GUI creating and managing configuration files for Breach typical workflows. They are clearly divided into three sections: models or data, requirements, and analysis, as illustrated on Figure 2. Sections can be imported from configuration to another, and they are designed to be robust to model or requirement changes, so that, e.g., a small change in the model can be reflected by a small change in a corresponding configuration;

- **Various features helping with high cost of simulation and heterogeneity:**
  - Use of parallel computation when possible;
  - System of efficient disk caching mechanism to reload previously computed simulations with the same parameters;
  - Use of custom scripts and functions (user-defined callbacks) for model initialization, inputs and requirements;
  - A mechanism of pre-conditions on input signals with constraints possibly expressed in STL so that whenever a test case or input is invalid, the corresponding simulation is skipped;
- **Input and requirement builders** consisting of a GUI pre-loaded with a collection of parameterized templates. Requirement templates are expressed in structured plain English which are mapped to STL formulas.

Breach is available as open source at `https://github.com/decyphir/breach` and BreachFlows is available on request at `info@decyphir.com`.

References

1 Koen Claessen, Nicholas Smallbone, Johan Liden Eddeland, Zahra Ramezani, Knut Åkesson, and Sajed Miremadi. Applying valued booleans in testing of cyber-physical systems. In *MT@CPSWeek*, pages 8–9. IEEE, 2018.

2 Jyotirmoy V. Deshmukh, Xiaoqing Jin, James Kapinski, and Oded Maler. Stochastic local search for falsification of hybrid systems. In *ATVA*, volume 9364 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2015.

3 Alexandre Donzé. Breach, A toolbox for verification and parameter synthesis of hybrid systems. In *Proc. of CAV 2010: the 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 167–170. Springer, 2010.

4 Alexandre Donzé, Thomas Ferrère, and Oded Maler. Efficient robust monitoring for STL. In *Proc. of CAV 2013: the 25th International Conference on Computer Aided Verification*, volume 8044 of *LNCS*, pages 264–279. Springer, 2013.

5 Tommaso Dreossi, Alexandre Donzé, and Sanjit A. Seshia. Compositional falsification of cyber-physical systems with machine learning components. *J. Autom. Reasoning*, 63(4):1031–1053, 2019. `doi:10.1007/s10817-018-09509-5`.

6 Mathworks Inc. Test generated code with sil and pil simulations. Cf. `https://www.mathworks.com/help/ecoder/examples/software-and-processor-in-the-loop-sil-and-pil-simulation.html`.

7 Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS/FTRTFT*, pages 152–166, 2004.